Security Requirements for the Prevention of Modern Software Vulnerabilities and a Process for Incorporation into Classic Software Development Lifecycles

Lee M. Clagett II

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Applications

James D. Arthur, Chair
Randolph C. Marchany
Osman Balci

December 14, 2009
Blacksburg, Virginia

Keywords: Software Security, Vulnerabilities, Requirements, Constraints, Assumptions, Access Driven VV&T

Security Requirements for the Prevention of Modern Software Vulnerabilities and a Process for Incorporation into Classic Software Development Lifecycles

by

Lee M. Clagett II

Abstract

Software vulnerabilities and their associated exploits have been increasing over the last several years - this research attempts to reverse that trend. Currently, security experts recommend that concerns for security start at the earliest stage possible, generally during the requirements engineering phase. Having a set of security requirements enables the production of a secure design, and product implementation. Approaches for creating security requirements exist, but all have a similar limitation - a security expert is required.

This research provides a set of software security requirements that mitigate the introduction of software vulnerabilities, and reduces the need for security expertise. The security requirements can be implemented by software engineers with limited security experience, and be used with any computer language or operating system. Additionally, a tree structure, called the software security requirements tree (SSRT), is provided to support security requirement selection, based on project characteristics. A graphical interface for the SSRT is provided through a prototype Java tool, to support the identification and selection of appropriate software security requirements.

This research also provides a set of security artifacts to support a comprehensive verification, validation, and testing (VV&T) strategy. Those artifacts are generic, and represent design and implementation elements reflecting software security requirements. The security artifacts are used in verification strategies to confirm their necessity and existence in the actual design and implementation products.

Acknowledgements

I received helped from numerous groups of people that must be mentioned for their hard work and efforts in helping me achieve my goal. I would like to thank the following people:

- My advisor Dr. James D. Arthur, for having unlimited patience, and providing continual direction throughout the entire work.
- The entire committee, James Arthur, Randolph Marchany, and Osman Balci for providing support, concerns, and ideas along the way.
- The resident security expert Randolph Marchany, who endured many iterations of requirements and theories.
- The original brainstorming committee who spent countless hours attempting to help me solve the requirements generation problem: Mahima Gopalakrishnan, Yannick Verdie, Song Huang, and Dr. Pardha Pyla.
- Special thanks to Dr. Pardha Pyla for supplying the idea that led to the SSRT, and suggesting the idea to conduct a subject matter expert evaluation to validate the work.
- I need to thank the security experts for spending so much time to review the long list of requirements: Dr. Matt Bishop, Preston Chrisman, and Paul Schmehl.
- Lastly, all the software engineers that took their precious time to evaluate the requirements: Matthew Dunlop, John Paul Dunning, Tannous Frangieh, Edward Frazier, Michael Gora, William Klinefelter, Phillip Kobezak, Ben Moyers, Sean Ponce, Theresa Nelson Robinson, and David Shelly.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

The prevalence of computers has continued to increase since their introduction, and the Internet revolution has sped this increase by making individual computers nearly a requirement for anyone wishing to conduct modern business. This has also increased the reliance that individuals, businesses, and governments place on computers daily, and the data they protect. Computers store sensitive information, monitor and control critical infrastructure components, and are relied upon for business applications. The importance of generating bug-free code has been realized for decades because incorrect software costs money to those that rely on the operation of that software. The high Internet connection rate among modern computers means the impact of security is now greater, because bugs in software can be exploited by a remote machine on the network. These exploits create problems in modern society, from identity theft to potential terrorism possibilities [50]. Any software executing on a computer with Internet access has remote vulnerability potential, even small desktop applications.

The reliance on computing has put software in control of the operation of critical and personal applications, which often execute on a computer with an Internet connection. This places an importance on writing secure software applications, in particular software involving safety or infrastructure management. The results of a compromised machine at the infrastructure level can be catastrophic, recently hackers broke into PBX telephone exchange systems to steal $55 million in phone-calls worldwide [55]. The importance of security cannot be underestimated, and any current software project must consider security during development. Future technologies will continue to require software that is bug-free, and highly secure software.

As the use of software increased, so has the number of security incidents originating from software faults. The total number of vulnerabilities cataloged by CERT has increased from 216 in 1998 to 6,058 through the first three quarters in 2008 [24]. Any security incident relating from a software fault is unacceptable. Future software is not expected to contain zero security vulnerabilities, but it is expected that the developers of future software take precautions to evaluate security implications. In 2002, NIST reported that up to $59.5 million is lost annual to faulty and insecure software, but with improvements to infrastructure the losses could be lowered to $22.2 million [66]. These cost savings along with threats of information theft provide the incentive for software vendors to take their security development processes seriously.

Our current reliance on software is unquestionable, and resources are required to face the growing issue of software security, particularly because the amount of money made in exploiting software is growing. Although progress has been made identifying and solving some of the problems that security poses to modern software, work in security is still in its infancy. Secure software education is lagging behind, leaving many modern programmers unable to identify key, but crucial areas that need attention and focus.

1.1 Secure Development Problem

Original security analysis was conducted through a penetrate-and-patch approach [47]. Software penetration testing is an approach conducted during the testing and deployment phases of software development, where a system is subjected to rigorous testing in hopes of discovering vulnerabilities. A vulnerability is a condition that permits software to move to an incorrect state [15]. If a vulnerable condition can be manipulated by an adversary to force software to an incorrect state, the system is said to have an exploit. For an exploit to be possible, a vulnerable condition must be present, and a method to manipulate the software into an incorrect state must be available. Any vulnerability discovered in software is considered dangerous, because the possibility of an exploit exists. During penetration testing, once a vulnerability is discovered within the software, a patch for the vulnerability is developed to prevent the possibility of an exploit. A major fault with this approach is the reactive rather than proactive nature. This is the same reason that previous work in software development processes have pushed for more upfront analysis to reduce bugs. Furthermore, given the limited time for penetration testing before deployment, generally only a small number of vulnerabilities are found [4].

The lack of successful vulnerability detection before deployment has generated an increase in secure software research. Research first targeted better methods for testing security, but eventually has moved to fully incorporating security in the software development lifecycle [27] [37] at the suggestion of security experts in 1998 [47]. Proposals to include security in the development lifecycle have been given because past studies on requirements engineering has shown that defects cost 10 to 100 times more if they are not caught during the requirements phase [46]. The advantages of successful security incorporation are clear, software will be designed from the ground-up with security in mind, and will reduce the cost of the system by finding faults during upfront analysis rather than in the testing and deployment stages.

Despite the obvious advantages to incorporating security in the development lifecycle, a major problem prevents security from being included. No clear examples of what security requirements should look like has been made. The problem stems from the difficulty and uniqueness of eliciting proper security requirements that are testable and verifiable. Often, attempts at generating security requirements result in a requirement set in a negative tone, such as “The program will not contain buffer overflows.” A requirement set in this tone is more difficult to test because proper testing would require testing of everything that the program cannot do, not what the program should do.

Proposed solutions to this problem include processes for secure requirements elicitation that require a dedicated software security expert or team available to the project. A software security expert is required because of heavy reliance on threat modeling, which all current security requirements elicitation processes leverage [75]. Threat modeling is a process for determining available assets a system protects, and the possible entry points an adversary would try to attack to get to those assets [74]. The process is effective, but relies heavily on the ability to identity threats and entry points. A team with little security background cannot be expected to identify all possible entry points to gain access to the system assets, the knowledge is not present to complete a thorough audit. According to Firesmith, this is because requirements engineers are not properly trained to produce requirements specifically for security [33]. This is also evident in the most prevalent security focused development lifecycle, the Security Development lifecycle, which uses threat models, and requires a security member to be present in the process [37].

Security has become extremely important in modern software development, and needs to be included in development from the beginning during the requirements phase. Furthermore, a process for providing software security requirements to development teams regardless of their security ability is currently needed. This thesis proposes a solution that is not threat-based, and does not require software security experts like previous threat-based approaches. Instead, the solution is vulnerability based, which views the system from an internal perspective [13]. By providing security requirements, any system will be able to prevent vulnerable states within their program, if the development process is done correctly.

1.2 Difficulties When Generating Requirements for Security

Producing software security requirements is complicated, and some issues increase the difficulty of producing such requirements. If these issues are not overcome, the development lifecycle cannot effectively prevent vulnerabilities. Current issues with generating software security requirements are discussed below.

Security is constantly changing. Even with a complete set of requirements for today's vulnerabilities, an expectation should be in place that vulnerabilities which differ from anything currently known will be found in the future. The process for generating requirements must be clear so that future security requirements can be generated should research provide insight into a new form of software threat.

Software security requirements should be stated in a positive tone. Security requirements are often stated in a negative tone, which makes verification and validation more difficult [49]. For example, a negative requirement might be "The program shall not allow remote exploits", which is difficult to validate. The process of verifying and validating that requirement would require testing of everything the program could not do, not what the program should do.

Software security requirements must be language and platform independent. Any approach designed to provide general requirements becomes less practical when linked

to a particular language or platform. Fortunately, the requirements can be written with a general tone to cover all possible scenarios that a project might face. Some requirements will be less visible in certain languages, for instance any language that abstracts memory management will not have requirements for failed memory allocation attempts, etc.

Software security requirements must be testable and verifiable for the development process to work. The principle idea behind requirements-driven software development involves generating requirements that can be verified at each phase of the development process, and that can be tested/validated during the testing phase of development [18]. Verification occurs at each phase after the requirements phase, and ensures that the criteria of each requirement is properly being met up to that point in the project. The testing of the requirements occurs at the end of the development process during the testing phase, and ensures inclusion of the requirement. A security requirement must be both testable and verifiable for it to be possible to track the progress of the requirement throughout the phases, and test to ensure the requirement was included into the project.

A project may only require some software security requirements, but not all. Software security requirements will cover everything from memory to cryptography, but some applications may only need a subset of those requirements. A process should exist so that the software security requirements needed for a project can be selected, based on the non-security related requirements.

1.3 Solution Approach

This thesis presents a set of software security requirements, and a process for selecting which of these requirements should be included in a particular project. The solution leverages previous work to identify current vulnerabilities that could produce exploitable software. This is different from current solutions which are threat based. The proposed solution provides four components:

(1) **Security Software Requirements:** The Software Taxonomy of Vulnerabilities [13] categorizes constraints and assumptions, that if violated, present a vulnerability that could potentially be exploited. By taking these constraints and assumptions as a starting platform, *software security requirements are generated so that if properly incorporated into the development process, they will prevent the constraints and assumptions from being violated.* By preventing the constraint/assumption from being violated, a vulnerability cannot exist for that constraint/assumption. These software security requirements are operating system and programming language independent, making them a viable option for the development of any software.

(2) **Software Security Requirements Tree:** The Software Security Requirements Tree (SSRT) is a structure designed to aid in the selection of particular software security requirements a project will need. *Based on the current requirements of project, the SSRT can be used to determine any additional requirements that may*

be required in that project to provide the most secure software possible. The tree structure is designed to be used by developers that may not have a robust security background by leveraging knowledge of their project.

(3) **Artifacts in the design & implementation phases:** A software requirement that has been properly verified during each phase will produce artifacts that can be seen during the design and implementation phases. Based on a requirement, general artifact expectations can be determined. Each of the software security requirements provided by this thesis have artifacts expected in the design and implementation phases, which should be verified when using any of the requirements. *The artifacts provided by this research are operating system and programming language independent, making them ideal for verification in any project.*

(4) **Full VV&T Process:** Access-driven VV&T is an approach to software development guided by accessibility to the development process, and the availability of development artifacts [5]. Full VV&T is a type within VV&T that describes when security has been included in development since the beginning during requirements generation. *The research provides a general process for how the software security requirements, and the artifacts can be used in Full VV&T.*

1.4 Organization of the Thesis

The organization of the remainder of this thesis is as follows: Chapter 2 discusses current processes that handle security in the software development lifecycle. Chapter 3 introduces the process for generating software security requirements, the list of requirements generated with this process, and the feedback from subject matter experts on the requirements. Chapter 4 presents support material for the software security requirements in full access-driven VV&T. The support material consists of security artifacts present in the design and implementation when each software security requirement is used correctly. Chapter 5 presents the SSRT, a tree structure that helps select only the necessary software security requirements. A prototype tool is also supplied, which provides a graphical interface for the SSRT. Chapter 6 contains the conclusion, and possible future work.

Chapter 2

Current State of Secure Software Development

The software development lifecycle referenced throughout this thesis is often referred to as a classic lifecycle or waterfall method, and is a sequential approach involving several phases. Each phase in the classic lifecycle builds on materials from prior phases. Although this approach could contain feedback loops, where a phase can modify materials from a previous phase, most organizations treat the development lifecycle as a purely linear approach [65]. Development lifecycles using this approach undergo four phases before deployment: requirements engineering, design, coding, and testing. During the first phase, requirements engineering, a document is created, which contains all the requirements for the project including any function, behavior, performance, and interface for the project. Next, the design phase begins, which use the requirements of the project to map data structures, software architecture, interface details, and algorithms. This information is documented, and with the requirements is used during the coding phase to translate the project into machine language so that it can be executed on a computer. The testing phase, ensures the correct system is developed, based on the requirements agreed upon during analysis. The classic software lifecycle provides a systematic approach to developing correct and bug-free software, but originally did not contain any direct considerations for security.

Penetration testing was originally the only security analysis process conducted on software in development [47]. Penetration testing attempts to find vulnerabilities during the testing and deployment phases, although discovering all the vulnerabilities in the testing phase is more desirable. This process is not ideal, and often only yields a small number of vulnerabilities during the testing phase [4]. Penetration testing is therefore recommended to be used in conjunction with other security analysis techniques. This shift from the testing only phase to the full development lifecycle has not been immediate.

This section first covers threat modeling because of its importance across approaches to secure software development. The phases of the development and their relation to security are then covered backwards through the software development lifecycle because this more closely mirrors the evolution that secure software development took.

2.1 Threat Modeling

Threat modeling is a key process used throughout different approaches to handling secure software development, and is seen among approaches that occur in the requirements, and testing phases. Because threat modeling is used by many formal approaches to security in software development, it is discussed before the breakdown of security approaches by development phase. Threat modeling is an attempt to capture the thought process of an adversary that wants to achieve a set of goals on a system [74].

Threat modeling takes the viewpoint of a potential adversary interacting with the system. Any methods of interaction with the system are potential entry points. Entry points are required so access can be obtained to desirable assets the system possesses. Using this knowledge, the threat modeling process starts by identifying the assets of a system, and potential threats to those assets. A threat does not exist unless an entry point leads to access of an asset. Attacks to achieve that threat can be enumerated and illustrated using a couple different diagrams. For example, a system may store passwords which are an asset to an adversary, and the threat is that an adversary steals those passwords. A diagram would then be made to represent the different attacks that could achieve the threat of stolen passwords. Once the diagram is complete, it highlights the areas that need mitigation to prevent the overarching threat from being realized. Any attack that is not mitigated, or is mitigated improperly, has a vulnerability that could be exploited to gain access to the asset that the system protects.

The most common diagram for enumerating threats are attack trees. In some literature they are referred to as threat trees, but this terminology is slightly incorrect, because the diagram used is actually an attack tree. Currently no literature uses an actual threat tree for modeling a threat, but the diagram is discussed here so that the basis and groundwork for attack trees can be seen. Misuse cases can also be classified as a threat modeling technique, because although the idea stemmed from use cases, the end result is still a diagram of threats to a system. Misuse cases have not yet been used in testing approaches where attack trees are used nearly exclusively, instead they are used in the requirements generation phase of development much like use cases. Threat trees, attack trees, and misuse cases are discussed in the subsections below, and can be used to enumerate and diagram a threat of a system.

A major fault with threat modeling is that it requires someone with security knowledge to be effective. The fault stems from having to analyze the system for possible entry points and attacks to gain access to a desired asset. It should be expected that only someone with a security background properly understand or possess the knowledge of how to gain access to an asset. Evidence of this can be seen in the secure development lifecycle which utilizes threat modeling to drive its requirements, and has a heavy reliance on security experts [27]. Any process which leverages threat modeling therefore is at a disadvantage because a security expert will be required. The proposed work avoids this disadvantage by providing security requirements that can be used in development without security experts.

2.1.1 Threat Trees

The threat tree diagram has more classification elements than attack trees. The root node is abstract and can be a specific threat, or a system that contains threats. The children of the root node depend on the type of root node, and are either an instance of the root threat, or first level threats to a system. The process can continue iteratively, with the each child of a threat being an instance of its parent. This allows for a classification system, where a parent is more of a generic threat, and its children are a specific instance. Any threat node within a tree, when adversarial based, can have attack possibilities as child nodes which are ways to achieve the threat of the parent. Those attack nodes can then have child nodes that enumerate the possible ways the attack can be achieved, and the process can also be done iteratively, to create several levels of an attack enumeration. The result is a tree that can define all the threats a system faces, and provide a clear classification structure.

The original design of threat trees by Weiss also allowed for values to be associated with a node, which starts in the leaf nodes of the structure [2]. Multiple values can be associated with a node, for example, effort and cost can simultaneously be described in threat trees. The values can then propagate up the tree, making values of a parent node a factor of their children. Areas that need a higher level of focus can then be identified based on the values of that section of the tree as compared to other sections. Mitigation of the threats and attacks can begin, and the values associated with the tree can highlight the areas that need focus.

2.1.2 Attack Trees

In 1999, Schneier introduced attack trees to provide an approach to describe the security of a system, based on potential attacks a system faced [69]. Attack trees are similar to threat trees, except they enumerate the possible attacks of a single threat, and only enumerate threats from an adversary. A system will commonly have multiple attack trees, one for each of threats the system faces. The differences in attack trees make them more suitable for threat modeling, and is the primary reason their use is preferred in the process.

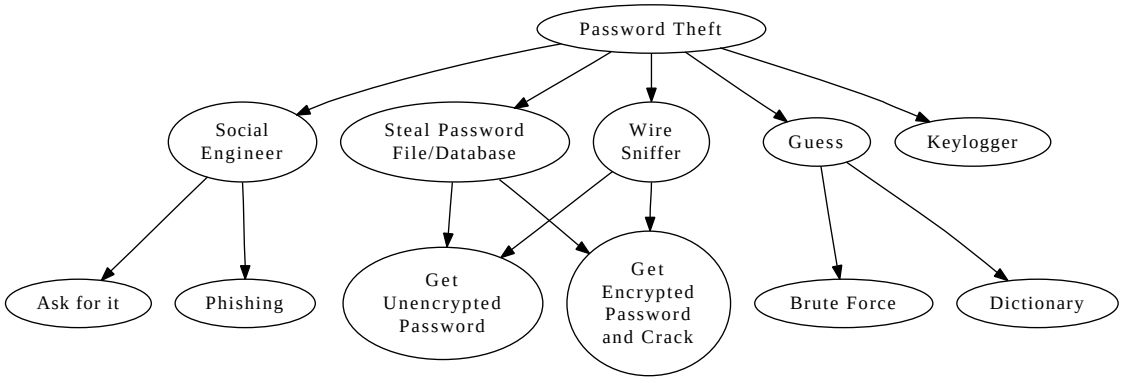


Figure 2.1: Example Attack Tree for Password Theft on a Website

Like threat trees, attack trees take the form of a tree structure, where the root node of the tree is a goal the attacker wishes to achieve, referred to as the threat thus far. Each node on the tree, other than the root node, is a way to achieve the goal of its parent, as shown in figure 2.1. The process can be done iteratively until an end leaf is reached and represents a single attack method to accomplish the goal of its parent. As with threat trees, the leaf nodes can then have a cost associated with the difficulty in achieving that attack, which can then propagate up the tree. The cost feature gives indication as to which attacks are easier or cheaper to achieve, and as such should be of higher focus for proper mitigation.

2.1.3 Misuse Cases

A better understanding of use case diagramming is required before misuse cases can clearly be explained. Use case diagrams have two main components, actors, and use cases [16]. Actors are a representation of things that interact with the system, which can include people or other systems. A use case is a single action that the system can perform for an actor, and includes text that describes what the use case does for the actor. Using these components a story is told from the perspective of an actor who uses the system. This behavior is diagrammed through arrows that start with an actor, and flow through one or more use cases, and possibly other actors. This provides a sequence for the story, which leads to how the system should interact with its users, and ultimately a set of requirements to provide the functionality described in the use case diagram.

Misuse diagrams borrow heavily from use cases but provide a separate classification because use cases tend to focus on how the system should act. This makes use cases suited for providing functional requirements, but not non-functional requirements, the current classification of most security requirements [72]. Presently, misuse case diagrams are only seen in approaches that utilize threat modeling during the requirements phase, and have not been used to drive any test phase approaches.

The key difference in misuse cases is that the diagram describes a story that the system owner does not want to occur [72]. Misuse actors are the only component added, and represent an adversary. The same basic principles are taken from use cases, except a sequence of actions can be modeled to represent an attack on the system. The attack sequence starts with the threat, which describes the threat the adversary wants to achieve. Multiple attack sequences can are possible from the same threat, and are akin to an attack tree enumerating the different attack possibilities of a threat. For example, if the threat is having credit card numbers stolen, there are multiple attacks to achieve that threat, which are modeled in the misuse diagram from the adversary to the attack used, as shown in figure 2.2.

Misuse cases still have all the properties one would expect for threat modeling. The viewpoint is from the adversary, any threats are enumerated into the possible attacks, and mitigation can be focused on preventing the attack path which prevents the threat from turning into a vulnerability. The difference that distinguishes misuse cases from attack trees is that a single diagram can be used to diagram all the threats a system faces, rather than

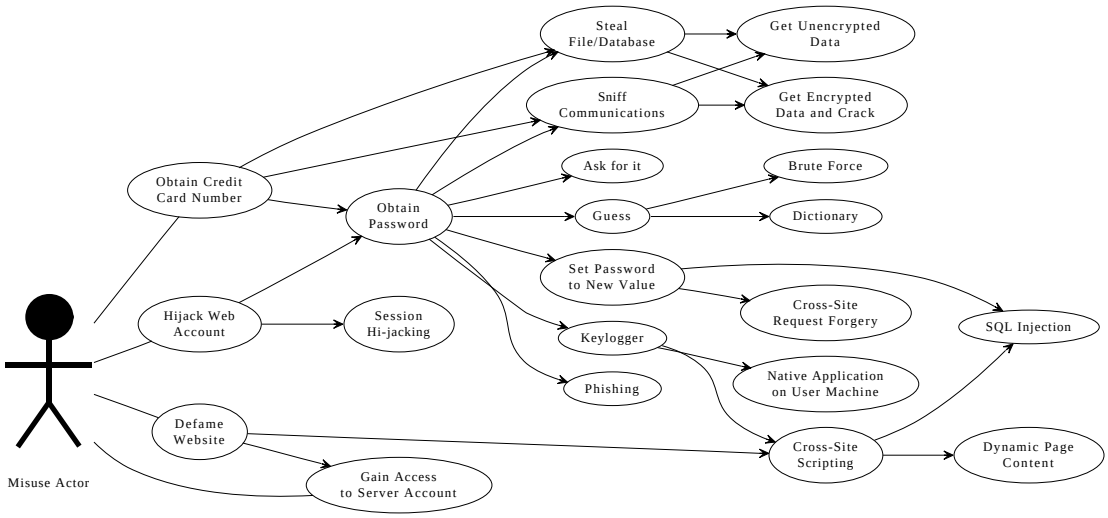


Figure 2.2: Example Misuse Case for a Website

requiring a different diagram for each threat.

2.2 Testing Phase

The testing phase occurs immediately before deployment, and is the last process in ensuring a correct product before release and maintenance begins. Initially the entire security process for software development occurred during this stage, but recently the importance of targeting security earlier in development has been recognized, particularly because of how late testing occurs in development. Security testing is unlike traditional software testing because of the uniqueness that security bugs pose to the operation of software. An adversary is actually looking for faults within the software, making it all the more likely that a vulnerability will surface. This requires security testing to push the software to its limits, because an adversary will do the same to find vulnerabilities. Testing for security can be broken into two main types, functional and risk-based testing, both discussed in 2.2.1. Tests of these types are used in the five stages within testing, which are discussed in 2.2.2 in the order they occur.

Because testing occurs in the last phase of development, security testing should augment other practices that occur earlier in the development cycle. Finding bugs later in development can be more costly, and even require a redesign should a major flaw be discovered. Most importantly, testing will only occur for a finite period of time before release, but an adversary will have unlimited time. So a system that passes rigorous security testing is not proved to be secure, but only proven to have no security issues with the tests that were executed. Modern software demands that security issues be discussed during the requirements, design, and implementation stages, so that no surprises arise during testing or maintenance.

2.2.1 Types of Security Testing

Security testing can be broken into two main types, functional and risk-based testing. Functional testing focuses on functional requirements, while risk-based testing focuses on negative requirements. There is some overlap between the two types, where testing exploration is encouraged in both types, but in general the focus of functional testing is to determine if the system behaves as expected [51].

2.2.1.1 Functional Testing

Functional testing certifies that the software behaves as expected [51]. The most critical technique for determining compliance is by testing functional security requirements. The other techniques for functional testing all have the ultimate goal of enforcing proper behavior of the software. This includes testing boundary conditions, load/performance testing, and testing typical fault conditions, which are all designed to attack typical vulnerable areas. Some of the techniques are exploratory in nature, meaning the effectiveness of the testing is highly dependent on the knowledge and ability of the tester. This is in contrast to testing a functional requirement, which is directly testable, making it the most attractive test technique. Functional testing occurs in all of the stages of testing, as discussed in 2.2.2.

2.2.1.2 Risk-Based Testing

Risk-based testing attempts to test areas previously identified during risk analysis [51]. The techniques include testing negative requirements, testing the system's interaction with components and environment, testing assumptions that developers often make, and building a fault model to determine what might go wrong with the system.

Because all risk-based testing involves critically analyzing the system and risks associated with the system, risk-based testing is highly dependent on the skill and security knowledge of the tester [51]. The approach taken in this thesis involves providing requirements that are testable and verifiable, and therefore functional requirements, alleviating the need for a knowledgeable security tester for those requirements. This is not say that testing with knowledgeable security experts should no longer occur; security experts are a critical part of evaluating the security of a system, but it is the assertion of this thesis that functional requirements testing remains more effective because the requirement is directly testable.

2.2.2 Stages within Testing, and their Relationship to Security

The activities required for functional and risk-based testing begin almost immediately when the development process starts. This is especially true for security testing, where planning must immediately consider the security risks of the system based on its desired features, desired executing environment, and the impacts of a security lapse.

2.2.2.1 Unit Testing

Unit testing is the first stage of testing for a system. A unit of code can be a function, method, class, etc. Because testing at this stage does not include the entire project, the type of testing is purely functional based [51], any risk-based approaches make more sense with an integrated system. Security threats can originate deep within a system [51], meaning a threat can be the result of a mistake within a single unit. For example, a race condition when reading a file or a lack of bounds checking for a buffer copy can all occur within a single unit.

2.2.2.2 Library and Executable File Testing

Library and executable file testing follows immediately after unit testing has ended. A project often makes use of libraries and other executables produced by a third party to cut down on development costs. This can include libraries provided by the underlying language. Library and executable file testing focuses on determining any faults or incorrect assumptions made about the underlying libraries and executables, which can lead to vulnerabilities. If a library or executable contains a vulnerability, any software system that leverages this library can potentially be vulnerable as well. An attacker will often be more familiar with a library than custom software, so a vulnerability in a library is an easier target, making testing here critical. Testing in this stage should also incorporate stress testing as described in 2.2.2.4 to determine the ability of leveraged libraries and executables to handle limited or blocked resource situations.

2.2.2.3 Integration Testing

When the subsystem components of a software system have reached a point where they can be integrated, testing must ensure that the subsystem components interact as planned. Incorrect assumptions made between those subsystem components can lead to vulnerabilities. Error handling routines are of particular interest since assumptions are often made that an interfacing subsystem component is doing the error-checking, or handling a detected error case.

2.2.2.4 System Testing

This stage tests, for the first time, the entire system at once. System testing is critical because it is generally the last process before the system is released. When testing security at this stage, the two techniques often employed are penetration testing and stress testing. Both focus on uncovering factors that could lead to a vulnerability with the system, and because the entire system is present, it is possible to uncover any real exploits.

Penetration testing attempts to compromise the security of the system [77], and is gen-

erally done at the system testing stage because it makes most sense; this is the first time the full-system, and real vulnerabilities will be present [51]. Modern penetration testing can be broken into two techniques, black box testing, and white box testing. Black box testing takes the standpoint of the adversary, where testing is done without knowledge of the system, and as if the internal workings were completely unknown. On the other hand, a tester in white box testing is given information about the system [77]. White box testing is preferred because the utility of black box testing is low since it requires the tester reverse engineer the system to determine its inner workings [77].

Penetration testing started as ad-hoc methods, but eventually moved to automated tools for repeatability, and quicker analysis. Off the shelf automated penetration testing tools, such as Acunetix [1] and WebScarab [64] which test web applications, are considered a black box testing technique because they test general problems within that domain, and are not written to test specific systems. The preferred method, white box testing, should start with off the shelf tools used as launching points, and from there the test team should create custom or tailored tools that target the specific system being tested.

Stress testing attempts to find problems when software is put under load or stress. Software executes on a system that provides resources, and when access to these resources is limited or blocked unexpected behavior can occur. Particular issues arise when developers incorrectly assume unlimited availability of those resources. To help facilitate this type of testing, tools such as Holodeck [38] exist that can simulate a system under stress by limiting or blocking resources so that software can be tested in such an environment.

2.2.2.5 Operational Testing

Once operational testing has been reached, the software system has been released to users. The system could still be in beta stages, where it is not yet considered an official release, or it could be the final delivered product. Finding vulnerabilities at this stage is still critical, especially considering an adversary could potentially find a vulnerability first, exploiting any users that use the software. Auditing should continue for the lifetime of the software, because new types of vulnerabilities not considered during development may be discovered.

2.3 Implementation Phase

The techniques for this stage focus on educating programmers on proper secure coding techniques, and source code analysis. Both techniques, when done properly, can be effective at eliminating vulnerabilities within code. Often times they are language focused because of the subtleties present in different programming languages, which eliminates their broad

appeal. For example, automated code analysis tools exist, but are written to analyze a specific language. Such a tool can often be very valuable, but unfortunately is only valuable when the project is using that programming language.

Despite the effectiveness of implementation phase techniques, issues are present and demand that security considerations start at an earlier phase of development. A flaw introduced in the design phase will be present in the implementation, and can only be fixed through a redesign. In the best case the flaw is discovered during the design phase, but may go unnoticed until after deployment. This does not detract from the importance of the implementation phase techniques discussed here, but the demands of modern software dictate the need for a process that begins before the implementation phase.

2.3.1 Secure Coding Practices

Secure coding will not prevent flaws introduced by a poor design, but proper education can eliminate bugs introduced during implementation. The major drawback with these practices is that all developers on the team must be aware of the secure coding practices, and must be aware of broader issues present in lower level languages. A mistake by a single programmer, involving a buffer overflow or integer overflow, can lead to a major vulnerability within the system, and may not be detected until after release. The solution is requirements that targets implementation phase issues, ensuring that each developer does not make a mistake during implementation. This follows the suggestion of researchers that security concerns start during the earliest phase of development possible [77].

2.3.2 Source Code Analysis

The purpose of source code analysis is to reduce implementation mistakes. Whether the technique is automated or human conducted, there are benefits to having experience with the programming language, and experience with the underlying system. Source code analysis techniques attempts to target areas that routinely have issues, and provide assurance that no vulnerabilities are present. Any vulnerabilities found will be bugs present at the implementation level, flaws introduced by a poor design must be met with work at an earlier phase of development.

Code Review is the most natural form of code analysis, and is conducted by a peer. The reviewer checks for compliance with requirements, coding standards, commenting, and other project documentation to determine if the correct guidelines were followed. The reviewer has two possible approaches to finding vulnerabilities, top down, and bottom up [44]. In the top down approach, the reviewer looks for specific vulnerabilities without needing knowledge of the inner-workings between components. The approach is quicker than the alternative, but misses some more complex issues in the code. The bottom-up approach makes it possible to find more complex issues in

the code, at the cost of requiring extra time to gain a better knowledge of how the code components interact.

Analysis Tools are designed to speed up the manual code review process. Tools can be very effective at finding specific issues, but struggle finding issues deep within the code, much like the top down approach mentioned above. Analysis tools can only be used for their targeted computer language, and are only as effective as the rules provided to them [44]. Because they are based on rules, the possibility of false positives and false negatives exist, and so they should not be relied upon solely for code analysis. Regardless, analysis tools can be very effective, especially in the hands of a skilled code reviewer augmenting the manual process. Some examples of analysis tools include ITS4 [26] and Flawfinder [81] which both attempt to detect common mistakes and vulnerabilities in C/C++ code, and BOON which attempts to find buffer overflows in C code [80].

2.4 Design Phase

The design phase is arguably more important than the implementation phase in terms of security, because an incorrect design will produce an incorrect implementation. Meaning, any security issues introduced by the design cannot be corrected by a simple source code fix and patch. The design has to be redone, and the effects on the implementation are dependent on the amount of required changes in the design. In the worst case, the design specification is an industry standard, requiring that a whole new specification be created, leaving implementations of the original specification insecure. For example, Wired Equivalent Privacy (WEP) is an encryption specification for wireless networks, but contained several flaws [19]. Because of the flaws, no WEP implementation could be secure, and new standards had to be introduced to fix the flawed WEP specification. The importance of design cannot be overlooked in any project, especially in larger projects which depend on clear, concise, and secure specifications.

Despite the importance of software design, few approaches focus on the design phase. Considerations for the design phase have mostly been left out in favor of requirements phase approaches, where a secure design can be achieved through writing a complete set of requirements. However, there are two available approaches for security in the design phase: security patterns and UMLsec. Security patterns are solutions to a particular context [83], and therefore not a suitable general approach to the requirements phase. UMLsec is a modeling approach based on the Unified Modeling Language (UML), designed to provide security-engineering to the design phase.

2.4.1 UMLsec

Jan Jürjens introduced UMLsec, as an approach for security engineering modeling using the Unified Modeling Language (UML) [41]. UMLsec adds extensions to the UML modeling

language, providing extra tools for modeling the security constraints of a system. The UMLsec process is designed to work with requirements of a project, meaning it is not a solely design based approach. The security requirements of a system are checked against the UMLsec model to ensure the model adequately fulfills the requirements. The model can then be used to generate test cases, ensuring the design was correctly implemented. UML was chosen as the base because of its wide use in the industry and the classroom, intuitive techniques, precise definitions, and the high number of available tools.

Modeling languages describe a system with pseudo code, actual code, diagrams, pictures, or descriptions [53]. UML uses diagrams to model a system, which is preferred because it provides unambiguous and explicit details while still being abstract enough to quickly explain how the system works. Precise meaning is accomplished in UML by defining a formal language, which has well defined notations for abstractions. Although UML can be used for modeling systems other than software, it its original purpose, and most widespread use is for software design. Figure 2.3 shows an example of a UML diagram for a simple banking system, with limited features. The diagram shows relationships between sections of the code, and the services provided by those sections. Using this diagram, the system can be broken into parts, and taken to the implementation phase.

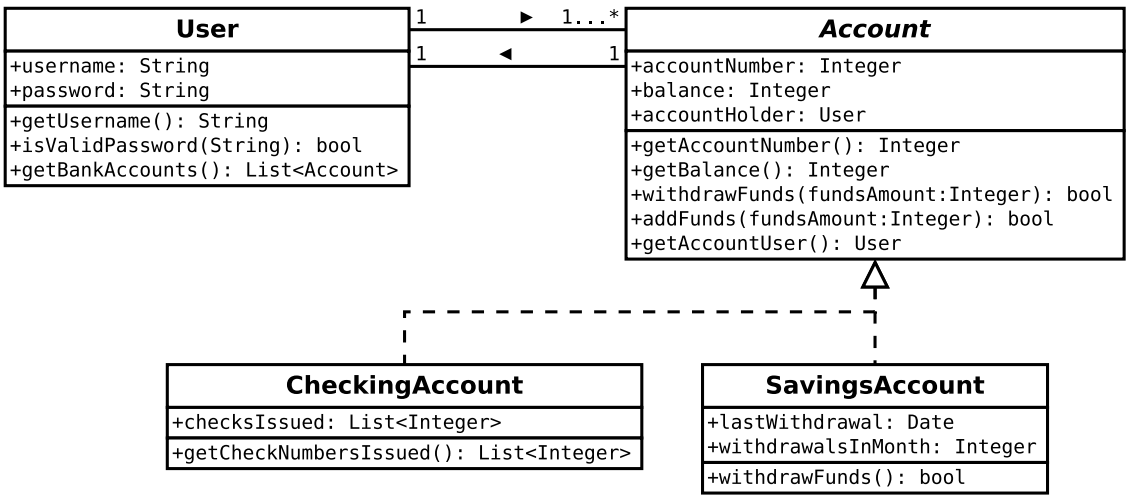


Figure 2.3: Example UML Diagram for a Simple Banking System

UMLsec is an extension that provides common information security requirements as specification elements to UML, such as “secrecy”, “authenticity”, and “adversary”. Information security requirements focus on what the program should do, not how the program should do it as with software security requirements. The differences between the two types of requirements are discussed further in 2.5.1. The security elements of UMLsec are used to add security features to the model of interactions between system components. For example, if an application were to transmit credit card information from client to server, the secrecy and authenticity elements would be used to designated that exchanges between the two components should be both secure, and authenticated. Using the provided validation rules,

UMLsec diagram can verify whether a design enforces given security policies, and indicate potential vulnerabilities.

The limiting factor to the approach is due to the focus on information security requirements, and not on software security requirements. For example, an information security requirement might require that data be transmitted securely, which can be modeled in UMLsec. The details on how this information will be kept secure is not covered, and is the domain of software security requirements.

2.4.2 Security Patterns

Security patterns provide a proven solution to specific, but similar security problems [70]. In other words, a security pattern is a method that uses past successful designs for the basis of the design of a new system. Current literature describes security patterns for the majority of security problems faced by modern software systems. These patterns are not absolute recommendations, and instead only describe the current practice of other systems. Considerations for the most secure technique are not always discussed either, as shown in the example below.

Security patterns are primarily a design phase technique because a generic solution is given for the problem, although security patterns could be looked at for implementation level concerns. The generic designs provided by security patterns produce an information security approach, not software security. Designs for an actual system will require more in-depth considerations to get correct. For example, the security pattern for "password design and use" provided by Schumacher et al [70] lists the techniques that several systems use, but does not give absolute recommendations for design. In this same pattern it is mentioned that *some* systems encrypt passwords, and do so either reversibly or irreversibly [70]. Not only does this miss the point that passwords must be encrypted, but the security pattern introduces another issue, when to use a reversible technique, and when to an irreversible one. Furthermore, the security pattern does not cover how to correctly mitigate the threat of rainbow table use on a irreversible hashed password because the issue is at the software security level, not information security level. Security patterns are too generic to provide real solutions to security problems, and ultimately require the need for more considerations in software security.

2.5 Requirements Phase

Security experts have recommended that security activities in software development start at the earliest stages of development, during the requirements phase [77]. The recommendation was given because proper requirements can lead to a secure design, and provide insight into proper implementation preventing security bugs. Unfortunately, requirements engineers struggle to properly elicit security requirements [33]. The need for processes to generate security requirements began to emerge, and solutions have been proposed, all of which leverage

threat modeling in some way [75]. Threat modeling, as discussed in 2.1, identifies assets the system protects that an adversary would want, then determines entry points an adversary would attack to get those assets, followed by identifying all threats to that entry point. By mitigating all the threats associated with an entry point, an adversary will be unable to access the associated asset. When generating requirements for security, the mitigations created in a threat model are transferred to a requirement for the project to prevent the threat from occurring. This basic approach is followed by all security requirements generation processes, except some contain more features such as prioritizing threats, and others have a heavier focus on identifying customers needs, etc.

The issue with all such approaches is that it requires a security expert to be present during the process because the threats and mitigations can only be identified by someone familiar with methods of exploiting an entry point. This currently forces all projects to have security experts present during requirements generation, and when any changes are made to the requirements. For example, if during the design process the project team decides that some network communication will have to be added, the threat model, and therefore the security requirements given by the security expert are no longer valid. This thesis presents software security requirements in 3.3 that are designed to mitigate the threats associated with modern software. Then, the requirements that are needed by a project can be selected using the SSRT described in 5.1, and even updated when non-security requirements are modified.

2.5.1 Information Security vs. Software Security

Prior literature has not made the distinction, but there are two main types of security requirements in existence. The first is information security requirements, which are high-level and focused on the function of the software. The second is software security requirements which focus on the design, and specifically the operation of the software. The difference is that information security requirements dictate what the system should do, while the software security requirements dictate how the system should be implemented. For instance the requirement, "The process shall authenticate all users before the system can be used" is an information security requirement. The how of authenticating is just as important, and more susceptible to errors because a simple high-level information security requirement will often lead to several software security requirements. In the given example, how the users will authenticate needs to be decided, how the process will verify correct authentication needs to be determined, which in turn will dictate the type of data being stored by the process and how that data needs to be stored. Even in the rare case that all software developers on a project have been trained in secure coding techniques, as described in 2.3.1, flaws will be introduced during the design phase without software security requirements. Security issues introduced during the design are an estimated minimum 50 percent of the security issues present in modern software [78], making them extremely important to the security of a system.

The approaches designed to provide security requirements are all capable of producing

either type of requirement. However, some approaches focus more on one type than the other, which is evident when reading the examples of requirements provided in their literature. This is most likely because people involved on the subject come from two fields, software security, and information security [78]. Proposed solutions tend to focus on the requirements for the field their authors work in.

This thesis focuses on software security requirements and provides requirements for that field, which are more difficult to define, and more often neglected in projects.

2.5.2 Current Requirements Phase Approach

Rather than discussing each approach in detail, the lightweight approach proposed by Tondel [75] will be the focus of requirements phase processes. Because all of the current approaches use threat modeling, their processes are all nearly identical, and Tondel simply produced an approach by piecing together the previous work. The lightweight approach is preferred by this thesis because of the less abstract approach, the simple clear process, and the benefit of incorporating ideas from all previous work. The approaches Tondel's work is based on are discussed in less detail for comparison purposes.

2.5.2.1 Lightweight Approach

Tondel proposed a lightweight approach to security requirements generation that builds on the experiences of the prior approaches [75]. The idea is to create a process that is more clear, less abstract, and still capable of yielding security requirements needed by a project. The following steps are listed in the approach:

(1) **Determine security objectives.** Proper security cannot be implemented without knowing the desires and needs of the customer using the product. The first goal of this step is to identify the customer's needs. The second is to identify legislation, policies, standards, and best practices required for the system. The information gathered here will affect the later steps.

(2) **Identify assets.** An adversary will not attack a system unless there is an asset available, so identifying assets is key to determining how the system needs to be protected. Tondel recommends that asset identification be done through different viewpoints: the customer's, the system owner's, and the attacker's. After identifying the assets, further analysis should be done on each to identify a priority on protection. This will ensure that the most valuable assets receive the most attention in threat mitigation.

(3) **Perform threat modeling.** With the assets identified and prioritized, threat modeling should be performed to identify the ways the asset could be compromised. Attack trees are recommended for this step, because of their flexibility to work with

any project. Misuse cases can be used with this step, because they are also a form of threat modeling, but are less recommended unless the project team is more comfortable using them. The threat model should consider all possible threats, and should not rule out any since the design of the system has not yet been done.

(4) **Document security requirements.** The last step requires taking the threats previously identified, determining how to mitigate those threats, and then writing a requirement that enforces that mitigation. These requirements should then be traceable back to the assets they are designed to protect, so that the same priority among requirements can be determined. Lastly, Tondel recommends putting all the security requirements in the same document as the general requirements.

2.5.2.2 Eliciting Security Requirements with Misuse Cases

The first approach designed to elicit security requirements was presented by Sindre and Opdahl in 2000 [72]. The paper simultaneously presented a new approach to threat modeling called misuse cases, discussed in 2.1.3. The focus of their work is on describing a process for threat modeling with misuse cases, and not eliciting security requirements. Subsequent work has leveraged misuse cases, mentioning that misuse cases should be used, and then expanding on the process to provide more steps to the elicitation process. Sindre and Opdahl's approach is more direct, suggesting that requirements be elicited to prevent the threats identified in the misuse cases, and nothing more.

2.5.2.3 Microsoft's Requirements Approach

Microsoft's approach to security requirements generation is part of an entire lifecycle process, described in 2.6.1. In the security development lifecycle, requirements generation focuses on a tight partnership with a security team. This security team evaluates the project's goals and objectives, the required timeline, and how the project is to be integrated with other software. The security team then assists the product team in developing documentation that considers those factors.

A major drawback of Microsoft approach is the lack of a formally defined process for creating security requirements. While the security team is expected to provide help in producing requirements, no process for the security team is provided [37]. The importance of threat modeling, and the importance of starting security as early as possible is discussed, but no indication of how the results from threat modeling are to be used for the requirements phase is provided.

2.5.2.4 Van Wyk and McGraw's Approach

The approach presented by Van Wyk and McGraw not only focuses on requirements, but also on each step of the lifecycle [78]. For the requirements phase, abuse cases of

the system are created to identify the areas under the most risk. The approach suggests that the development team create the abuse cases with information security experts. This recommendation is given because developers often lack the necessary knowledge to identify many of the attack forms [78].

2.5.2.5 Haley *et al's* Approach

The requirements generation approach presented by Haley *et al* defines a process that starts with a project's business goals and functional requirements, and concludes with necessary security requirements [35]. The approach is similar to the lightweight approach, except for the business considerations, and formal process for requirements verification. The four step process provided is:

(1) Identify functional requirements

(2) Identify security goals

(3) Identify security requirements

(4) Construct satisfaction arguments

The first step, identifying functional requirements, is listed to provide indication that non-security requirements are to be generated before their process takes place. The process for identifying such functional requirements is left to other literature. With the functional requirements in place, the security goals can be identified, which is Haley *et al's* term for threat modeling. The process for threat modeling is the same as described in 2.1, where assets are identified, followed by the identification of threats to those assets. The difference in their security goals approach is that business policies set by the organization using the system are to be considered as well. When an identified threat has a security-based business policy designed to prevent that threat, the systems goal should be to enforce that business policy. The enforcement of such policies will lead to information security requirements, not software security requirements, as security based business policies are written to provide separation of duty, auditing, etc., which are all information security based.

After producing security requirements, the last step in the approach is verification. The verification step contains two parts, a formal outer argument, and an informal structured inner argument [35]. An outer argument shows that if a claim about the underlying domain or system is true, then the security requirement satisfies its intended purpose. An inner argument is a set of claims to support a claim made in an outer argument. The process is not perfect, but reveals security requirements that need to be reevaluated because of a basis on incorrect assumption(s).

2.5.2.6 SQUARE Methodology

The SQUARE methodology contains many of the same steps found in the lightweight approach, while adding a few of its own. Like the Haley *et al* approach, a review step is given at the end to evaluate the security requirements that are generated. The only drawback of this approach is the failure to properly make a connection between threat modeling and requirements elicitation, but instead opting to suggest requirements elicitation techniques designed for non-security use [48]. The approach is as follows:

(1) **Agree on definitions.** This step is unique to the SQUARE method, and suggested to provide clear communication throughout the process. All technical terms used within the field are to be defined, including such terms as attack, patch, etc.

(2) **Identify security goals.** This step is the same as the first step in the lightweight approach.

(3) **Develop artifacts to support security requirements definition.** This step, and the next both cover steps 2 & 3 in the lightweight approach.

(4) **Perform risk assessment.** This step, and the previous both cover steps 2 & 3 in the lightweight approach.

(5) **Select elicitation techniques.** The other approaches imply that the threat modeling should yield security requirements, without providing any clear process as how to derive them. The SQUARE method differs by suggesting a list of requirements elicitation techniques. Unfortunately the techniques are not security focused, and no process for using threat modeling during elicitation is provided.

(6) **Elicit security requirements.** In this approach the requirements elicitation is based on the technique chosen in the previous step. Additional suggestions are provided to ensure the security requirements are unambiguous and verifiable.

(7) **Categorize requirements.** Categorizing requirements is unique to the SQUARE method, and breaks down each requirement into 6 possible categories. They define when the requirement will be used: at the system level, software level, or architectural level. Within each of these categories a requirement can also be labeled as essential or non-essential, bringing the total to 6 possible categories.

(8) **Prioritize requirements.** In the lightweight approach prioritizing is applied at the the threat level, and then priority of the requirements is determined by its associated threat. This method suggests prioritizing threats directly, enabling finer control over each requirement.

(9) **Requirements inspection.** The inspection attempts to determine the viability of requirements for use in the project. The method is informal verification process that attempts to prove that each requirement successfully prevents a threat, as in

Haley et al's approach. The inspection includes verifying that each requirement is testable/verifiable, financial viable, technically feasible, and whether it is applicable to one of the security goals.

2.5.2.7 Apvrille and Pourzandi's Approach

The approach presented in Aprville and Pourzandi's article, *Secure Development by Example*, contains an entire lifecycle approach, omitting only the maintenance stage [3]. The requirements generation process begins with defining the security environment of the system, which defines the potential attack surface. Examples of environments are: desktop application, client-server application, etc. With the security environment identified, business-centric security objectives can be created. These objectives determine the expected level for security during the risk evaluation process. Separate from the first two processes, a threat model should be created of the system in development. The threat model then drives the creation of security policies, referred to as information security requirements by this thesis. The last process in this stage is risk evaluation of the threats identified during threat modeling. Aprville and Pourzandi provided their own method for risk evaluation, which prioritizes the threats, and determines typical attack profiles. Threats identified during modeling can then be eliminated by assuming risk on low priority, and/or non-typical attack profiles. The amount of risk taken should be dictated by the security objectives created earlier. Threats that have been eliminated no longer require information security requirements, and so they too can be eliminated from the project.

2.5.2.8 XP Practices and Requirements Engineering

EXtreme Programming (XP) is a development approach that differs from the classic lifecycle that this thesis has exclusively discussed so far. A security requirements engineering approach for XP is mentioned for completeness on the state of approaches designed for eliciting security requirements.

The XP development approach is lightweight, intended to address constraints on software development, work with a development team of any size, and be capable of adapting to vague or changing requirements [14]. The last feature, the ability to adapt to vague or changing requirements, is the feature that primarily differentiates XP the classic lifecycle. As stated in the introduction, the classic lifecycle is generally sequential. Even when feedback loops are present, where a phase can modify a previous phase, the classic lifecycle needs the requirements for the project to be defined up front. If a requirement has to be changed or added during the design or implementation, the design and implementation phases have to restart, to ensure the project meets any change in the requirements. The result is an approach that is not very flexible, and requires much up front work before development can begin. The intent of XP was to solve this problem, by being more flexible and adaptable during the development process. This thesis presents an approach that is designed to work with the classic lifecycle because it is more mature, and more widely used.

The approach presented by Boström *et al* for eliciting security requirements in XP development will not be discussed in length since the approach is not developed for use in a classic lifecycle. Their approach is focused on maintaining terminology, and practices of XP programming, which is not a focus of this thesis. However, their approach still involves identifying assets, producing threat scenarios, and then finding ways to mitigate those threats [20]. There are additional steps required for inclusion in the XP development approach, which are needed to fit into the XP development ideology. Overall Boström *et al* presents an approach for eliciting security requirements that adapts a process similar to the lightweight approach to XP.

2.6 The Entire Lifecycle Processes

Entire lifecycle processes are the area of newest research for secure software development, and as such has a small number of proposed approaches. The secure lifecycle approaches discussed in this section are mentioned in the requirements section, but this time the entire lifecycle will be reviewed instead of just the requirements elicitation. The same techniques are often proposed in more than approach because they are leveraging techniques created previously for a specific phase of development. For example, all three approaches listed in this section recommend the use of penetration testing, which is a technique that has been researched heavily before any of the entire lifecycle approaches were produced. These types of techniques are leveraged because of the effectiveness in their targeted part of the lifecycle.

2.6.1 The Trustworthy Computing Security Development Lifecycle

Lipner and Howard presented a secure development lifecycle approach designed for use by software that address malicious attack [37]. Their approach is designed to work with Microsoft's software development process, which is a classic lifecycle. Security focused activities and deliverables have also been created for use in the secure development lifecycle.

Requirements Phase focuses on a tight partnership with a security team. This security team will evaluate the project's goals and objectives, the required timeline, and how the project will have to be integrated with other software. The security team assists the product team in developing documentation that considers those factors. The requirements come from threat modeling, or from security features stemming from customer demands.

Design phase contains different considerations for evaluating and defining the system under development. Evaluations include the attack surface, and threat modeling. The attack surface includes the parts of the system most exposed to users, and should have higher focus and lower execution privilege to mitigate risks. Threat modeling is a technique discussed in 2.1 - the process used here is the same. With the evaluations underway, the information can be used to define the security architecture,

and the ship criteria. The security architecture should focus on defining a strongly typed language for use, minimizing the attack surface, and design layering that will avoid circular dependencies. Ship criteria defines security goals the system should reach before shipment.

Implementation phase in the Microsoft development lifecycle includes all testing phases except operational testing. Their primary recommendation is to use the results of threat modeling, so that areas with higher risk can be of more focus. Coding and testing standards are defined and applied in this stage to mitigate vulnerabilities that a single developer may introduce during coding. The remaining processes during this phase include code reviews, use of automated code scanning tools, and penetration testing.

Verification phase in the Microsoft development lifecycle has been referred to as the operational testing stage by this thesis, and is when the system enters beta testing. Microsoft calls the process during this phase the security push, which contains several activities which should have been done previously if the secure development lifecycle has been done properly. The authors reiterate that despite the name, the security practices should begin before the security push. The techniques used during the security push will be the second time they have been applied to new code, so the focus of the security push should be primarily on legacy code [37]. During the security push, code review, threat model updates, penetration testing, attack surface scrubbing, and documentation scrubbing activities take place. The scrubbing activities involve verification and review of material, to ensure the information is correct in those documents. The length of time required for the security push will depend on the project, with short projects taking a minimum of three weeks, and larger projects taking a minimum of six weeks.

Release phase is not completed until the security team agrees that the secure development lifecycle has been followed properly. Once the security team is satisfied with the use of the secure development lifecycle, the product can be released, and the maintenance can begin.

2.6.2 Apvrille and Pourzandi's Secure Software Development

Apvrille and Pourzandi's approach focuses on a practical method for secure software development, inserting concerns for security at each step in the lifecycle, with the exception of the maintenance phase [3]. This differs from the Van Wyk and McGraw approach which includes concerns for the maintenance/release stage. The approach is by far the most practical of the three, and included in their published article is an example program developed with their approach. The concerns and techniques provided for each of the development stages is not new however, buts is a culmination of previous work.

Requirements and analysis begins with defining the security environment of the system, which defines the potential attack surface. Examples of environments are, desktop application, client-server application, etc. With the security environment identified, business-centric security objectives can be created. These objectives will determine the expected level of security during the risk evaluation process. Separate from the first two processes, a threat model should be created for the system in development. The threat model then drives the creation of security policies, referred to as information security requirements by this thesis. The last process in this stage is risk evaluation of the threats identified during threat modeling. Aprville and Pourzandi provided their own method for risk evaluation, which prioritizes the threats, and determines typical attack profiles. Threats identified during modeling can then be eliminated by assuming risk on low priority, and/or non-typical attack profiles. The amount of risk taken should be dictated by the security objectives created earlier. Threats that have been eliminated no longer require information security requirements, and so they too can be eliminated from the project.

Security design is performed using UMLsec and security patterns, both already discussed in 2.4

Implementing Security is accomplished through a clear focus on secure programming techniques in this approach. The implementation language is the primary concern, and developers should consider choosing an implementation language based on security since some languages have a higher rate of bugs associated with their use [3]. Once the implementation language has been identified, their approach has several recommendations for preventing implementation level issues, some of which will may be irrelevant depending on the implementation language chosen. Recommendations are given in three areas: cryptography, preventing buffer overflows, and preventing format-string vulnerabilities. As with the rest of their approach, the recommendations are practical, and provides recommendations with real examples. For cryptography, the only recommendation is to never provide a new implementation of a cryptographic algorithm, instead relying on previous implementations that have already been well tested. The buffer overflow prevention recommendations are only applicable to C/C++, and consist of three techniques: (a) never using str* or gets() functions, (b) providing a chokepoint function to ensure that only a fixed number of bytes are read with a null-terminator at end, and (c) using fixed sized buffers. As with buffer overflows, the format string recommendations are only applicable to C/C++. The recommendations to prevent format string vulnerabilities include never relying on user input to specify format strings, and limiting format functions to only those that output. Code reviews are also mentioned, however their approach provides little information on the expected procedure for code review.

Testing security in this approach partly involves using techniques that are listed in the implementation phase of this thesis. The techniques from the implementation phase are code reviews, and the use of code analysis tools, both discussed in 2.3.2. The last

technique in this approach for testing security of a system is the use of fuzz testing, which is a form of penetration testing.

Operating and maintenance is not covered in this approach.

2.6.3 Van Wyk and McGraw's Approach

Van Wyk and McGraw's approach to lifecycle security focuses on closing the gap between software security and information security [78]. Their argument is that software security experts are very good at understanding the issues that modern software faces, but do not adequately understand the issues within the information security field. The argument continues that the inverse could be said about information security experts, despite having a closely related ultimate goal. The approach they present attempts to bring the concerns of both fields to produce a more secure system, by combining the strengths of each.

Their approach begins with studying current software development based approaches to security, each of which is focused on a particular phase within the lifecycle. Then, they add their own recommendations to the processes by incorporating concerns from the information security field. The result is an entire lifecycle approach, that aggregates the best secure development ideas from each of the phases, starting from the requirements, and continuing through deployment. The activities in their approach are:

Abuse cases of the system are to be created to identify the areas under the most risk. The development team should create the abuse cases with information security experts. This recommendation is given because developers often lack the necessary knowledge to identify many of the attack forms [78].

Business risk analysis is conducted exclusively by information security experts to determine how potential security issues caused by the system under development affects the business. Information security experts must collaborate with the business stakeholders to determine their expectations of security, their concerns for security, and need for security. The information security experts will then have to evaluate systems similar to the one being developed to analyze the costs associated with attacks, potential system downtime due to attack, and how successful attacks affected the image of the business. Information gathered during this process affects priority of potential threats that are determined later.

Architectural risk analysis involves all the components used by the system in development, including the execution platform, required libraries, computer languages, and the design of the actual system. Although not explicitly stated in their approach, it is implied that threat modeling is performed on these components and the system in development to determine the possible threats the system will face. Information security experts are then asked for feedback, primarily on existing components the system will be leveraging. Their feedback provides info to the software developers on

what leveraged components are of higher risk because of any current vulnerabilities, vulnerability reputation, or higher exposure to attack. Information security experts provide real world issues present with the underlying components that a new system must rely on. Ultimately, their feedback can be combined with the threat model data to prioritize threats, and identify weaknesses not previously identified by software security experts.

Test Planning should focus on real-world attack scenarios, rather than pretend attacks [78]. Again, information security experts will be of value, providing information on attacks they have seen in real world environments. The attack plan should focus on risk-based testing, based on their experiences.

Code review is conducted by the development team, with the little input from the information security experts. Code review is discussed in 2.3.2, and the procedure used by this secure development approach is the same.

Penetration testing has a higher reliance on information security experts, since this is their field. The approach is same as the techniques discussed in 2.2.2.4.

Deployment does not end the need for security in software development. The system is tuned based on user experiences, and the development team must keep up with any vulnerabilities found in the system, or components leveraged by the system. Information security experts provide insight into logging and monitoring practices, in case an incident response is required.

2.7 Evaluation Techniques

Instead of providing a direct process for secure software development, criteria for evaluating security have been created in an attempt to standardize security evaluations of software. Often the focus is more on evaluating the process, rather than a technical review of the implementation. When the process is under evaluation, the expectation is that through ensuring a correct process, vulnerabilities can be limited. When the actual implementation is under evaluation, the security claims the developer makes about the product are scrutinized and reviewed. The important distinction is that the product is not critically evaluated for vulnerabilities, but only evaluated against the claims made about its security. These techniques are discussed outside of the lifecycle because they are intended to evaluate the lifecycle, rather than directly influence any phase.

2.7.1 Criteria for Evaluating Security

The most prominent criteria for evaluating security of a product is the Common Criteria for Information Technology Security Evaluation (CC) [17], capable of evaluating the security of software, firmware, or hardware. CC was developed by North American and European governments in hopes to completely standardized product security evaluations, limiting the

need for vendors to comply with multiple criteria. There are three targeted users of CC: consumers, vendors, and independent evaluators. The goal is to provide consumers with a way to (a) supply vendors security expectations of a product, (b) provide vendors a way to give assurance to customers of security built into the product, and finally (c) provide evaluators a standard process for reviewing how successfully vendors were at incorporating security concerns in the product. Some of the common terms associated with CC:

Targets of evaluation (TOE) - product under evaluation. The product can be software, firmware, and/or hardware.

Security functional requirements (SFR) - derived from the security objectives of the TOE. SFRs must be more detailed than the security objectives, completely addressing the objective but without specifying an implementation.

Security assurance requirements (SAR) - description of how the TOE is to be evaluated.

Consumers of the CC include users of the TOE, government or corporations wishing to define a standard for security, or developers of the TOE. The consumers define protection profiles (PP) which provide details on how TOEs using the PP are implemented. A PP contains security objectives, which are then refined to SFRs or SARs. Each PP contains either SFRs or SARs, but not both. This provides developers more flexibility when selecting a PP to use in their TOE.

Developers are defined as the implementors of a TOE. The developers create a security target (ST) for their TOE, which is similar to a requirements specification document. A ST contains a set of SFRs and SARs that must be followed by the TOE, which mirrors the requirements to implementation process already in place when using a classic lifecycle. The SFRs and SARs in the document comes from either PP(s) defined by the consumers, or are newly devised requirements by the development team. Should the requirements from a PP be selected for inclusion into a ST, then all the requirements listed in the PP must be included without exception. This will guarantee that the concerns of the customer(s) that created the PP will be implemented.

Evaluators use the ST provided by the developer, to determine if the security objectives of the developer are met. The exact process used by the evaluators depends on the evaluator assurance level (EAL), where higher levels or assurance requires more evaluation. The evaluation does not critically review the security through penetration testing or other security analysis, but only validates the requirements in the ST. If a requirement is needed for correct implementation or design of the TOE, but it is not present in the ST, then the TOE can pass the evaluation even if the missing requirement caused a vulnerability.

Security evaluation criteria, and CC in particular, come very close to matching the designed function of the software security requirements and the SSRT. However, security evaluation criteria only validate the claims the developer made about the system, and do

not critically evaluate the actual security of a system as implemented. CC mitigates this issue by providing standard requirements through the use of PPs. Nonetheless, a PP must be made by the industry or customer(s) for maximum effectiveness. Furthermore, the focus of the requirements in the CC are on information security requirements, where the SSRT focuses on software security requirements. CC also has other weaknesses including high costs of evaluation, and the inability to evolve with emerging threats due to bureaucracy [67].

Similar approaches to CC include CESG Claims Tested Mark (CCT MARK) [25], Trusted Computer System Evaluation Criteria (TCSEC) [54], and Information Technology Security Evaluation Criteria (ITSEC) [34]. The common issue with all criteria for evaluating security is that they test the vendor's security claims on the product, instead of critically evaluating the actual security of the product.

2.7.2 Development Process Evaluations for Security

The security of a system is highly dependent on the process used for development; if a strict process is not followed there is higher chance of failure being introduced. While a good process does not guarantee better results, it reduces the risk factor. In terms of security, the process for development can influence the vulnerabilities present in the system. For example, even if a full set of security requirements are provided, if a process does not use those requirements correctly, there is a high chance that a vulnerability will be introduced during design or implementation.

The need for process evaluation has led to System Security Engineering-Capability Maturity Model (SSE-CMM) [76], the most widely used process evaluation for security. The goal of SSE-CMM is to provide a model process that will be used to evaluate real-world industry processes for system development. A process that has been granted a high capability level does not guarantee a vulnerability free system as a result, but instead guarantees that a better process for security is followed. The SSE-CMM can evaluate the classic lifecycle used by an organization, even if the software security requirements and the SSRT are used.

2.8 Summary

The current state of secure software development includes approaches that cover the entire lifecycle at the recommendation of security experts. These lifecycle security approaches leverage techniques that have been previously generated over the years for each of the phases of the classic lifecycle. While these phase specific techniques are effective at handling security during that phase of development, all of them are only effective when used by someone highly trained in security. The need for security training is required in part because of the heavy reliance on threat modeling used throughout the techniques. Despite this flaw, these techniques when used properly, should reduce the number of vulnerabilities in a system. The next chapter describes software security requirements that can be used to design and implement a system even when the development team has limited security experience.

Chapter 3

The Software Security Requirements

Once researchers suggested that security concerns be included from the beginning of the software development lifecycle, processes for generating security requirements began to emerge. All current processes for generating security requirements utilize a technique called threat modeling, discussed in 2.1. Unfortunately, threat modeling places a heavy burden on a security expert being available for the process; a development team without a security background will not be capable of using threat modeling to generate security requirements. The software security requirements mitigate this problem, by providing general security requirements that can be used during the development of any project.

3.1 Foundation

The taxonomy of software vulnerabilities [13] provides a starting platform for the software security requirements. The taxonomy provides a categorization of constraints/assumptions, that if violated by a software process, could ultimately lead to an exploit. The software security requirements are written to prevent the constraints/assumptions from being violated, in hopes of preventing vulnerabilities caused by their misuse. The idea behind the Taxonomy is discussed below.

3.1.1 Object Model of Computing

The theoretical model of computing provides a foundation for the taxonomy of software vulnerabilities by establishing a relationship between vulnerabilities, computer system resources, and the software process [13]. The relationship starts with the software process, which executes on a computer system, receives input, and provides corresponding output after performing operations based on the input. The computer system provides resources that the software process uses to perform the operations, but the resources have constraints that must be enforced by the software process and/or assumptions about the usage of the resource that a software process must consider. This places a restriction on the software security requirements, they are targeted at software processes running in an operating system environment. If a software process violates a constraint or assumption of a computer system

resource, undefined behavior will result. The undefined behavior means a vulnerability is present in the software process, which may be exploitable by an attacker. The differences between vulnerabilities and exploits are discussed in 1.1.

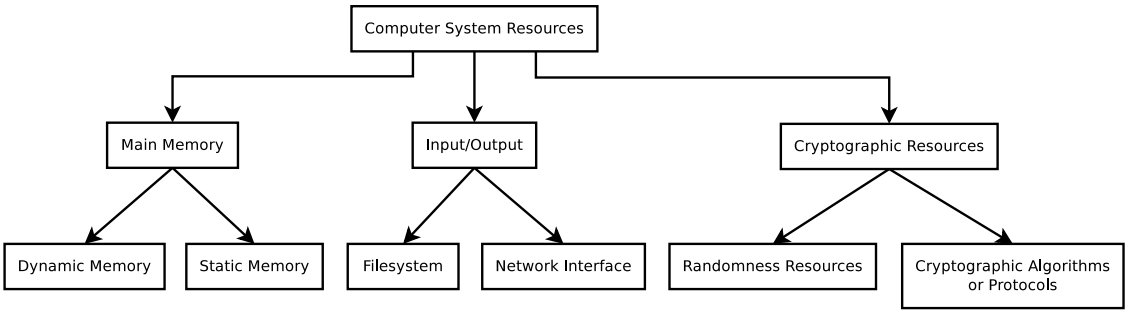


Figure 3.1: The Categories in the Theoretical Object Model of Computing

The object model of computing describes three categories of resources provided by the computer system: memory, input/output, and cryptographic resources. Each of the three categories has two sub-categories, which are displayed in the bottom row of the tree in figure 3.1.

3.1.2 Taxonomy of Software Vulnerabilities

When a process violates a constraint/assumption on a computer system resource, a vulnerability exists in software. This makes it possible to describe a software vulnerability by the associated constraint/assumption. Listing all possible constraints/assumptions will enumerate, by association, all possible software vulnerabilities. A classification is also possible, where the constraints/assumptions, and their associated vulnerabilities, can be grouped by the computer system resource they affect. The taxonomy of software vulnerabilities is an enumeration of constraints/assumptions, classified by the computer system resources they impose restrictions on, making the structure of the taxonomy the same as in figure 3.1.

If the list of constraints/assumptions cover every possible vulnerability, then a software process will have no vulnerabilities if it manages to use computer system resources without violating a constraint or assumption. This is useful for software audits because the converse is also true, if the usage of any of those resources violates a constraint/assumption listed in the taxonomy, a vulnerability is present. The work by Bazaz describes this approach [13].

A proactive approach that starts during requirements engineering can be achieved by determining the computer system resources the future software process will utilize. Then, a set of requirements are written to prevent all possible scenarios that the relevant constraints/assumptions can be violated. The result is a software process that contains no vulnerabilities from the taxonomy of software vulnerabilities.

3.2 Requirements Generation Process

The process for converting constraints/assumptions into requirements is critical to the success of vulnerability prevention. If an issue in a constraint/assumption is not correctly addressed by a set of requirements then it could result in a vulnerability. The process provided in this section created the entire list of software security requirements listed in 3.3, and must be used if any future constraints/assumptions are discovered.

A correct requirement can only be generated from a constraint/assumption if the problem is properly understood. Therefore, an assumption is made about the level of knowledge of anyone using this process; a user of the process must have enough knowledge to design and implement code that will prevent the constraint/assumption from being violated.

The process for generating software security requirements begins with a constraint/assumption. The process for generating a constraint/assumption is not part of this work, and is described by Bazaz [13]. Then, each constraint/assumption is given to the following process for requirements generation:

(1) **Determine how to violate the constraint/assumption.** This step may seem obvious, but determining the actual process to violate the constraint/assumption is harder once attempted. The step starts by taking an adversarial view of the constraint/assumption. Then, a methodical approach that manipulates a process into an exploitable statement is produced. Multiple approaches to violating a constraint/assumption may exist, in which case each of the attack approaches must be outlined. For example, the assumption "The series of random data being produced by the PRNG is unpredictable (assuming unpredictable seed)" has scenarios in which it is violated by a process. A PRNG that is not cryptographically secure could be used by the process, the process could use the PRNG before seeding, the internal state of the PRNG could be leaked by the process, and/or the process could continue using the PRNG after an additional re-seeding is required.

(2) **Determine the best method(s) to prevent the constraint/assumption from being violated.** The results from the previous step must now be analyzed to determine how the exploit is prevented in software. While the entire attack path generated in the previous step is useful to gain a good working knowledge on the subject, the most important aspect is looking at how the constraint/assumption is actually violated. Then, a statement is created about what the process must accomplish to prevent the violation from occurring. At least one statement must exist from each attack path established in the previous step, or violation scenarios will go unmitigated.

    Multiple statements may be required for a single attack path, because often times the method for preventing the violation will depend on the specific instance of a process in development. For example, proper data authentication can differ depending on whether the data is encrypted. Providing multiple prevention options

allows for more flexibility so a software security requirement is tailored to a projects needs. Prevention options are only available when there is no clear indication of which option is superior, and instead is solely a matter of discretion based on the project.

(3) **Based on method(s) to prevent a constraint/assumption, develop a security software requirement that is testable and verifiable.** The most important consideration in this step is ensuring that the requirement is clearly elucidating what the process must do, and what the process must not do. For example, "The process shall ensure that all addresses placed in a pointer variable are to memory locations other than its own" is specific, and clearly states what the process must do.

The next consideration is whether to group several of the prevention statements in the previous step, into a single requirement. Two cases will dictate the need for this, when the prevention statements must be done in a specific sequence, and when a prevention statement is a secondary issue caused by the misuse of another prevention statement. The best example of the second case is a requirement ensuring that a NIST approved encryption algorithm is used rather than a data obscurity algorithm. For an encryption algorithm to work, the key used for encryption must be kept a secret. Both of the prevention statements must be covered in a single requirement, because the second prevention statement was caused by the first prevention statement of encrypting through a NIST approved algorithm.

The final consideration is whether the requirement is testable or verifiable. A requirement is testable if output by the software process can determine that the requirement was properly implemented, and is verifiable if correct usage of the requirement will provide a specific artifact in the design or implementation. If the requirement is neither, it must be scrapped, and the process must start from the beginning.

(4) **Refine Requirement(s)** The requirement must cover exactly the prevention method from step 2, but no more, otherwise the requirement becomes less flexible. A good example is a memory allocation assumption, where a process may incorrectly assume that all memory allocation requests are successful. The requirements do not dictate how to check for a successful memory allocation, or what to do in the event of a failed memory allocation. But the requirement must cover the two methods that will prevent the assumption from being violated, checking for failed allocations, and only using memory that has been successfully allocated. Each requirement is scrutinized to ensure it succinctly covers the prevention statements while being flexible.

Once that is finished, any language in the requirement that connects it to a specific computer language or computer platform is removed so that the requirement is usable in any development scenario. Finally, the entire set of requirements is reviewed judiciously, to ensure that all the prevention statements from step 2 are correctly covered in the set of requirements generated for the constraint/assumption.

After the process has concluded on each constraint/assumption, the entire list of requirements are reviewed. The review process will reveal issues that are the result of a process only concerning a single requirement at a time. In many cases the requirements will be the same for different constraints/assumptions, which is perfectly acceptable. The review process must be strict to catch any conflicting requirements, which must be addressed immediately. The situation is rare, and is the result of inflexible requirements. In those cases, the conflicting requirements are edited to remove the conflict while still maintaining the ability to prevent the constraint/assumption violations. The completed review provides an entire set of requirements ready to prevent all constraint/assumption violations.

3.3 Results of Requirements Generation Process

The software security requirements are the result of the generation process in the previous section. Since the generation process used the constraints/assumptions in the taxonomy of software vulnerabilities, there is a limited scope to the software security requirements. The software security requirements only cover vulnerabilities present in a software process executing on an operating system. Potential vulnerabilities at the operating system level are not completely prevented by the software security requirements - although many of the potential issues are covered. The software security requirements are also limited in scope at a higher abstraction; information security concerns are not covered by the requirements. Information security requirements are discussed in 2.5.1, along with a comparison between software security requirements.

Throughout the software security requirements, there are two high level choices that are left to information security requirements: when to encrypt data, and data authentication algorithm selection. In the software security requirements, the choice for data encryption is accompanied with the phrase "sensitive data", which indicates that the requirement is only necessary when sensitive data is involved. Since the software security requirements are designed to be used without a security expert, it may be a challenge for development teams to determine if data is sensitive. Therefore, if the development team is unsure whether the data is sensitive, or does not know how to classify sensitive data, then the related encryption requirement must be used. This will prevent misuse because of a lack of information security requirements in the project. The second choice, data authentication, is always necessary in the software security requirements when dealing with data input/output. However, the cryptographic algorithms and protocols categories provides a choice of authentication algorithms. There are three choices for data authentication, one for non-encrypted data, one for encrypted data, and one for data requiring non-repudiation. The first two choices are already handled previously, so the choice is really whether the data needs non-repudiation. Luckily, non-repudiation is a well defined term, which can be looked up by the development team. Software developers will therefore be able to make this decision on their own. The information security aspects complicate the use of the software security requirements, but do not prevent them from being usable without security experts. However, a security expert would be valuable to complement the use of the requirements, and provide better insight to

which encryption and data authentication algorithms to use.

The remainder of this section lists all the software security requirements under the associated constraint/assumption. The support material is also provided, which includes a description of how the constraint/assumption is violated, and a brief statement on how to prevent the violation.

3.3.1 Main Memory

The main memory section contains resources that are run-time storage space for processes. Generally this will refer to the RAM of the system, and is used to hold either instructions or data for a software process. This resource is primarily misused by allowing input to affect memory designated for instruction usage, allowing an attacker to modify the instructions of the program. The main memory section is decomposed into two sub-categories, dynamic memory and static memory.

3.3.1.1 Dynamic Memory

Dynamic memory refers to both the execution stack, which contains local variables, parameters, and return addresses; and refers to the heap, which is storage space requested dynamically at run-time. The reasoning for not further breaking dynamic memory into subcategories containing the execution stack and heap is discussed by Bazaz [13]. The dynamic memory resource has more constraints/assumptions associated with it than static memory because all local variables and parameters are stored in dynamic memory, which are both a source of common misuse by software processes.

(1) ***Constraint/Assumption:*** Data accepted as input to a process and assigned to a buffer must occupy and modify only specific locations allocated in the buffer.

How the constraint/assumption is violated:

- External input to a process is copied into a buffer of smaller size in dynamic memory, and the external input is permitted to copy in memory locations past the allocated buffer space. This will write data to unknown memory locations, resulting in undefined behavior [36] [71].

How to prevent the constraint/assumption from being violated:

- If a process copies input into a buffer, the input must only be written to the space allocated to the buffer.

Software requirements that prevent the constraint/assumption from being violated:

- When copying data into a buffer, the process shall ensure that the data being copied does not exceed the bounds of the buffer.
- If data being copied into a buffer is expected to have a termination character to determine end of the data, then the process shall ensure that the termination character is present in the data, and ensure that the data being copied does not exceed the bounds of the buffer.

(2) ***Constraint/Assumption:*** The process will not interpret data present on the dynamic memory as executable code.

How the constraint/assumption is violated:

- A process variable (return address, exception pointer, etc.) is overwritten to contain a location in dynamic memory where an attacker has placed machine-level executable code. The control of the process continues when the instruction pointer is assigned the address that points to the memory location containing the executable instructions given by the attacker [36].

How to prevent the constraint/assumption from being violated:

- The operating system must be designed to prevent executing instructions residing in dynamic memory.
- The process must be designed to only execute code within code space.

Software requirements that prevent the constraint/assumption from being violated:

- The process must not contain code that either directly or indirectly causes the instruction pointer to load with an address outside of instruction space.

(3) ***Constraint/Assumption:*** Environment variables being used by the process have expected format and values.

How the constraint/assumption is violated:

- An attacker modifies the environment that will be used to execute the process. The process then inherits the entire environment (including dynamic library path locations) from the executing environment, even if the executing environment is another process. If the process assumes these environment variables are safe, the process is at risk of being manipulated by an attacker (in the case of the dynamic library path, they are automatically used by dynamically-linked processes) [79].

How to prevent the constraint/assumption from being violated:

- If an environment variable is used by the process, the environment variable is to be treated as external input, and therefore must be validated before being used.
- Before a process makes a call to execute another process, the current process must clear all environment variables, setting required ones to trusted values.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall ensure the value of an environment variable is in expected format before use.
- If an environment variable used by the process is expected to have a set of values, the process shall ensure the value of the environment variable is one of those expected values before use.
- Before invoking another process for execution, the invoking process shall clear all environment variables, and set environment variables required for execution with trusted values for the process being invoked.

(4) ***Constraint/Assumption:*** The process will be provided with the dynamic memory that it requests.

How the constraint/assumption is violated:

- A request for memory is made but the operating system cannot supply the process with the memory it requests. When the request fails, the process does not check to see if the memory request was successful, and the process continues executing as if the memory was allocated, potentially writing data to locations unknown locations, resulting in undefined behavior [71].

How to prevent the constraint/assumption from being violated:

- The process shall ensure that memory was successfully allocated before use.

Software requirements that prevent the constraint/assumption from being violated:

- After a process makes a request for memory, the process shall check to see that the memory was properly allocated, and only use memory that has been successfully allocated.

(5) ***Constraint/Assumption:*** Data present on the dynamic memory cannot be observed while the process is in execution.

How the constraint/assumption is violated:

- A process stores sensitive data in memory that should not be viewed. An attacker reads the process' memory or forces the process to write to swap space, and then views the sensitive data [79].

How to prevent the constraint/assumption from being violated:

- Keep sensitive data in memory only as long as necessary, and store the sensitive data in an encrypted format.
- Do not reallocate memory that contains sensitive data [79].

Software requirements that prevent the constraint/assumption from being violated:

- The process shall store sensitive data in memory only when necessary, and use a subroutine provided by the operating system to store the sensitive data in an encrypted format when the sensitive data is not in active use.
- The process shall not reallocate memory containing sensitive data.

(6) ***Constraint/Assumption:*** Data owned by the process and stored on the dynamic memory cannot be accessed after the process frees the memory.

How the constraint/assumption is violated:

- A process stores sensitive information in memory, and then releases the memory to the operating system without modifying the sensitive data. An attacker can then view the memory at any point before the memory is allocated to another process and overwritten.

How to prevent the constraint/assumption from being violated:

- Before a process releases memory that contains sensitive information, the process must erase the contents of the entire block of memory.
- Ensure that memory erasure techniques are not removed during optimizations [79].

Software requirements that prevent the constraint/assumption from being violated:

- If a process holds sensitive data in memory, the process shall write zeros to the entire block of memory before releasing the memory to the operating system. The process shall not use a built in subroutine for writing the zeros.

(7) ***Constraint/Assumption:*** A pointer variable being used by the process references a legal memory location.

How the constraint/assumption is violated:

- A pointer is set to a valid memory location, but the pointer becomes invalid because the memory location was freed by the process or the lifetime of the stack memory used for the pointer ends. The process then attempts to use the memory location as if it were still a valid location, which has an undefined result since the contents at the memory location are now unknown [71].

How to prevent the constraint/assumption from being violated:

- Prevent pointers to memory located on the heap from being dereferenced after the memory has been freed.
- Prevent pointers to stack memory from being dereferenced after the stack's lifetime ends.

Software requirements that prevent the constraint/assumption from being violated:

- After a pointer to a memory location on the heap is deallocated, the process shall set that pointer's value to NULL.
- Before dereferencing a pointer, the process shall ensure that the pointer's value is not set to NULL.
- The process shall only set a pointer's value to an address located on the stack, if the lifetime of the stack variable will end after the lifetime of the pointer.

(8) ***Constraint/Assumption:*** A memory pointer returned by the underlying operating system does not point to zero bytes of memory.

How the constraint/assumption is violated:

- The process makes a request for zero bytes of memory, which is undefined and generally will return a NULL pointer or a pointer to a location with zero bytes [39]. If the process writes any data to this location, an overflow occurs.

How to prevent the constraint/assumption from being violated:

- Never make a request to the operating system for zero bytes of memory.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall never make a request to the operating system for zero bytes of memory.

(9) ***Constraint/Assumption:*** A pointer variable being used by the process cannot reference itself.

How the constraint/assumption is violated:

- A pointer is set to its own memory location, so that any dereference to modify the value at the location will also change the value of the memory location. An attacker could then change the memory location with an assignment intended to change the value at that memory location.

How to prevent the constraint/assumption from being violated:

- When assigning a value to a pointer, ensure that the memory location is not for the pointer itself.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall ensure that all addresses placed in a pointer variable are to memory locations other than its own.

(10) ***Constraint/Assumption:*** Data accepted by the process must not be interpreted as a format string by the I/O routines.

How the constraint/assumption is violated:

- A string input is used within the process without first being checked against the expected format or value. An attacker supplies the process with input that contains special characters that break the functionality of the process because the special characters were not expected and not handled correctly, putting the process into an undetermined state. For example, if a string input is given to the C utility printf without being checked against expected format, unexpected formatting characters could be given in the input and passed to the printf utility. Because the printf call will not be expecting the additional formatting characters, it will perform read or write operations on memory locations in data space, providing the opportunity for an exploit [79] [71] [36].

How to prevent the constraint/assumption from being violated:

- Discard input that does not meet expected format and values.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall ensure the value of string input is in expected format before use.

- If an input has an expected set of values, the process shall ensure the value of the input is one of those expected values before use.

(11) ***Constraint/Assumption:*** The value of the integer/expression (signed & unsigned) accepted/calculated by the process cannot be greater (less) than the maximum (minimum) value that can be stored in the integer variable.

How the constraint/assumption is violated:

- Input to the process is crafted to cause a math operation to evaluate to a very large (small) number. If the resulting value of the math operation is larger (smaller) than the possible size the data type can store, the process is now in an undetermined state if the overflow (underflow) goes undetected. Improper memory allocation, or other exploitable behavior is now possible depending on how the value of the math operation is used later in the code [71].

How to prevent the constraint/assumption from being violated:

- Check for possible overflow conditions before evaluating an expression, and refuse further use of any input that was associated with producing the overflow evaluation.

Software requirements that prevent the constraint/assumption from being violated:

- Before an addition operation where both operands are positive, the process shall ensure that subtracting the left operand from the largest possible value is greater than or equal to the right operand. When the previous condition is not met, the process shall not perform the addition operation, and block any corresponding input from further use.
- Before an addition operation where both operands are negative, the process shall ensure that subtracting the right operand from the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the addition operation, and block any corresponding input from further use.
- Before a subtraction operation with unsigned numbers, the process shall ensure that the left operand is greater than the right operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.
- Before a subtraction operation where the left operand is nonnegative and the right operand is negative, the process shall ensure that adding the right operand to the largest possible value is greater than or equal to the left operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.

- Before a subtraction operation where the left operand is negative and the right operand is positive, the process shall ensure that adding the right operand to the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.
- Before a multiplication operation where both operands have the same sign, the process shall ensure that dividing the right operand by the largest possible value is greater than or equal to the left operand. When the previous condition is not met, the process shall not perform the multiplication operation, and block any corresponding input from further use.
- Before a multiplication operation where operands have different signs, the process shall ensure that dividing the right operand by the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the multiplication operation, and block any corresponding input from further use.
- The process shall ensure that a division operation never contains the largest negative value in the numerator, with a -1 in the denominator. When the previous condition is not met, the process shall not perform the division operation, and block any corresponding input from use.

(12) ***Constraint/Assumption:*** An integer variable/expression used by the process as the index to a buffer must only hold values that allow it access to the memory locations assigned to the buffer.

How the constraint/assumption is violated:

- A variable used by the process to store an index to a buffer is given a value that is not within the buffer bounds. Any read or writes that use the index variable will be writing to an unknown memory location, resulting in an undefined behavior.

How to prevent the constraint/assumption from being violated:

- Constrain variables used as an index to a buffer to only hold values within the buffer's range.

Software requirements that prevent the constraint/assumption from being violated:

- Before assigning a new value to a variable used as an index to a buffer, the process shall ensure the new value is within the buffers bounds.

(13) ***Constraint/Assumption:*** An integer variable/expression used by the process to indicate length/quantity of any object must not hold negative values.

How the constraint/assumption is violated:

- A variable used to indicate length/quantity of an object is set to a negative value. Any code expecting a nonnegative value is now in an undetermined state because numerical comparisons or data manipulation techniques may be invalid with a negative value for length/quantity. For example, if a variable that holds the length of a buffer is set to a negative value and the memory of the buffer is copied, the negative value will be passed to the memory copying subroutine. Since the subroutine may only accept unsigned values, the negative value will be converted to a positive number larger than the memory allocated to the buffer, creating the possibility of a buffer overflow exploit.

How to prevent the constraint/assumption from being violated:

- Variables used to hold length or quantity values for objects must only be assigned nonnegative values.

Software requirements that prevent the constraint/assumption from being violated:

- Before assigning a new value to a variable used to hold the length or quantity of an object, the process shall ensure that the new value is nonnegative.

3.3.1.2 Static Memory

The size of static memory does not change during execution, and is used to store the global variables in a process. There are less constraints/assumptions associated with static memory because static memory is used less frequently than dynamic memory during process execution.

(1) ***Constraint/Assumption:*** Data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer on the static memory.

How the constraint/assumption is violated:

- External input to a process is copied into a buffer of smaller size in static memory, and the external input is permitted copy in memory locations past the allocated buffer space. This will write data to unknown memory locations, resulting in undefined behavior [36] [71].

How to prevent the constraint/assumption from being violated:

- If a process copies input into a buffer, the input must only be written to the space allocated to the buffer.

Software requirements that prevent the constraint/assumption from being violated:

- When copying data into a buffer, the process shall ensure that the data being copied does not exceed the bounds of the buffer.
- If data being copied into a buffer is expected to have a termination character to determine end of the data, then the process shall ensure that the termination character is present in the data, and ensure that the data being copied does not exceed the bounds of the buffer.

(2) ***Constraint/Assumption:*** Data held on the static memory cannot be observed while the process is in execution.

How the constraint/assumption is violated:

- A process stores sensitive data in memory that should not be viewed. An attacker reads the process' memory or forces the process to write to swap space, and then views the sensitive data [79].

How to prevent the constraint/assumption from being violated:

- Processes do not require the use of static memory, and therefore sensitive data must not be stored in static memory.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall not store sensitive data in static memory.

3.3.2 Input/Output

A software process receives input for processing, and returns output to show the results of the processing. A computer system provides input/output resources for a software process so that it can achieve those necessary goals. As expected, there are many constraints/assumptions associated with input/output resources. Some inputs, such as command line, are not listed in a sub-category here because the usage of such resources has no associated constraints/assumptions, and instead has issues when combined with other resources. For example, the usage of command line input has no constraints/assumptions, but if the input from the command line is being copied into a buffer, a resource constraint now exists.

The input/output section, perhaps unexpectedly, only has two sub-categories for the reasons listed above. The network interface, and the filesystem have constraints/assumptions that must be taken into consideration on their own, and therefore require their own category.

3.3.2.1 Network Interface

Any software process that uses a network interface card to transmit or receive data falls into this sub-category. The network interface is generally used to communicate with another computer system, but can be used for inter-process communication on the same computer system. In all cases, the usage of a network interface is difficult, and the requirements below must be taken into consideration by any process that uses a network interface.

(1) ***Constraint/Assumption:*** The data received by the software process through network interface is neither read nor modified by anyone other than the intended recipient.

How the constraint/assumption is violated:

- Data transmitted over a network between two hosts may be handled by third parties. Any third party involved can read or modify data the two hosts are sending. A process that assumes the data it receives cannot be read or modified by a third party is in danger of being manipulated by an attacker [73].

How to prevent the constraint/assumption from being violated:

- Use a method that provides authenticity verification of data received over a network, to detect modifications by a third party.
- Encrypt sensitive data so that a third party eavesdropping on a network cannot decipher data sent between two hosts.

Software requirements that prevent the constraint/assumption from being violated:

- When receiving encrypted data over a network, the process shall decrypt and authenticate all data before its use, and reject data that does not authenticate or decrypt properly.
- When receiving unencrypted data over a network from another host, the process shall authenticate all data before its use, and reject data that does not authenticate properly.

(2) ***Constraint/Assumption:*** The data received by the software process through the network interface is from a legitimate client or peer or server and has expected format and length.

How the constraint/assumption is violated:

- Networks are susceptible to address spoofing where one machine sends data claiming to be from an address of another machine. An attacker can use this design flaw to spoof another host on a network, appearing to send legitimate data from the spoofed host. If a process expects, but doesn't enforce that the data from a host be in a specific format and length, an attacker can spoof the host and supply data that does not follow the format/length expected by the process, leaving the process in an undetermined state [73].

How to prevent the constraint/assumption from being violated:

- Authenticate a sending host before accepting any data from that host.
- If received data is expected to be in a particular format and length, check to ensure the received data meets those expected values, rejecting the data if it does not.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall authenticate the remote host to verify its identity before sending or accepting any data from that host.
- The process shall ensure data received from a remote host is in expected format before use.
- The process shall only send data in the expected format.

(3) ***Constraint/Assumption:*** The data sent by the software process via the network interface will not be read/modified before it reaches its destination.

How the constraint/assumption is violated:

- Data transmitted over a network between two hosts may be handled by third parties. Any third party involved can read or modify data exchanged between the hosts. A process that assumes the data it sends cannot be read or modified by a third party is in danger of being manipulated by an attacker [73].

How to prevent the constraint/assumption from being violated:

- Use a method that permits authenticity verification of sent data so the remote receiving host can determine if data sent over a network was modified by a third-party.
- Encrypt sensitive data so that a third party eavesdropping on a network cannot read data sent between two hosts.

Software requirements that prevent the constraint/assumption from being violated:

- If a process is sending sensitive data to another host on a network, the process shall encrypt and provide authentication for the data.
- When sending data over a network to another host, the process shall provide authentication for the data.

(4) ***Constraint/Assumption:*** The software process will be able to utilize the network interface to send and receive data

How the constraint/assumption is violated:

- The network is a system resource, and is subject to use by other processes on the system. A process which assumes that a port is available for its use, or that a network connection is available at any time, is susceptible to denial of service attacks if the assumption is incorrect.

How to prevent the constraint/assumption from being violated:

- Never assume that a port is available to listen for incoming data.
- Never assume that a connection is made to a remote host.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall check to see if the desired port is available for listening before waiting for connections on that port.
- The process shall check to see if a connection to a remote host was accepted before sending data to that host.

(5) ***Constraint/Assumption:*** The byte order of numerical data accepted from the network interface is the same as that of the host machine

How the constraint/assumption is violated:

- Different formats exist at the hardware level for storing numerical data. If two communicating hosts use a different format for numerical data, numerical values will be incorrect when read by the other host. The receiving process will enter an undefined state relative to the sending process because the response to the incorrect data is unknown.

How to prevent the constraint/assumption from being violated:

- Send numerical data over a network in network byte order.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall send all numerical data over a network in network byte order.
- The process shall expect all numerical data received over a network to be in network byte order.

3.3.2.2 Filesystem

While memory is used as a short-term storage for a software process, the filesystem resource is generally used as long-term storage. Many constraints/assumptions exist for the filesystem because of the permanent storage, and shared usage among software processes.

(1) ***Constraint/Assumption:*** Access permissions assigned to newly created files/directories are such that only the required principals have access to them.

How the constraint/assumption is violated:

- A process creates a file/directory with default permissions, giving read, write, or execute access to unintended users. The contents of the file can then be read or modified by these unauthorized users, giving them access to the operation of the process they should not have.

How to prevent the constraint/assumption from being violated:

- When creating a file/directory, restrict access and users to the file/directory. Only users that need read or write permissions for successful operation of the process are given those permissions [28].
- A file descriptor, and not the filename must be used to set access permissions [28] [71].

Software requirements that prevent the constraint/assumption from being violated:

- When creating a new file/directory, the process shall set the access permissions so the fewest number of users possible have access. The process shall set the access permissions using the file descriptor and not the filename.

(2) ***Constraint/Assumption:*** Access permissions of the files/directories being used by the process are such that only the required principals have access to them.

How the constraint/assumption is violated:

- A process expects that only specific users have permission to a file/directory, and therefore trusts the contents of the file/folder. If the permissions of the file/folder allow more than the expected user(s) access, the process can be manipulated by anyone that has been given unexpected access to the file.

How to prevent the constraint/assumption from being violated:

- Check file/directory permissions before use, only using files/directories with expected access permissions and ownership.

Software requirements that prevent the constraint/assumption from being violated:

- Before using a file/directory, the process shall check the ownership and the access permissions of the file/directory, and only use files/directories with expected ownership and access permissions.

(3) ***Constraint/Assumption:*** A file being created by the process does not have the same name as an already existing file.

How the constraint/assumption is violated:

- A process tries to create a file, but a file with that name already exists. The operating system then gives the existing file to the process, and the process modifies the existing file or the process fails because it does not have access permissions to the existing file. In the event of the former, the data in the file cannot be known after process execution since the original state of the file may have changed [28].

How to prevent the constraint/assumption from being violated:

- Never create a new file when a file with that path and name already exists

Software requirements that prevent the constraint/assumption from being violated:

- When creating a file, the process shall ensure the path and name are unique.

(4) ***Constraint/Assumption:*** A filename (including path) being used by the process is not a link that points to another file for which the user executing the process does not have the required access permissions.

How the constraint/assumption is violated:

- A process opens a file for reading or writing, but does not check whether the file is a link to a file that has different ownership or access permissions. If a process has elevated privileges compared to an attacker, the attacker could create and supply the process with a link to a file that denies the attacker access, but not the process. The attacker then gains an advantage when the process performs read or write operations on a file for which the attacker does not have proper access. For example, a setuid process is designed to read a file

in temporary storage (/tmp, etc.) which has global read/write access, and then display the contents of that file. If an attacker were to create a symbolic link to the password file (/etc/shadow, etc.), and then supplies the link to the process, the process will display the contents of the password file to the attacker who should not have access [79] [28] [71].

How to prevent the constraint/assumption from being violated:

- Refuse to open files through links [28], or compare access permissions of the link to the open file descriptor [71].

Software requirements that prevent the constraint/assumption from being violated:

- If the process does not need to open files/directories through links, the process shall use a file open subroutine that blocks the opening of files/directories through links.
- If the process needs to open files/directories through links, the process shall first check the access permissions of the link itself. Then, the process shall open the file, and use the file descriptor to check the access permissions of the file. If the access permissions of the file and link do not match, the process shall not use the file.

(5) ***Constraint/Assumption:*** A file created/populated by a principal other than the process, and then being used by the process, will have expected format and data.

How the constraint/assumption is violated:

- A process reads a file that could be modified by anyone, but assumes that the format and content are in an expected state. If the process does not ensure the format and content of the file are as expected, the process could potentially be manipulated when it tries to use an invalid file.

How to prevent the constraint/assumption from being violated:

- Check the format of any file used by the process, and only use files that follow expected format.
- Check the data of any file used by the process, and only use files containing expected data.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall ensure that data contained in a file is in expected format before use.

- If a file used by the process is expected to have a set of data, the process shall ensure that the expected data is in that file before use.

(6) ***Constraint/Assumption:*** A file being used by a process cannot be observed/-modified/replaced by any other process while the initial process is in execution.

How the constraint/assumption is violated:

- A process opens a file for reading or writing but no guarantee is given that the file cannot be read or modified during this time. The contents of the file could be changed while the process is in execution, putting the process and the file contents into an undetermined state if it assumes it is the only one that can modify the file.

How to prevent the constraint/assumption from being violated:

- Request that the operating system lock the file, preventing reading or modification of the file until the file is longer being used.

Software requirements that prevent the constraint/assumption from being violated:

- After opening a file, but before reading or writing to that file, the process shall request the operating system lock the file, using the file descriptor returned by the file open operation.
- Before closing a file, but after all reading and writing operations have been completed on that file, the process shall request the operating system unlock the file, using the file descriptor given to the previous lock file operation.

(7) ***Constraint/Assumption:*** A file/directory being used by the process and stored on the filesystem (information used by the process over multiple runs) cannot be observed/modified/replaced in-between these runs.

How the constraint/assumption is violated:

- A process stores information it needs on the filesystem in a file. The filesystem is a shared resource, and can be read or modified by an attacker even if the file was set with proper access permissions. A process that assumes data it writes cannot be read is in danger of leaking sensitive information. A process that assumes data it writes cannot be modified is in danger of being manipulated by an attacker.

How to prevent the constraint/assumption from being violated:

- Encrypt sensitive data before writing to the filesystem.

- Use a method for data authentication to confirm the authenticity of files used.

Software requirements that prevent the constraint/assumption from being violated:

- If writing sensitive data to a file, the process shall encrypt and provide authentication for the data.
- The process shall provide authentication for unencrypted data in files.
- The process shall only use data from files that have verified authenticity.

(8) ***Constraint/Assumption:*** Data held by files owned/used by the process must not be accessible after the process deletes them.

How the constraint/assumption is violated:

- When a process requests that the operating system delete a file, the operating system marks the space the file occupied as free, but does not overwrite or erase the data immediately on deletion. An attacker could read any file that the process requested for deletion until the operating system re-uses the space for another file [12].

How to prevent the constraint/assumption from being violated:

- If the contents of the data, even when encrypted, should no longer be accessible, the only secure erasure method available to user space programs is to overwrite the data on the entire partition once [82] [12] [40].

Software requirements that prevent the constraint/assumption from being violated:

- After deleting a file, if the data is too sensitive to be left on disk even in an encrypted format, the process shall write zeros to the entire hard-drive partition that contained the file. The process shall take into consideration that the procedure could also remove the operating system from disk.

(9) ***Constraint/Assumption:*** The process must be provided with the filesystem space that it requests.

How the constraint/assumption is violated:

- A process uses a filesystem shared by other processes, but assumes space will always be available for storage. When attempting to write to a file, there is no check for whether the write was successful. Because the process makes an assumption that space will always be available, it continues to operate assuming the data was written. If the process attempts any further reads or writes, the state of the process is now unknown because the previous write was unsuccessful.

How to prevent the constraint/assumption from being violated:

- After every write attempt, check to ensure that the write was successful.

Software requirements that prevent the constraint/assumption from being violated:

- After attempting a write to a file, the process shall ensure that the write was successful.

(10) ***Constraint/Assumption:*** A file having proprietary or obscure format cannot be understood or modified.

How the constraint/assumption is violated:

- A process uses a proprietary format that is only known to the developers of the process. An assumption is then made that only the process is capable of interpreting the information contained within the file, and that only the process is capable of intelligently modifying the file. However, an attacker could reverse engineer the format, making it possible for an attacker to read and modify data thought to be secure.

How to prevent the constraint/assumption from being violated:

- Encrypt sensitive data before writing to the filesystem.
- Use a method for data authentication to confirm the authenticity of files used.

Software requirements that prevent the constraint/assumption from being violated:

- If writing sensitive data to a file, the process shall encrypt and provide authentication for the data.
- The process shall provide authentication for unencrypted data in files.
- The process shall only use data from files that have verified authenticity.

3.3.3 Cryptographic Resources

The inclusion of cryptographic resources as a computer system resource is not an obvious one since there is no physical connection to a part of a computer system as with the other categories. The inclusion was necessary because software processes often make use of cryptographic resources to prevent the violation of constraints/assumptions in other categories. The resources provided by this category include authentication, encryption, among other privacy and control concerns. This category is further decomposed into two more categories, randomness resources, and cryptographic algorithms and protocols.

Unique to this section is the heavy reliance on National Institute for Standards and Technology (NIST) recommendations. Instead of providing specific recommendations for encryption algorithms, etc., the software security requirements dictate that a process use a current recommendation from NIST when necessary for long term viability. This will enable the software security requirements to remain relevant, accurate, and usable even as cryptographic algorithms in use during the time of writing become obsolete. All references to NIST include explicit details as to what the recommendations are called, and the name of the current document giving those recommendations.

3.3.3.1 Randomness Resources

The sub-category randomness resources is separated from cryptographic algorithms/protocols because the resources provided in that category often make heavy use of randomness. Despite the reliance that cryptographic algorithms/protocols have on randomness, it is the one resource that is the most misunderstood, and so is often misused. Providing unguessable numbers on a deterministic machine is the primary concern in this sub-category, and is the concern that produces all the constraints/assumptions here.

(1) ***Constraint/Assumption:*** The series of random data being produced by the PRNG is unpredictable (assuming unpredictable seed).

How the constraint/assumption is violated:

- An attacker does not know the seed value to a PRNG, but knows a series of previous values produced by the PRNG. A weak PRNG, or a cryptographically secure PRNG that has not been re-seeded after the suggested amount of usage will produce a series of predictable values, meaning an attacker can guess upcoming values based on the knowledge of the previous values [79] [11]. Any cryptographic algorithm or protocol that relies on unpredictable data for its security is susceptible to attack when it receives predictable data.
- An attacker does not know the seed value to a PRNG, but knows the internal state of the PRNG. Once the PRNG algorithm is determined by the attacker, the generated values can be duplicated using the same internal state. Any cryptographic algorithm or protocol that relies on unpredictable data for its security is susceptible to attack when it receives predictable data [11].

How to prevent the constraint/assumption from being violated:

- Use a cryptographically secure PRNG algorithm recommended by NIST. Cryptographically secure PRNG recommendations are in FIPS 140 Annex C [22].
- Seed the cryptographically secure PRNG before generating any values.
- Reseed the cryptographically secure PRNG after the maximum allowable number of generated values for that algorithm has been reached.

- Keep the internal state of the cryptographically secure PRNG a secret.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall use a random number generation algorithm that is currently recommended by NIST (as of publication, the recommended random number generators are described in SP 800-90 [11], FIPS 186-3 [63], ANSI X9.31 Appendix A.2.4 [6], and ANSI X.9.62-1998 Annex A.4 [7]).
- The process shall seed the cryptographically secure PRNG before any values are generated.
- The process shall reseed the cryptographically secure PRNG when the NIST defined maximum number of generated values per seed for that algorithm has been reached.
- The process shall limit the access to the the internal state of the cryptographically secure PRNG to the fewest users possible.

(2) ***Constraint/Assumption:*** The seed being used by the PRNG is unpredictable.

How the constraint/assumption is violated:

- PRNG's rely on seed values to produce pseudo-random numbers. If the seed value is predictable there is a chance that the value generated by the PRNG is also predictable, because PRNG's are pseudo-random [11]. Since any cryptographic algorithm or protocol that relies on unpredictable data for its security is susceptible to attack when it receives predictable data, any seed value that is predictable may weaken the security of those algorithms or protocols.

How to prevent the constraint/assumption from being violated:

- Properly seed the cryptographically secure PRNG with the required amount of entropic data [11] [79].
- Keep the seed value given to the cryptographically secure PRNG a secret [11].

Software requirements that prevent the constraint/assumption from being violated:

- The process shall use a process recommended by NIST to seed the cryptographically secure PRNG in use, and discard the seed value after seeding.
- The process shall provide the seed with enough entropy to satisfy NIST requirements for the cryptographically secure PRNG in use.

(3) ***Constraint/Assumption:*** The process will have easy access to entropic data on a computer system.

How the constraint/assumption is violated:

- Unpredictable data is required for some cryptographic algorithms or protocols to provide their security. Cryptographically secure unpredictable data is produced by using entropic data, where entropy is the measure of uncertainty associated with a value. However, computers are deterministic machines, which provides a difficulty in gathering good entropic data. A process that exhausts available entropic data will not be capable of producing cryptographically secure random numbers, leaving cryptographic algorithms or protocols that rely on unpredictable data susceptible to attack [79].

How to prevent the constraint/assumption from being violated:

- Do not use entropy directly in cryptographic algorithms or protocols, and instead seed a cryptographically secure PRNG, giving the cryptographic algorithms or protocols the data generated by the cryptographically secure PRNG.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall use entropic data to seed a cryptographically secure PRNG, and provide cryptographic algorithms or protocols that require unpredictable data values generated from the cryptographically secure PRNG.

(4) ***Constraint/Assumption:*** The process will be able to accurately estimate entropy of a data set.

How the constraint/assumption is violated:

- Entropy is the measure of uncertainty in data, and is required to provide security in some cryptographic algorithms or protocols, particularly cryptographically secure PRNGs. Entropy is difficult to estimate because computers are deterministic machines, and therefore can be predictable. While it is possible to receive data that is unpredictable to an outside observer, overestimates in entropy, especially with sources that contain zero entropy, can result in data that is more predictable than thought [11] [79]. This will make the cryptographic algorithms that rely on entropy susceptible to attack because their security has been weakened with predictable data

How to prevent the constraint/assumption from being violated:

- Take security experts recommendations on entropy estimates for a particular source: (a) 1 bit for each keyboard character pressed if it was different than the previous, (b) log2(time) - 1 bits of entropy for the timing between keyboard events starting with the third event, (c) 1 bit of entropy per mouse event if local

events cannot be detected between user accounts, and (d) 0.5 bits of entropy for the timing between incoming packets if a clock resolution of milliseconds or greater is available [79].

- For entropy sources not listed by an expert, divide any estimate of entropy by 8, to prevent an overestimation [79].

Software requirements that prevent the constraint/assumption from being violated:

- The process shall estimate 1 bit of entropy for each character pressed on the keyboard, but only if the character is different than the previous character.
- The process shall estimate log2(time) - 1 bits of entropy for the timing between local keyboard events, starting with the third event.
- If mouse events from a user account cannot be detected from another user account, the process shall estimate 1 bit of entropy per local mouse event.
- If a clock resolution of milliseconds or greater is available, the process shall count 0.5 bits of entropy for the timing between incoming packets, starting with with the third event.
- For entropy estimates that are not from keyboard events, network packet timings, or mouse events, the process shall divide the estimate of entropy from a data set by 8.

(5) ***Constraint/Assumption:*** User selected passwords/keys will have a sufficient amount of entropy.

How the constraint/assumption is violated:

- Entropy is the measure of uncertainty in data. Entropy increases as the length of the password/key increases or the set of possible characters increases. When a user selects a password/key, the tendency is to create a short password only using alphabetic characters, limiting the amount of entropy to the password/key [45]. Furthermore, user often choose words contained in the dictionary, decreasing the entropy even more since the practice limits the number of possible passwords [79]. This decrease in entropy increases the possibility of the password/key being guessed, and is susceptibility to a brute-force or dictionary-based attack. Any cryptographic algorithm or protocol that relies on a password/key to provide security is susceptible to attack when a user selects a password/key that has low entropy.

How to prevent the constraint/assumption from being violated:

- Do not allow users to generate passwords, and instead have the computer randomly generate passwords/keys.

- If the user has to remember the key, use a NIST recommended automated password generator [57].
- If a user generated password will be used, use a password-key deriviation function described in PKCS #5 [43] [79].

Software requirements that prevent the constraint/assumption from being violated:

- If a user does not have to remember a password/key required for a cryptographic algorithm or protocol, the process shall use a cryptographically secure random data generator to generate the password/key.
- If the user is required to remember a password for a cryptographic algorithm or protocol, the process shall use a random password generator recommended by NIST to provide passwords for users (as of publication, recommended automated password generators are described in FIPS 181 [57]).
- If a user generated password will be used in a cryptographic algorithm or protocol, the process shall use a password-key derivation function described in PKCS #5 to create a cryptographic key from a user supplied password.

(6) ***Constraint/Assumption:*** If two different seeds are provided to the PRNG, it is computationally infeasible to produce the same series of data both times.

How the constraint/assumption is violated:

- A weak PRNG is used to generate random data that may produce the same series of data given two different seed values. An attacker can get a PRNG to generate multiple series of data, and store the values. The more series of data the attacker stores, the higher the chance that the same series of data will be generated, even with different seed values [79]. Any cryptographic algorithm or protocol that relies on unpredictable data for its security is susceptible to attack when its receives predictable data, meaning use of a PRNG that produces the same series of data given two seeds is insecure.

How to prevent the constraint/assumption from being violated:

- Use a cryptographically secure PRNG algorithm recommended by NIST. Cryptographically secure PRNG recommendations are in FIPS 140 Annex C [22].

Software requirements that prevent the constraint/assumption from being violated:

- The process shall use a random number generation algorithm that is currently recommended by NIST (as of publication, the recommended random number generators are described in SP 800-90 [11], FIPS 186-3 [63], ANSI X9.31 Appendix A.2.4 [6], and ANSI X.9.62-1998 Annex A.4 [7]).

(7) ***Constraint/Assumption:*** Given that the PRNG is continuously producing random data, it is computationally infeasible to produce the same sequence of random data after some time.

How the constraint/assumption is violated:

- A weak PRNG is used to generate random data, which after some time will begin to repeat the same series of data generated previously. An attacker can record the series of data output, and use this recorded data to predict future values [79]. Any cryptographic algorithm or protocol that relies on unpredictable data for its security is susceptibility to attack when it receives predictable data, meaning use of a PRNG that repeats series of values is insecure.

How to prevent the constraint/assumption from being violated:

- Use a cryptographically secure PRNG algorithm recommended by NIST. Cryptographically secure PRNG recommendations are in FIPS 140 Annex C [22].

Software requirements that prevent the constraint/assumption from being violated:

- The process shall use a random number generation algorithm that is currently recommended by NIST (as of publication, the recommended random number generators are described in SP 800-90 [11], FIPS 186-3 [63], ANSI X9.31 Appendix A.2.4 [6], and ANSI X.9.62-1998 Annex A.4 [7]).

3.3.3.2 Cryptographic Algorithms and Protocols

The cryptographic algorithms and protocols sub-category provides many unique services to a process, including confidentiality, authentication, and non-repudiation. Those services are not required on their own, and in all cases are required because of constraints/assumptions imposed on resources in other categories. For example, software processes that are transmitting confidential network data will only be successful, if none of the constraints/assumptions in this category are violated. This sub-category is very closely related to the randomness resources sub-category, because many of the cryptographic algorithms and protocols are only secure when given unpredictable data.

(1) ***Constraint/Assumption:*** Random data being used by the cryptographic algorithm/protocol is unpredictable.

How the constraint/assumption is violated:

- A process uses random data to thwart attackers from guessing the next number that will be used by a cryptographic algorithm or protocol. If a process is using random data that is predictable, an attacker can gain an advantage by predicting the next series of numbers produced. Because the cryptographic algorithm or protocol provides security by using random data, if the cryptographic algorithm or protocol uses predictable data, it is susceptible to attack.
- Random data being used by a cryptographic algorithm/protocol is unpredictable because it is generated by a cryptographically secure PRNG. However, the data is given to another party insecurely, providing the possibility for an attacker to capture this data, leaving any cryptographic algorithm/protocol relying on unpredictable data insecure.

How to prevent the constraint/assumption from being violated:

- Use a cryptographically secure PRNG algorithm recommended by NIST. Cryptographically secure PRNG recommendations are in FIPS 140 Annex C [22].
- If a secret key has to be established between two peers, make sure the key is established securely using a NIST recommended key establishment scheme. Key establishment scheme recommendations are in FIPS 140 Annex D [23].

Software requirements that prevent the constraint/assumption from being violated:

- The process shall use a random number generation algorithm that is currently recommended by NIST (as of publication, the recommended random number generators are described in SP 800-90 [11], FIPS 186-3 [63], ANSI X9.31 Appendix A.2.4 [6], and ANSI X.9.62-1998 Annex A.4 [7]).
- The process shall establish secret keys between two parties using a key establishment scheme that is currently recommended by NIST (as of publication, key establishment schemes are listed in SP 800 56-A [10], and SP 800 56-B [9]).

(2) ***Constraint/Assumption:*** The length of the key being used by the cryptographic algorithm or protocol is sufficient.

How the constraint/assumption is violated:

- Any key used for cryptographic algorithms or protocols is susceptible to a brute-force attack. A key length that is too short by current standards is in risk of being compromised, which in turn will compromise the security provided by the cryptographic algorithm or protocol used by the process [73].

How to prevent the constraint/assumption from being violated:

- The length of a key must be long enough to provide computational security, making attacks technically infeasible by modern computing hardware [73].

Software requirements that prevent the constraint/assumption from being violated:

- The process shall use the current NIST recommendation for key length for the cryptographic algorithm or protocol in use.

(3) ***Constraint/Assumption:*** The hashing algorithm will not produce same hash for two different inputs.

How the constraint/assumption is violated:

- A hashing algorithm accepts an infinite number of possibilities as input, and produces a finite number of possibilities as output. All hashing algorithms have potential for a collision where the same hash is produced with two different inputs [79] [68]. A process that trusts a weak hashing algorithm that is more likely to produce a collision will be susceptible to attack when an attacker supplies an input that produces the same hash output as a different input, leaving the process unable to detect modifications to data.
- A hash algorithm always produces the same output when given the same input. Attackers use this knowledge to their advantage by pre-computing hash values, so that when given an output of a hash function, the input can be determined by looking at their pre-computation list [79]. A process that uses a hash algorithm to encode sensitive information can leak this information if hash values are compared against a pre-computed list of hashes.
- A hash algorithm is being used to authenticate data, by concatenating a secret key to a message before hashing. If the length of the entire concatenation is known, and the value of the hash is known, then an attacker can concatenate any data to the end of the original message, and still produce a hash value that authenticates [42]. Any program relying on the ability of a hash function alone to verify data authenticity cannot be considered secure.

How to prevent the constraint/assumption from being violated:

- Use a NIST recommended hash function. Hash function recommendations are in FIPS 140 Annex A [21].
- Strings given to a hash algorithm must have random data concatenated to the string, so pre-computation is more difficult [79].
- The result of the hash is hashed a second time, and the result of the second hash value is used [79].

Software requirements that prevent the constraint/assumption from being violated:

- The process shall use a hash algorithm that NIST currently recommends (as of publication, recommended hash functions are described in FIPS 180-3 [62]).
- Before giving data to a hash function, the process shall generate a random string of data whose length is equal to the length of the hash function output, then pad the end of the random string with zeros to equal the length of the internal block size of the hash function. The process shall concatenate this newly generated string before and after the data that was destined for the hash function, then give this large string concatenation to the hash function. The process shall then give the result of the hash to the hash function a second time, and discard the result of the first hash, only using the result of the second hash. The process shall store the random string with the hash result so the process can be duplicated with the same random string for any required verification.

(4) ***Constraint/Assumption:*** The process cannot use encryption to ensure data integrity or data authentication.

How the constraint/assumption is violated:

- Encryption provides confidentiality of data, but provides no assurance to integrity or authentication because the ciphertext can modified, and still properly decrypt. A process that relies on encryption for integrity or authentication can be manipulated by an attacker that modifies the encrypted data. The process will be unable to detect these modifications using decryption alone.
- A proper data authentication method is being used to verify the authenticity of data sent between two hosts. The key used in the authentication process is static, so an attacker can capture previous data, including the authentication value, and re-send the information as if it were new [45] [79]. A authentication method that uses a static key between two hosts cannot determine new requests, from old ones.

How to prevent the constraint/assumption from being violated:

- Use a method that provides verification to the authenticity and integrity of data. The method required depends on the circumstance. For data that requires non-repudiation, use a NIST recommended signature algorithm. For data that does not require non-repudiation: (a) use a NIST recommended message authentication algorithm if data is unencrypted, or (b) if encrypted, use a NIST recommended block cipher mode that supports authenticated encryption. All data authentication recommendations are in FIPS 140 Annex A [21].
- The key used to authenticate the data must be kept a secret. For signature algorithms, this means keeping the private key a secret.
- If authenticating data between two hosts, concatenate a non-repeating increasing number value with the authentication key. That concatenated string is used

by the authentication routine as the actual key. The host checking authentication enforces that the number is larger than the previous number [79].

Software requirements that prevent the constraint/assumption from being violated:

- If non-encrypted data needs to be verified for integrity and authenticity but not verified as to what party generated the data, the process shall use a message authentication algorithm that is currently recommended by NIST (as of publication, recommended message authentication algorithms are described in FIPS 113 [56], FIPS 198-1 [60], and SP 800-38B [31]). The process shall keep the key used for authentication a secret.
- If encrypted data needs to be verified for integrity and authenticity but not verified as to what party generated the data, the process shall use a block cipher mode when encrypting the data that supports authenticated encryption (as of publication, recommended cipher modes that support authenticated encryption are described in SP 800-38C [30], and SP 800-38D [32]). The process shall keep the key used for authentication a secret.
- If non-repudiation is required for encrypted or non-encrypted data to prove that the data came from only one source, the process shall sign the data with a signature algorithm currently recommended by NIST (as of publication, recommended signature algorithms are described in FIPS 186-3 [63]). The process shall keep any private keys used by the signature algorithm a secret.
- If authentication of encrypted or unencrypted data is being used by two remote hosts, the sending process shall concatenate an increasing number value to the authentication key before generating an authentication value. The sending process shall store the number value with the authentication value. The receiving process shall use this number during authentication, and ensure this number value increases with every authentication check. The process shall ensure the data space for the number value is large enough to prevent repeat numbers.

(5) ***Constraint/Assumption:*** The process cannot use a key more than once for a stream cipher.

How the constraint/assumption is violated:

- Stream ciphers perform an exclusive-or operation between the data that needs to be encrypted, and a keystream. The exclusive-or operation works at the bit level, comparing two inputs bit-by-bit. Each bit of the output of an exclusive-or operation is zero, unless one and only one of the two input bits is 1. This process not only encrypts data, but also decrypts data since an exclusive-or operation with the encrypted data and the same keystream will result in the original data. If the same key is used to encrypt multiple pieces of data, the same keystream will be generated, leaving the possibility for a exclusive-or operation that can

be performed on two sets of data encrypted with the same keystream. The result of an exclusive-or operation on two sets of encrypted data from the same keystream is the same as performing an exclusive-or operation on the original non-encrypted sets of data. Even with partial knowledge of the contents from one of the original non-encrypted sets of data, analysis can be done to decrypt a larger portion of the encrypted data [19]. Future data encrypted with that key could also be decrypted using this method, compromising the stream cipher process.

How to prevent the constraint/assumption from being violated:

- The key used by the stream cipher to generate a keystream must be unpredictable [19].
- A key must never be used twice by the stream cipher to generate a keystream.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall provide an unpredictable key to a stream cipher for keystream generation.
- The process shall never provide a duplicate key to a stream cipher for keystream generation.

(6) ***Constraint/Assumption:*** The process cannot use one time pads to encrypt large quantity of data.

How the constraint/assumption is violated:

- A one time pad is an encryption algorithm in which a key is combined with data to create the ciphertext. The design requires a random key that is equal to the length of the input data. If the input data is large, a large amount of random data is required which is not feasible. Without random data for a key, the one-time pad method can no longer be considered secure.

How to prevent the constraint/assumption from being violated:

- Use a NIST recommended block cipher and block cipher mode, where a limited amount of random data is required for encryption. Block cipher and block cipher mode recommendations are in FIPS 140 Annex A [21].

Software requirements that prevent the constraint/assumption from being violated:

- The process shall encrypt sensitive data using a block cipher that is currently recommended by NIST (as of publication, recommended encryption algorithms are described in FIPS 197 [59], FIPS 185 [58], and SP 800-67 [61]), using a block cipher mode currently recommended by NIST (as of publication, recommended block cipher modes are described in SP 800-38A [29], SP 800-38B [31], SP 800-38C [30], and SP 800-38D [32]). The process shall keep the encryption key a secret.

(7) ***Constraint/Assumption:*** The process cannot use keys that are self reported by a client or a server.

How the constraint/assumption is violated:

- Authentication is required between machines because spoofing is possible, where one machine falsely claims to be another machine. As a proof of identity, a process receives a key from a peer to use in an cryptographic algorithm or protocol. If the process cannot verify the identity of the peer with a trusted third party, an attacker could be spoofing this peer, and provide a key that the attacker can use to steal information from the process.

How to prevent the constraint/assumption from being violated:

- Use an authentication scheme, recommended by NIST, that verifies keys of peers. Recommended authentication schemes will be published in future standards.
- Designate a trusted third party that can verify any reported keys.

Software requirements that prevent the constraint/assumption from being violated:

- The process shall follow NIST guidelines for key management, to validate any key reported by another machine before use (as of publication, future key management guidelines are described across several documents, SP 800-57 [8], SP 800 56-A [10], and SP 800 56-B [9]).
- The process shall designate a trusted third party for key verification.

(8) ***Constraint/Assumption:*** The process cannot use obfuscation instead of encryption to ensure confidentiality.

How the constraint/assumption is violated:

- A process uses obfuscation to conceal the true value of its data. The process the application uses to achieve this obfuscation can be reverse engineered by an attacker, and therefore any sensitive data stored through obfuscation cannot be considered safe.

How to prevent the constraint/assumption from being violated:

- Use a NIST recommended block cipher and block cipher mode. Block cipher and block cipher mode recommendations are in FIPS 140 Annex A [21].

Software requirements that prevent the constraint/assumption from being violated:

- The process shall encrypt sensitive data using a block cipher that is currently recommended by NIST (as of publication, recommended encryption algorithms are described in FIPS 197 [59], FIPS 185 [58], and SP 800-67 [61]), using a block cipher mode currently recommended by NIST (as of publication, recommended block cipher modes are described in SP 800-38A [29], SP 800-38B [31], SP 800-38C [30], and SP 800-38D [32]). The process shall keep the encryption key a secret.

(9) ***Constraint/Assumption:*** The process cannot store keys/passwords in clear text.

How the constraint/assumption is violated:

- A process stores a key/password in clear text that is used for authentication. If an attacker finds the password/key, authentication can no longer be considered secure, since the attacker can use that same password/key to properly authenticate.

How to prevent the constraint/assumption from being violated:

- Encrypt or hash with a salt [79] any user selected password stored by the process depending on the circumstance.

Software requirements that prevent the constraint/assumption from being violated:

- If storage is required for a password/key for later retrieval, the process shall encrypt the password/key with a second password/key, immediately discarding the original password/key. The process shall not store the second password/key with the encrypted password/key.
- If storage is required for a password/key for later validation, the process shall give the string to a secure hash procedure. The process shall discard the password after providing it to the secure hashing procedure, only storing the results of the secure hashing procedure.

3.4 Survey with Subject Matter Experts

The software security requirements given in the previous section are designed to be (a) testable and verifiable, (b) clear, concise and non-ambiguous, (c) implementable by software engineers without security knowledge, (d) usable in modern development lifecycles, and (e) capable of preventing modern software vulnerabilities when used correctly in development. Before these goals can be considered achieved by this research, an evaluation process needs to substantiate these claims. The best evaluation method would be to incorporate the software security requirements in a real development project to determine their successfulness at preventing vulnerabilities. Unfortunately, that evaluation approach has many complications due to time considerations, and vulnerability detection. Instead, this research focused on a survey targeted at experts in security and software engineering, in hopes that they substantiate the goals of this research. Three security experts reviewed all the requirements, while 11 software engineering experts reviewed a partial set of the requirements. The software engineers were divided so that the software security requirements provided in the randomness, and the cryptographic algorithms and protocols sections each had three reviewers, while the other resource categories had four software engineering reviewers.

The surveys for both the software engineers and the security experts were conducted through a web browser using a custom application written in php. Each survey was separated into sections based on the computer resources, enabling subject matter experts to submit a section without having to complete all the responses required at once. Since static memory only has two software security requirements it was grouped with dynamic memory, and given to the experts as the category "memory". Therefore, when discussing the results of the survey, five computer resource categories will be mentioned: (a) memory, (b) networking, (c) filesystem, (d) randomness, and (e) cryptographic algorithms and protocols. Each survey contained statements linked to a goal of the software security requirements, and the response to the statement was limited to a quantitative value. A high quantitative response meant the surveyor felt the statement is more likely to be true, and likewise the associated goal. All responses to the survey were anonymous, and cannot be tracked to an individual.

3.4.1 Software Engineers Survey

The major focus of the survey for software engineers is to substantiate the goal of non-security experts being able to implement each of the software security requirements. Software engineers were questioned to determine how well each requirement achieved that goal, with satisfactory results, as shown below. Then the software engineers, as with the security experts, were asked their opinion on whether each requirement is clear, concise, and non-ambiguous. This goal is evaluated across both surveys to determine if terms not familiar to most software engineers would negatively impact the perceived conciseness of the requirement. Lastly, software engineers were surveyed on two other intended goals of the software security requirements; whether each requirement is testable/verifiable, and whether each requirement can be incorporated into current software development lifecycles.

In all, software engineers responded to four statements for each software security requirement. Each of the statements is linked to a goal discussed in the previous paragraph, and the response provided by the software engineer evaluates how well they perceived the goal was achieved. So that quantitative results were possible, the responses were limited to a scale from 1 to 10, where a higher score corresponds to a statement that is thought to be more true, and therefore better meeting the intended goal. The format, and the statements given to software engineers are shown in appendix A.1. All software engineers that participated in this survey either have a degree in computer science or computer engineering, or have completed a software engineering course.

3.4.1.1 Results from Software Engineers Survey

The survey results from software engineers are evaluated in two ways: on a per requirement basis, and on a per resource category basis. In each case the responses are averaged by the statement that produced the response, leading to 368 quantitative values when evaluated on a per requirement basis, and 20 quantitative values when evaluated on a per resource category basis. Since the responses are limited to numbers between 1 and 10, any average that is above 5.5 can be considered closer to achieving the corresponding goal, while an average below 5.5 can be considered further from achieving the goal.

When evaluating the survey on a per requirement basis, only 23 of the possible 368 responses have an average less than 5.5, resulting in the software security requirements achieving their intended goals 93.8% percent of the time. A majority of the low responses came from the goal of a software engineer being able to implement a software security requirement. All the averages on a per requirement basis are included for reference in appendix B.1.

Goal	Memory	Networking	Filesystem	Randomness	Cryptographic Algorithms
Implementable	8.87	8.2	8.06	5.45	5.51
Concise	9.24	8.91	9.21	6.92	8.16
Testable / Verfiable	9.18	9.77	9.66	7.14	8.12
Incorporable into Lifecycles	9.42	9.5	9.66	7.8	8.49

Table 3.1: Software Engineers Evaluation of the Software Security Requirement Goals

Evaluating the software engineers survey on a per resource category basis reveals more information about how the surveyors perceive the software security requirements. Table 3.1 shows the average response by resource category, that the software engineers gave on the goals of the software security requirements. The columns of the table correspond to the resource categories, while the rows correspond to four of the software security requirement

goals. The values from the table are shown in a bar graph in figure 3.2 to highlight the result differences between the categories. Each bar color represents a goal of the software security requirements, and the bars are grouped by the resource category. The heights of the bar represent well software engineers felt the particular goal was achieved in that category. The goal of the software security requirements being implementable is thought to be significantly less achieved in the randomness, and cryptographic algorithms categories. Analysis on the data from security experts shows that this can be explained by the software engineers lack of knowledge in those categories.

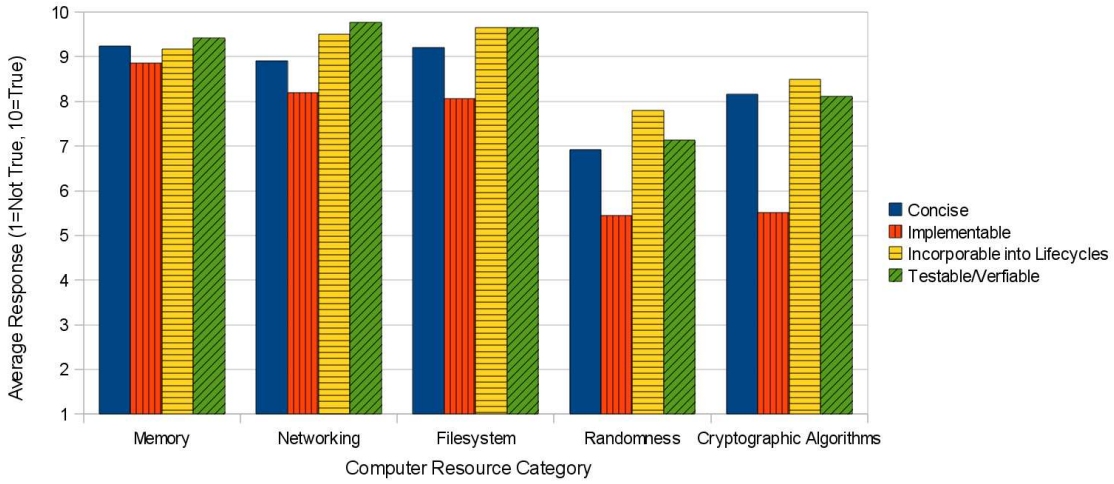


Figure 3.2: Software Engineers Evaluation of the Software Security Requirement Goals

The majority of the software security requirement goals received an average evaluation of 5.5 or higher, meaning the software engineers thought the goals have been mostly achieved. The goals that were not considered achieved by the software engineers, are primarily due to unfamiliarity with those resource categories. This is shown with the one goal that was given to both software engineers and security experts. While software engineers felt that the software security requirements in the randomness, and cryptographic algorithms categories dropped in conciseness, clearness, and non-ambiguity, the security experts did not. The most encouraging data are the results from the testable/verifiable, and incorporable into lifecycle goals, because both remained high throughout all categories of the survey. Overall, the lower responses can be explained by a lack of knowledge in the categories, and the four goals evaluated by software engineers can be considered achieved by this research.

3.4.2 Security Experts Survey

A survey was provided to security experts to substantiate the most critical goals of the software security requirements: whether they prevent the entire set of constraints/assumptions from being violated, and whether they prevent all software vulnerabilities. Security

experts were also asked to evaluate whether each requirement was clear, concise, and non-ambiguous. The process described in 3.2 has a major focus on ensuring that goal is achieved by the software security requirements, otherwise the requirements would be less effective in real software development lifecycles. This was the one question that was also given to software engineers, to determine if knowledge of the material affects the results.

The security experts survey contained statements linked to one of the targeted goals, and the response provided by the software engineer evaluated how well the security expert perceived the goal is achieved. Like the software engineers survey, the responses were limited to a scale from 1 to 10, where a higher score corresponds to a statement that is thought to be more true, and therefore better meeting the intended goal. The survey had security experts respond to: (a) each software security requirement to determine the success of the conciseness goal, (b) each group of software security requirements associated with a constraint/assumption to determine the success of the preventing a constraint/assumption violation goal, and (c) each computer resource category to determine the success of the vulnerability prevention goal. The format, and the statements given to software engineers are shown in appendix A.2. All security experts that participated in this survey are either a security researcher or IT security professional.

3.4.2.1 Results from Security Experts Survey

The survey results from security experts are evaluated in three ways: on a per requirement basis, on a per constraint/assumption basis, and on a per resource category basis. Like the software engineers survey, the responses are averaged by the statement that produced the response, but the security experts survey differs because some of the goals can only be evaluated in a partial set of the methods. For example, the goal to prevent constraint/assumption violations cannot be evaluated on a per requirement basis since the responses were tracked by constraint/assumption. Similarly, the goal to prevent vulnerabilities can only be evaluated per resource category since the responses to that goal were evaluated by resource category. By design, the only way to evaluate all goals simultaneously is through a per resource category method. This leads to a total of 92 quantitative values when evaluated on a per requirement basis, 92 quantitative values when evaluated on a per constraint/assumption basis, and 15 quantitative values when evaluated on a per resource category basis. As with the software engineering survey, any average that is above 5.5 can be considered closer to achieving the corresponding goal, while an average below 5.5 can be considered further from achieving the goal.

Only the conciseness goal can be evaluated when looking at the security experts survey results on a per requirement basis. This single goal was rated very highly by the security experts, with all 92 requirements receiving an average score of 5.5 or higher. This is in contrast to the software engineers, which evaluated some of the requirements much lower on the same goal. All the averages on a per requirement basis are included for reference in appendix B.2.

When the software security requirements are evaluated by the 46 constraints/assumptions, only 3 received an average response lower than 5.5 for violation prevention. Furthermore, 69.5% of the constraints/assumptions had an average response of 7 or higher for the same violation prevention goal. The three constraints/assumptions with low marks are "The process shall not interpret data present on the dynamic memory as executable code", "Data held on the static memory cannot be observed while the process is in execution", and "A file having proprietary or obscure format cannot be understood or modified." The software security requirements written to prevent the violation of the first constraint/assumption are designed to hinder software processes from intentionally executing code in data space. The requirement is then relying on the operating system to monitor memory marked as data to prevent the constraint/assumption from being violated. The security experts are correct to mark the goal lower than the others because of the indirect way the software security requirements handle the issue. Requirements for the second constraint/assumption were marked low because the requirement included the term sensitive which security experts thought to be inadequately defined. In fact, the term caused several of the requirements to be marked lower, but as stated in 3.3, sensitive data is defined in the information security requirements. If a project has no definition for sensitive data, and no security expert is available, then all data must be considered as sensitive. Finally, the third constraint/assumption involving an obscure file format received uncharacteristically low responses as compared to similar constraints/assumptions. Most likely the responses were lower due to a misunderstanding of the constraint/assumption. All the averages on a per constraint/assumption basis are included for reference in appendix B.2.

Goal	Memory	Networking	Filesystem	Randomness	Cryptographic Algorithms
Concise	9.12	9.45	7.94	9.55	9.61
Prevent Violation of Constraint / Assumption	7.58	6.93	7.3	8.9	8.37
Prevent / Vulnerabilities	6.33	5	5.67	8.67	6.67

Table 3.2: Security Experts Evaluation of the Software Security Requirement Goals

Table 3.2 shows the average response by resource category, that the security experts gave on the goals of the software security requirements. The columns of the table correspond to the resource categories, while the rows correspond to three of the software security requirement goals. The values from the table are also shown in a bar graph in figure 3.3 to provide a better visualization for the results. Each bar color represents a goal of the software security requirements, and the bars are grouped by the resource category. The heights of the bar represent how much the security experts felt the particular goal was achieved in that category.

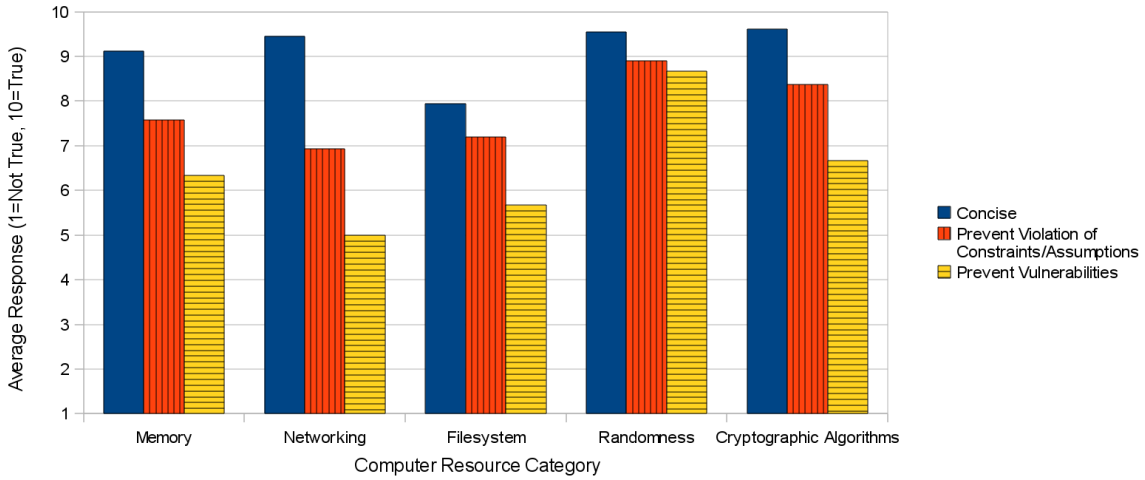


Figure 3.3: Security Experts Evaluation of the Software Security Requirement Goals

The responses for the conciseness goal are consistently higher in the security experts survey than in the software engineers survey. Most notable are the responses in the randomness, and cryptographic algorithms categories, which are the highest of any resource category, unlike in the software engineers survey where they are the lowest. This confirms the idea that the results in those resource categories are skewed in the software engineers survey because of the responders lack of content knowledge. In the security experts survey, the one resource category that received noticeably lower responses is the filesystem category. Further analysis revealed that it could be explained by security experts inexperience with programming terms used frequently throughout the software security requirements in that section. The term "file descriptor" is the one term that immediately sticks out as troublesome, however this explanation still does not answer all the responses in that category. More than likely the lower responses are due to the complexities of the software security requirements in that category, making the result completely valid.

The security experts responses to whether the software security requirements prevent each constraint/assumption from being violated are promising, but not great. The networking resource category received the lowest responses overall, which can be directly attributed to the difficulties at correctly using the network resource. The surprising result is that the software security requirements ranked highest for preventing constraint/assumption violations are in the randomness and cryptographic algorithms resource categories. This data is less surprising when considering that these are the two categories that are universal across all platforms and operating systems. The security experts undoubtedly had some restraint at supplying excellent responses to the other three categories because of the multitude of configurations that could be faced by those software security requirements. In the end, the results are high enough to conclusively say that the security researchers confirmed the software security requirements are capable of preventing the violation of the constraints/assumptions listed in the taxonomy of software vulnerabilities.

The poorest results are from the most far ranging goal of the software security require-

ments, to prevent all software vulnerabilities. The filesystem and networking resource categories are significantly lower, with the network category dropping below a 5.5 average. Again, the difficulties of correctly using the network resource are thought to be the main cause. High marks in randomness, and cryptographic algorithms suggest that the security researchers may have been considering attacks that are information security based. Such attacks are not prevented by the software security requirements, and therefore may have resulted in lower scores. Regardless, the scores reflect apprehension that any set of generic requirements could completely prevent vulnerabilities. This evaluation is correct, the software security requirements can still be used incorrectly, interpreted incorrectly, etc.; the requirements merely provide a new protection against vulnerabilities. That said, the responses are still relatively high, confirming the software security requirements do prevent software vulnerabilities, perhaps just not every conceivable threat to a process.

3.4.3 Conclusions from the Subject Matter Experts Survey

The software security requirements are designed to be: (a) testable and verifiable, (b) clear, concise and non-ambiguous, (c) implementable by software engineers without security knowledge, (d) usable in modern development lifecycles, and (e) capable of preventing modern software vulnerabilities when used correctly in development. Each of those design goals is substantiated by subject matter experts in the survey evaluation, and can be considered achieved by this research. However, there are some concerns from the results that need addressing. The software security requirements received low responses for the implementable goal from software engineers, and vulnerability prevention goal from the security experts.

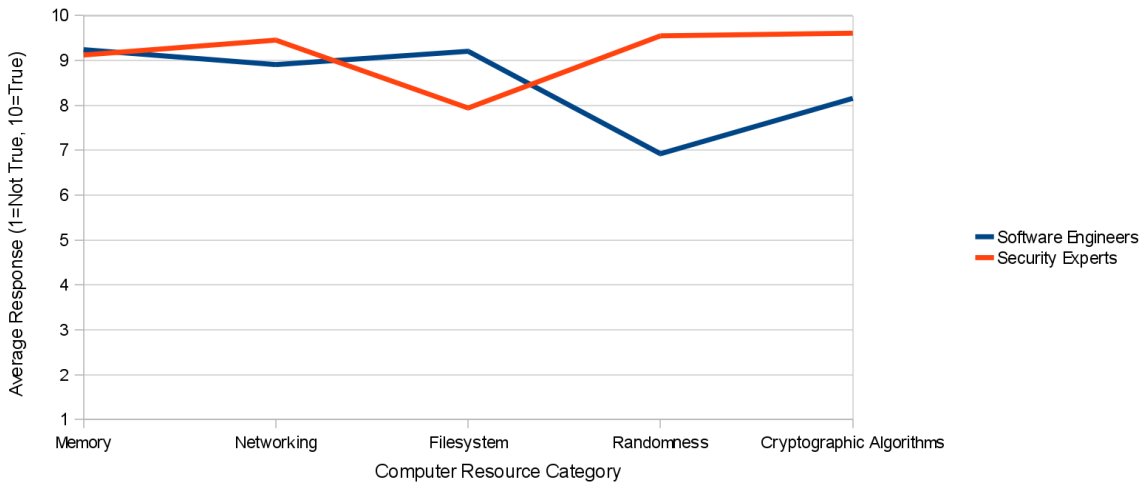


Figure 3.4: Comparison of the Conciseness Goal Between the Two Surveys

Software engineers gave lower responses to several of the targeted goals in the randomness and cryptographic algorithms resource categories. In particular, the goal to make the

software security requirements implementable by software engineers is thought to be less achieved in those categories. Figure 3.2 shows that not only did the ratings for the implementable goal drop in the software engineers survey, but so did the conciseness goal. This is in contrast to the security experts survey, which had the highest responses for conciseness in the randomness and cryptographic resources. Figure 3.4 shows a comparison in the conciseness goal between the two surveys. Since the conciseness evaluations did not drop in the security experts survey, it is reasonable to conjecture that software engineers had a difficult time with the unfamiliar material. Unfortunately, whether the software security requirements in those categories are implementable is now inclusive. Some of the concern is mitigated because the responses for whether the software security requirements are incorporable into lifecycles, and testable/verifiable are still adequate. Therefore, a safe assumption can be made that the software security requirements in those categories are not weaker in terms of their usability.

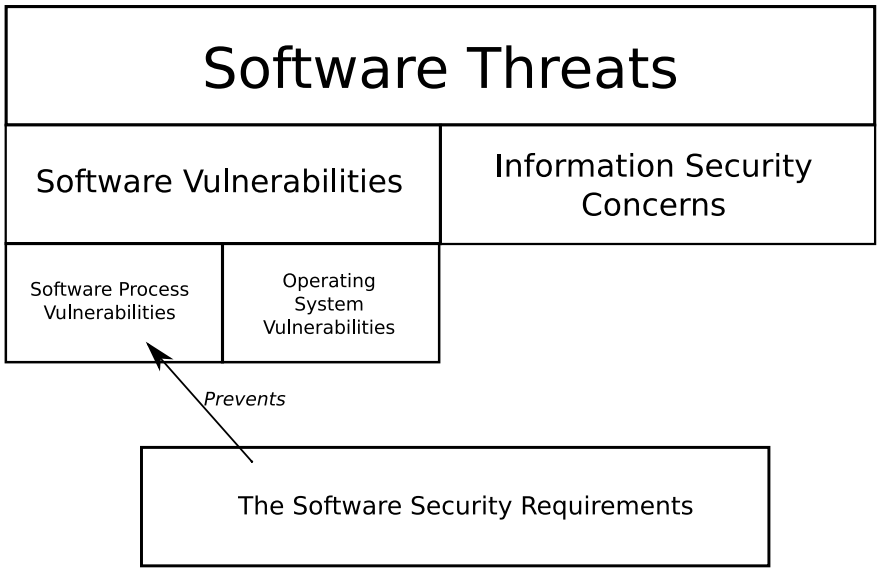


Figure 3.5: The Scope of the Software Security Requirements

The security experts low evaluations for vulnerability prevention can be rationalized several different ways. Security experts are generally cautious at claiming software is vulnerability free, and probably exhibited the same cautious behavior when responding to the survey. This is a fair claim, this research would never suggest that the software security requirements end all vulnerability concerns for software. Results shown in figure 3.3 support this data insight, within each resource category the software security requirements are ranked higher at preventing constraint/assumption violations than preventing the entire set of vulnerabilities. This shows that as the complexities of the goal increased, the security experts became more apprehensive at confirming success. The second rationalization for the low evaluations is an issue of scope; the software security requirements do not prevent against operating system specific vulnerabilities or issues. Figure 3.5 depicts the scope issue with the software security requirements, with the top block representing all software threats.

At each level, the scope is more narrowly defined until the scope of the software security requirements are reached.

The subject matter experts survey ultimately supported the goals of the software security requirements, because even the low marks in the survey are relatively high. The security experts perhaps had the biggest issue with the requirements, but still provided evaluations satisfactory enough to consider success. Ultimately, any future users of the software security requirements can confidently use them in the development process, but should also consider other security focused methods discussed in chapter 2 to ensure the best product.

3.5 Summary

Software requirements targeted at security concerns need to be included in software development lifecycles, but to date no list of such requirements exist. The software security requirements intend to fill that gap, by providing requirements that can be used in any project, completely independent of the platform or language used. The software security requirements are written to prevent the violation of constraints and assumptions listed in the taxonomy of software vulnerabilities, which categorizes vulnerabilities based on their associated computer resource. Security researches validated this main goal, and showed that the requirements do in fact prevent the constraints and assumptions from being violated. While the security experts responded lower to the the requirements ability to prevent all software vulnerabilities, the issue is mainly with scope; the security experts are considering a much broader range of issues that the software security requirements do not cover.

The software security requirements also differ from other attempts at security requirements because their use does not require security experts for the process. Currently, all other methods of generating security requirements leverage a technique called threat modeling, which must be done by security experts. The software security requirements have no need for a security expert during the development process, because the software security requirements provide detailed requirements for all modern software vulnerabilities. Security experts validated this claim through their review of the requirements, showing that the software security requirements prevent the vulnerabilities represented by the taxonomy of software vulnerabilities.

Chapter 4

Using Software Security Requirements During Development

Once requirements engineering has concluded, a translation process must occur to correctly map requirements into the design and implementation. A project that fails to translate requirements correctly has no guarantee of correct output. The assurance of requirements translation is handled through verification and validation (V&V) strategies, which occur during the design, implementation, and testing phases.

Verifying and validating security of a project has differed from standard strategies because typically security has received different access to development. Projects that start security concerns during testing, a common practice until recently, are providing the security review process less access to development than projects that start security concerns during requirements. Therefore, the verification and validation strategies for security must differ depending on the access to development.

4.1 Access-Driven VV&T

Access-Driven VV&T defines verification, validation, and testing strategies for software security based on the access to development, and the availability of artifacts (documentation) [5]. *Verification* ensures that software correctly implements a specific function [65]. It occurs at the end of each phase of development, to verify that requirements imposed from the previous phase have been correctly fulfilled. The next phase of development cannot begin until all requirements have been verified. *Validation* ensures that the built software can be traced to requirements [65]. The focus is on testing functional requirements, to ensure the correct product was built. Lastly, *testing* attempts to find desirable (undesirable) characteristics in the software [5]. This differs from validation testing because no functional requirements are validated, and the testing procedures are primarily of the risk based type described in 2.2.1.2. The three types of VV&T are as follows:

Full VV&T strategies are used when security considerations start during requirements engineering, and full-access to development artifacts (documentation) is provided.

Partial VV&T strategies are used when security considerations start at the end of development, but the access to all development artifacts is available.

End-Game VV&T strategies have no access to the development artifacts, and consequently will start at the end of development. Typically the strategies in end-game are penetration testing style techniques.

The strategies employed in VV&T differ on the type because limited access to development alters the capabilities of some of the strategies. For example, verification can only occur in full VV&T and partial VV&T since end-game does not provide the documentation necessary for verification. The most desirable VV&T type is full VV&T, followed by partial VV&T because in each case the increased accessibility to development allows for strategies to occur earlier, rather than later in development.

4.1.1 Full VV&T with the Software Security Requirements

Security concerns are best integrated into the development process when full VV&T is employed because verification, validation, and testing can all occur. This research does not consider partial VV&T and end-game VV&T an option for secure software development because security concerns start too late. The focus is on full VV&T, which contains five strategies: concept definition, requirements engineering, test plan and test description, unit testing, validation, and verification [5]. The following is a description of how each full VV&T strategy is utilized in this approach:

Concept definition focuses on the high-level objectives of a system [5]. The customer needs are taken into consideration, which ultimately affects the security needs. For example, systems that store passwords will have more security concerns. The concerns will always relate to the system in development, and the intended operating environment.

Requirements engineering provides a set of requirements for security that are testable or verifiable. At a minimum, the software security requirements, presented in 3.3, are used within this strategy but additional security requirements can be added by knowledgeable security experts. In particular, information security may need considerations because the provided requirements only cover software security.

Test plan and test description documents specify the tests that will be executed to validate every software security requirement. The documents contain the test cases, test scenarios, procedures, and test descriptions [5]. Each test will have specified input, and expected output. At a higher level, the documents will contain system configurations, and the sequence of testing. The documents are generated during the design phase, before any testing begins.

Unit testing activities are the same as described in 2.2.2.1. Each individual software module is tested by the implementer of the module. The software security requirements are used as a guide for testing to ensure correct implementation of each individual unit.

Validation is conducted to ensure that the finished product meets requirements. The list of software security requirements must be validated, otherwise a vulnerable system could result. Validation follows the test plan and test description derived earlier to ensure all software security requirements have been properly tested and validated.

Verification checks for successful translation of requirements into the design and implementation phases by reviewing artifacts. The process ensures the product is being built correctly before moving on to the next phase. A list of expected security artifacts for the software security requirements is provided in 4.2.2. The security artifacts must be verified by comparing them against the actual results of the design and implementation.

The strategies in full VV&T make it ideal for security because they cover the entire lifecycle. All of the strategies except for concept definition use the software security requirements in some way, making them integral to this security approach. Unfortunately, the responses from software engineers have shown that there may be an issue with understanding and implementing some of the software security requirements. This presents a problem for the verification strategy in particular, because it requires knowledge of how to properly translate the requirements into the design and implementation. However, a generic set of expected artifacts for the design and implementation, such as the security artifacts presented below, can be compared against the design and implementation documentation for an effective verification process.

4.2 The Security Artifacts

Each of the supplied software security requirements are testable and/or verifiable, and ready for use in full VV&T. However, verification in full VV&T may be more difficult, especially for developers that lack security knowledge. Without verification, correct usage of the software security requirements cannot be assured until validation, during the testing phase.

Support for verifying the software security requirements in full VV&T is provided through a set of expected security artifacts, listed in 4.2.2. A developer will compare the security artifacts for each software security requirement against the actual materials from the design and implementation phases. When the security artifacts match the actual materials, the software security requirements are verified, and have achieved proper translation into the design and implementation. The process for using the security artifacts in verification is described in 4.2.3.

The security artifacts are broken into two types:

Design artifacts are found in the design phase of the software development lifecycle. The Security artifacts found in design include the choice of encryption ciphers and cipher

modes, file formats, network data formats, etc., that provide details for interoperability between the different components in the implementation phase.

Code artifacts are found in the implementation phase of the software development lifecycle. All security artifacts of this type are compared against the code of the project for verification.

The security artifacts must be created from the software security requirements methodically, so that all of the stipulations in the requirement are correctly mapped to the artifacts. Also, like the software security requirements, the security artifacts must be generic so that their use supports any platform or language. This places an importance on the security artifact generation process which must take these factors into consideration, along with the type of security artifact being generated.

4.2.1 Security Artifacts Generation Process

The security artifacts are generated directly from the software security requirements, but the process differs for each type of artifact. Therefore, each software security requirement is given to two processes, so that security artifacts for the design and code will be generated. The process for generating design artifacts does not always yield a security artifact, but the process for generating code artifacts will always result in at least one security artifact. This forces each software security requirement to have a minimum of one security artifact. The two processes for generating security artifacts are:

- ***Process for generating security artifacts found in design:***

  (1) **Examine the software security requirement to determine if anything requires a decision that will potentially affect multiple components in the implementation phase.** The implementation of a project is typically broken into components to be developed by separate people or groups, which are then later integrated. When this occurs, design decisions become critical so that each component can be easily integrated. If the inclusion of a software security requirement will affect multiple components, then a decision will be required in the design phase. The decision represents a design security artifact, and is often seen in requirements covering formatting or value concerns, encryption concerns, or authentication concerns. If no decision is required during the design phase, the process immediately stops, and no design security artifact will be associated with the software security requirement.

  (2) **Decompose the software security requirement into individual design segments.** While the first step determined whether a design artifact would be present, the second step must decompose the requirements into segments that will each become a design security artifact. For example, encryption and authentication provided separate functionality, so they must be separated to represent their own design artifact.

(3) **Take each segment, and state the decision that must be made.** The step starts with a segment determined in the previous step, and ends with an unconfirmed design security artifact. The segment is re-stated in a manner that describes what decision must be made, and an example is included if the decision is for selection of an algorithm/process. Taking the authentication segment from the above example into this step would yield, "A method that provides data authenticity is selected", which describes exactly the decision that must be made during the design. This step can be made more helpful by supplying an example of the decision result.

(4) **Compare each design artifact against the original software security requirement.** An assurance must be made that each newly created design artifact matches the intent of the original software security requirement. If the software security requirement includes a provision that a NIST algorithm will be selected, the design artifact must stipulate the same.

- ***Process for generating security artifacts found in code:***

(1) **Write pseudocode that will implement the software security requirement correctly.** The pseudocode must cover all security concerns from the original requirement. In most cases it is helpful to write in C/C++ style coding, because the low-level nature of the language is conducive to mistakes that are not possible by design in other languages. Specifically, software security requirements covering memory management are written in a form of C/C++ pseudocode. Lastly, when producing the psuedocode, no extras that are out of the scope of the requirement are included. Requirements covering encryption ciphers are not taken to the implementation of an encryption cipher, instead it is adequate to assume a library is available.

(2) **Convert pseudocode to English language.** Pseudocode is too verbose and lengthy, so it is converted to explicit sentence form English language, that is non-ambiguous, specific on the ordering of events, and specific on library and subroutine calls.

(3) **Remove any computer language specific material.** The implementation security artifact cannot contain any material that only references a specific subroutine/function in a language. Instead, a clear descriptive phrase as to what the subroutine does is provided. If devising an artifact for sending network data, the C socket function send cannot be referenced directly in the artifact, and is changed to "a subroutine is called to send data to a remote host". However, the send function is included as an example of such a subroutine.

(4) **Compare the implementation artifact against the original software security requirement.** The implementation artifact must correctly address all concerns in the original requirement. Also, the artifact is reviewed to ensure the language is non-ambiguous, otherwise implementations that would match the artifact but still violate the requirement could be created.

The one step that is equivalent in both processes is the final step in each, comparing against the original software security requirement. The step is mandatory so that the resulting security artifact translates all of the stipulations in a software security requirement. The effectiveness of verification would be limited without this guaranteed translation, so the security artifacts presented in the next section have been carefully compared to the original software security requirement to ensure proper translation.

4.2.2 Results of the Security Artifacts Generation Process

Each software security requirement is given to the two processes provided in the previous section, which generate the security artifacts presented in this section. The security artifacts are grouped with the connected software security requirement, and the requirements are organized by the associated computer system resource. If a requirement is used to prevent multiple constraints/assumptions, it will only be listed once for each computer system resource. This creates a situation where the same software security requirement can be listed in different computer system resources, a choice made for clarity when reading through the security artifacts.

4.2.2.1 Dynamic Memory

This section presents the security artifacts from software security requirements imposed on the dynamic memory resource.

(1) ***Software Security Requirement:*** When copying data into a buffer, the process shall ensure that the data being copied does not exceed the bounds of the buffer.

Security artifacts present in code:

- Before using subroutines (sprintf, etc.) that require an input and a buffer for copying the input into, the size of the buffer is compared against the size of the input. If the size of the input exceeds the size of the buffer, the buffer is allocated more space to equal the size of the input or the input is truncated.

(2) ***Software Security Requirement:*** If data being copied into a buffer is expected to have a termination character to determine end of the data, then the process shall ensure that the termination character is present in the data, and ensure that the data being copied does not exceed the bounds of the buffer.

Security artifacts present in code:

- Before using subroutines (sprintf, etc.) that require an input to contain a termination character, the input being used by the subroutine is checked one character at a time for the termination character. After every character in

the input is checked, the termination character is appended to the end of the input if the termination character is not found. In this way the subroutine is guaranteed an input with the expected termination character. Then, the size of the buffer is compared against the size of the input. If the size of the input exceeds the size of the buffer, the buffer is allocated more space to equal the size of the input or the input is truncated.

(3) ***Software Security Requirement:*** The process must not contain code that either directly or indirectly causes the instruction pointer to load with an address outside of instruction space.

Security artifacts present in design:

- The design does not include an artifact (architectural or algorithmic) that rewrites the instruction pointer to load with a new address outside of code space.

Security artifacts present in code:

- When an operation is being specified that causes an address to load a new instruction pointer value, then that address must be an address in code space.

(4) ***Software Security Requirement:*** The process shall ensure the value of an environment variable is in expected format before use.

Security artifacts present in design:

- Environment variables required by the process are determined.
- Expected format of required environment variables are determined.

Security artifacts present in code:

- Environment variables retrieved by subroutines (getenv, etc.) are compared against the expected format determined during the design phase. The environment variable is only used if it matches the expected format.

(5) ***Software Security Requirement:*** If an environment variable used by the process is expected to have a set of values, the process shall ensure the value of the environment variable is one of those expected values before use.

Security artifacts present in design:

- Environment variables required by the process are determined.
- Any expected values of required environment variables are determined.

Security artifacts present in code:

- Environment variables retrieved by subroutines (getenv, etc) that have a set of expected values are compared against any expected values determined during the design phase. The environment variable is only used it matches one of the expected values.

(6) ***Software Security Requirement:*** Before invoking another process for execution, the invoking process shall clear all environment variables, and set environment variables required for execution with trusted values for the process being invoked.

Security artifacts present in design:

- If the process needs to invoke another process, the environment variables required for the process invocation are determined.
- If the process needs to invoke another process, trusted values for environment variables required for process invocation are determined.

Security artifacts present in code:

- Before using a subroutine to invoke another process (execv, etc.), the process creates a trusted environment for the invoked process by clearing all environment variables (clearenv, etc.), and setting environment variables (setenv, etc.) required for execution to the trusted values determined during the design phase.

(7) ***Software Security Requirement:*** After a process makes a request for memory, the process shall check to see that the memory was properly allocated, and only use memory that has been successfully allocated.

Security artifacts present in design:

- A plan for handling failed memory attempts is determined to prevent the use of memory that was never allocated.

Security artifacts present in code:

- After using a subroutine (malloc, realloc, etc) that will request a memory allocation from the operating system, a check is made for a failed allocation. If the allocation fails, the value returned by the operating system is ignored, and the plan determined the design phase is used.

(8) ***Software Security Requirement:*** The process shall store sensitive data in memory only when necessary, and use a subroutine provided by the operating system to store the sensitive data in an encrypted format when the sensitive data is not in active use.

Security artifacts present in design:

- Sensitive data that needs to be held in memory is determined.

Security artifacts present in code:

- When sensitive data held in memory is not in use but still needed, the memory is given to an operating system subroutine (CryptProtectData, etc.) that encrypts the entire block of memory.

(9) ***Software Security Requirement:*** The process shall not reallocate memory containing sensitive data.

Security artifacts present in design:

- Sensitive data that needs to be held in memory is determined.

Security artifacts present in code:

- Memory containing sensitive data is never given to a subroutine that reallocates memory (realloc, etc).

(10) ***Software Security Requirement:*** If a process holds sensitive data in memory, the process shall write zeros to the entire block of memory before releasing the memory to the operating system. The process shall not use a built in subroutine for writing the zeros.

Security artifacts present in design:

- Sensitive data that needs to be held in memory is determined.

Security artifacts present in code:

- Before using a subroutine (free, etc.) to release sensitive data from memory, the entire block of memory is overwritten with zero values by iterating over each byte in the block of memory.

(11) ***Software Security Requirement:*** After a pointer to a memory location on the heap is deallocated, the process shall set that pointer's value to NULL.

Security artifacts present in code:

- After a call to a subroutine (free, etc) that returns memory to the operating system, the pointer associated with that call is immediately set to NULL.

(12) ***Software Security Requirement:*** Before dereferencing a pointer, the process shall ensure that the pointer's value is not set to NULL.

Security artifacts present in code:

- Before dereferencing a pointer (*i, i->foo(), etc.), a check is made to see if the pointer's value is to set to NULL. The pointer is not dereferenced if the value is NULL.

(13) ***Software Security Requirement:*** The process shall only set a pointer's value to an address located on the stack, if the lifetime of the stack variable will end after the lifetime of the pointer.

Security artifacts present in code:

- A pointer assignment operation (int i* = &x) is made to a memory location on the stack when the stack's lifetime will not end before the pointer's lifetime.

(14) ***Software Security Requirement:*** The process shall never make a request to the operating system for zero bytes of memory.

Security artifacts present in code:

- Before using a subroutine (malloc, etc.) that requests memory from the operating system, a check is made for the amount of memory being requested. The request is only made if the request is for more than zero bytes of memory.

(15) ***Software Security Requirement:*** The process shall ensure that all addresses placed in a pointer variable are to memory locations other than its own.

Security artifacts present in code:

- After a pointer is assigned a memory address, the address is compared against the address of the pointer. If the memory addresses are the same, then the pointer is assigned a NULL value.

(16) ***Software Security Requirement:*** The process shall ensure the value of string input is in expected format before use.

Security artifacts present in design:

- String inputs required by the process are determined.
- Expected format of required string inputs are determined.

Security artifacts present in code:

- Before a string input is used, a check is made against the expected format determined during the design phase. The string input is only used if it matches the expected format.

(17) ***Software Security Requirement:*** If an input has an expected set of values, the process shall ensure the value of the input is one of those expected values before use.

Security artifacts present in design:

- String inputs required by the process are determined.
- Any expected values of required string inputs are determined.

Security artifacts present in code:

- Before a string input is used, a check is made against any expected values determined during the design phase. The string input is only used it matches one of the expected values.

(18) ***Software Security Requirement:*** Before an addition operation where both operands are positive, the process shall ensure that subtracting the left operand from the largest possible value is greater than or equal to the right operand. When the previous condition is not met, the process shall not perform the addition operation, and block any corresponding input from further use.

Security artifacts present in code:

- Before adding two numbers, the two operands are checked. If the two operands are positive, the left operand is subtracted from the largest possible value, and that result is compared to the right operand. If the right operand is greater, the addition operation does not occur, and any corresponding input is rejected from use.

(19) ***Software Security Requirement:*** Before an addition operation where both operands are negative, the process shall ensure that subtracting the right operand from the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the addition operation, and block any corresponding input from further use.

Security artifacts present in code:

- Before adding two numbers, the two operands are checked. If the two operands are negative, the right operand is subtracted from the smallest possible value, and that result is compared to the left operand. If the left operand is less, the addition operation does not occur, and any corresponding input is rejected from use.

(20) ***Software Security Requirement:*** Before a subtraction operation with unsigned numbers, the process shall ensure that the left operand is greater than the right operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.

Security artifacts present in code:

- Before subtracting two unsigned numbers, the left operand is compared to the right operand. If the right operand is greater, the subtraction operation does not occur, and any corresponding input is rejected from use.

(21) ***Software Security Requirement:*** Before a subtraction operation where the left operand is nonnegative and the right operand is negative, the process shall ensure that adding the right operand to the largest possible value is greater than or equal to the left operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.

Security artifacts present in code:

- Before subtracting two numbers, the two operands are checked. If the left operand is nonnegative and the right operand is negative, the right operand is added to the largest possible value, and that result is compared to the left operand. If the left operand is greater, the subtraction operation does not occur, and any corresponding input is rejected from use.

(22) ***Software Security Requirement:*** Before a subtraction operation where the left operand is negative and the right operand is positive, the process shall ensure that adding the right operand to the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.

Security artifacts present in code:

- Before subtracting two numbers, the two operands are checked. If the left operand is negative, and the right operand is positive, the right operand is added to the smallest possible value, and that result is compared to the left operand. If the left operand is greater, the subtraction operation does not occur, and any corresponding input is rejected from use.

(23) ***Software Security Requirement:*** Before a multiplication operation where both operands have the same sign, the process shall ensure that dividing the right operand by the largest possible value is greater than or equal to the left operand. When the previous condition is not met, the process shall not perform the multiplication operation, and block any corresponding input from further use.

Security artifacts present in code:

- Before multiplying two numbers, the signs of the two operands are checked. If the two operands have the same sign, the right operand is divided by the largest possible value, and that result is compared to the left operand. If the left operand is greater, the multiplication operation does not occur, and any corresponding input is rejected from further use.

(24) ***Software Security Requirement:*** Before a multiplication operation where operands have different signs, the process shall ensure that dividing the right operand by the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the multiplication operation, and block any corresponding input from further use.

Security artifacts present in code:

- Before multiplying two numbers, the signs of the two operands are checked. If the two operands have different signs, the right operand is divided by the smallest possible value, and that result is compared to the left operand. If the left operand is greater, the multiplication operation does not occur, and any corresponding input is rejected from further use.

(25) ***Software Security Requirement:*** The process shall ensure that a division operation never contains the largest negative value in the numerator, with a -1 in the denominator. When the previous condition is not met, the process shall not perform the division operation, and block any corresponding input from use.

Security artifacts present in code:

- Before dividing two numbers, the numerator and denominator values are checked. If the numerator is the largest negative value, and the denominator is -1, the division does not occur and any corresponding input is rejected from use.

(26) ***Software Security Requirement:*** Before assigning a new value to a variable used as an index to a buffer, the process shall ensure the new value is within the buffers bounds.

Security artifacts present in code:

- Before an assignment operation (ie. x = i) to a variable used as an index to a buffer, the value is compared against the index range of the buffer. The assignment operation is only completed if the new value is within that range.

(27) ***Software Security Requirement:*** Before assigning a new value to a variable used to hold the length or quantity of an object, the process shall ensure that the new value is nonnegative.

Security artifacts present in code:

- Before an assignment operation (i.e. x = i) to a variable used to hold a length or quantity of an object, the new value is determined. The assignment operation on the variable is only completed if the new value is nonnegative.

4.2.2.2 Static Memory

This section presents the security artifacts from software security requirements imposed on the static memory resource.

(1) ***Software Security Requirement:*** When copying data into a buffer, the process shall ensure that the data being copied does not exceed the bounds of the buffer.

Security artifacts present in code:

- Before using subroutines (sprintf, etc.) that require an input and a buffer for copying the input into, the size of the buffer is compared against the size of the input. If the size of the input exceeds the size of the buffer, the buffer is allocated more space to equal the size of the input or the input is truncated.

(2) ***Software Security Requirement:*** If data being copied into a buffer is expected to have a termination character to determine end of the data, then the process shall ensure that the termination character is present in the data, and ensure that the data being copied does not exceed the bounds of the buffer.

Security artifacts present in code:

- Before using subroutines (sprintf, etc.) that require an input to contain a termination character, the input being used by the subroutine is checked one character at a time for the termination character. After every character in the input is checked, the termination character is appended to the end of the input if the termination character is not found. In this way the subroutine is guaranteed an input with the expected termination character. Then, the size of the buffer is compared against the size of the input. If the size of the input exceeds the size of the buffer, the buffer is allocated more space to equal the size of the input or the input is truncated.

(3) ***Software Security Requirement:*** The process shall not store sensitive data in static memory.

Security artifacts present in design:

- Sensitive data that needs to be held in memory is determined.

Security artifacts present in code:

- A global or static variable is never used to store sensitive data.

4.2.2.3 Network Interface

This section presents the security artifacts from software security requirements imposed on the network interface resource.

(1) ***Software Security Requirement:*** When receiving encrypted data over a network, the process shall decrypt and authenticate all data before its use, and reject data that does not authenticate or decrypt properly.

Security artifacts present in design:

- Sensitive data that needs to be received in an encrypted format from the remote host is determined.
- A method that provides data authenticity verification (CCM cipher block mode, etc.) from the sending host is selected.
- A method that provides data encryption/decryption (AES encryption, etc.) from the sending host is selected.

Security artifacts present in code:

- Sensitive data received from a remote host and retrieved by a subroutine (recv, etc.) is first verified for authenticity and decrypted using the method(s) selected during the design phase. The data is only used if it properly authenticates and decrypts, otherwise it is immediately discarded.

(2) ***Software Security Requirement:*** When receiving unencrypted data over a network from another host, the process shall authenticate all data before its use, and reject data that does not authenticate properly.

Security artifacts present in design:

- A method that provides data authenticity verification (HMAC-SHA256, etc.) from the sending host is selected.

Security artifacts present in code:

- Data received from a remote host and retrieved by a subroutine (recv, etc) is first given to the method that provides data authenticity verification selected during the design phase. The data is only used if it properly authenticates, otherwise it is immediately discarded.

(3) ***Software Security Requirement:*** The process shall authenticate the remote host to verify its identity before sending or accepting any data from that host.

Security artifacts present in design:

- An authentication scheme (X.509 standard, etc.) between two hosts is selected to uniquely identify a machine, and verify its identity.

Security artifacts present in code:

- Before using a subroutine (send, etc.) to start sending data to a remote host, the authentication scheme selected during the design phase is used to verify the identity of the remote host.
- Before using a subroutine (recv, etc.) to start receiving data from a remote host, the authentication scheme selected during the design phase is used to verify the identity of the remote host.

(4) ***Software Security Requirement:*** The process shall ensure data received from a remote host is in expected format before use.

Security artifacts present in design:

- Format of messages being sent between hosts is determined.

Security artifacts present in code:

- After using a subroutine (recv, etc.) to retrieve new data from a remote host, the format of the received data is checked. The received data is only used if its format matches the format determined during the design phase.

(5) ***Software Security Requirement:*** The process shall only send data in the expected format.

Security artifacts present in design:

- Format of messages being sent between hosts is determined.

Security artifacts present in code:

- Before using a subroutine (send, etc.) to send new data to a remote host, the format of the new data is checked. The data is only sent if its format matches the format determined during the design phase.

(6) ***Software Security Requirement:*** If a process is sending sensitive data to another host on a network, the process shall encrypt and provide authentication for the data.

Security artifacts present in design:

- Sensitive data that needs to be sent to the remote host is determined.
- A method that provides data authenticity verification (HMAC-SHA256, etc.) to the receiving host is selected.

- A method that provides data encryption/decryption (AES encryption, etc.) for sensitive data being sent to the remote host is selected.

Security artifacts present in code:

- Before sensitive data is given to a subroutine (send, etc.) to send to a remote host, the sensitive data is given to the method for encryption and authentication selected during the design phase. Then the encrypted data, along with the information needed to verify the authenticity of the message, is given to a subroutine (send, etc.) to send to the remote host.

(7) ***Software Security Requirement:*** When sending data over a network to another host, the process shall provide authentication for the data.

Security artifacts present in design:

- A method that provides data authenticity verification (HMAC-SHA256, etc.) to the receiving host is selected.

Security artifacts present in code:

- Before giving data to a subroutine (send, etc.) to send to a remote host, the data being sent is first given to the data authentication verification method selected during the design phase. Then the data, along with the information needed to verify the authenticity of the message, is given to a subroutine (send, etc.) to send to the remote host.

(8) ***Software Security Requirement:*** The process shall check to see if the desired port is available for listening before waiting for connections on that port.

Security artifacts present in code:

- After using a subroutine (listen, etc.) that attempts to listen for incoming connections on a port, a check is done to verify if the port is available for listening. A subroutine (accept, etc.) that waits and then accepts incoming connections is only used if the port is available for listening.

(9) ***Software Security Requirement:*** The process shall check to see if a connection to a remote host was accepted before sending data to that host.

Security artifacts present in code:

- After using a subroutine (connect, etc.) to attempt a connection with a remote host, a check is done to verify that the connection was established before any subroutine (send, etc.) is used to send data on that connection.

(10) ***Software Security Requirement:*** The process shall send all numerical data over a network in network byte order.

Security artifacts present in code:

- Before using a subroutine (send, etc.) to send numerical data over a network to a remote host, a subroutine (htonl, etc.) is used to convert the numerical data to network byte order.

(11) ***Software Security Requirement:*** The process shall expect all numerical data received over a network to be in network byte order.

Security artifacts present in code:

- Before using numerical data received from a remote host and retrieved by a subroutine (recv, etc), a subroutine (ntohl, etc.) is used to convert the numerical data to host machine order.

4.2.2.4 Filesystem

This section presents the security artifacts from software security requirements imposed on the filesystem resource.

(1) ***Software Security Requirement:*** When creating a new file/directory, the process shall set the access permissions so the fewest number of users possible have access. The process shall set the access permissions using the file descriptor and not the filename.

Security artifacts present in design:

- The users that need access to a file/directory are determined, and corresponding user and group ownership required to give these users access is determined.
- The access permissions that give the fewest users access to the file/directory while still providing access to the required users is determined.

Security artifacts present in code:

- After using a subroutine (open, etc.) to create a new file/directory, a subroutine (fchown, etc.) uses the file descriptor returned in its creation to set user and group ownership of the file/directory, setting ownership to what was determined during the design phase. Finally a subroutine (fchmod, etc.) uses the same file descriptor to set access permissions of the file/directory, setting access permissions to what was determined during the design phase.

(2) ***Software Security Requirement:*** Before using a file/directory, the process shall check the ownership and the access permissions of the file/directory, and only use files/directories with expected ownership and access permissions.

Security artifacts present in design:

- Users that need access to a file/directory are determined, and corresponding user and group ownership required to give these users access is determined.
- The permissions that give the fewest users access to the file/directory while still providing access to the required users is determined.

Security artifacts present in code:

- After using a subroutine (open, etc.) to open a file/directory, a subroutine (fstat, etc.) uses the file descriptor returned when opening the file/directory to check ownership and access permissions. If user ownership, group ownership, or access permissions fails to match what was determined during the design stage, the file is not used.

(3) ***Software Security Requirement:*** When creating a file, the process shall ensure the path and name are unique.

Security artifacts present in code:

- When a subroutine (open, etc.) is used to open a file, the subroutine is instructed to return an error (using O_CREAT | O_EXCL flags, or equivalent) if the name of the file already exists.

(4) ***Software Security Requirement:*** If the process does not need to open files/directories through links, the process shall use a file open subroutine that blocks the opening of files/directories through links.

Security artifacts present in design:

- Whether files/directories will need to open through links is determined.

Security artifacts present in code:

- When using a subroutine (open, etc.) to open a file/directory that cannot be opened through links, an operating system flag (O_NOFOLLOW, etc.) is set in the subroutine.

(5) ***Software Security Requirement:*** If the process needs to open files/directories through links, the process shall first check the access permissions of the link itself. Then, the process shall open the file, and use the file descriptor to check the access permissions of the file. If the access permissions of the file and link do not match, the process shall not use the file.

Security artifacts present in design:

- Whether files/directories will need to open through links is determined.

Security artifacts present in code:

- When opening a file/directory that can opened through links, a call is made to a subroutine (lstat, etc.) that retrieves the ownership and access permissions of the filename given, but does not follow links. A subroutine (open, etc.) that opens files/directories is then used, and the file descriptor returned by the open subroutine is given to a subroutine (fstat, etc.) that retrieves the ownership and access permissions of the opened file. The process only uses the file if user ownership and access permissions from both subroutine retrievals match.

(6) ***Software Security Requirement:*** The process shall ensure that data contained in a file is in expected format before use.

Security artifacts present in design:

- The format of a file being used by the process is determined.

Security artifacts present in code:

- After using a subroutine (open, etc.) to open a file, the format of the file is checked. The file is only used if its format matches the format determined in the design phase.

(7) ***Software Security Requirement:*** If a file used by the process is expected to have a set of data, the process shall ensure that the expected data is in that file before use.

Security artifacts present in design:

- Any data expected to be in a file used by the process is determined.

Security artifacts present in code:

- After using a subroutine (open, etc.) to open a file, the data in the file is checked. The file is only used if its data matches the expected data determined during the design phase.

(8) ***Software Security Requirement:*** After opening a file, but before reading or writing to that file, the process shall request the operating system lock the file, using the file descriptor returned by the file open operation.

Security artifacts present in code:

- After using a subroutine (open, etc.) to open a file, but before using a subroutine (fread, fprintf, etc.) that reads or writes to that file, a subroutine (LockFileEx, etc.) is used to lock the file using the file descriptor returned by the open subroutine.

(9) ***Software Security Requirement:*** Before closing a file, but after all reading and writing operations have been completed on that file, the process shall request the operating system unlock the file, using the file descriptor given to the previous lock file operation.

Security artifacts present in code:

- Before using a subroutine (close, etc.) to close a file, but after all calls to subroutines (fread, fprintf, etc) for reading and writing to the file, a subroutine (unlock, etc.) is used to unlock the file with the same file descriptor that locked the file.

(10) ***Software Security Requirement:*** If writing sensitive data to a file, the process shall encrypt and provide authentication for the data.

Security artifacts present in design:

- A method that provides data encryption/decryption (AES encryption, etc.) is selected.
- A method that provides data authentication (HMAC-SHA256, etc.) is selected.

Security artifacts present in code:

- Before writing sensitive data to the filesystem, the sensitive data is given to the method for encryption and authentication selected during the design phase. Then the encrypted data, along with the information needed to verify the authenticity of the message, is written to the filesystem.

(11) ***Software Security Requirement:*** The process shall provide authentication for unencrypted data in files.

Security artifacts present in design:

- A method that provides data authentication (HMAC-SHA256, etc.) is selected.

Security artifacts present in code:

- Before using a subroutine (unlock, etc.) to unlock a file, but after all calls to subroutines (fread, fprintf, etc) for reading and writing to the file, the method for data authentication selected during the design phase is used with the contents of the file. The resulting verification value is then stored for later use with the file.

(12) ***Software Security Requirement:*** The process shall only use data from files that have verified authenticity.

Security artifacts present in design:

- A method that provides data authentication (HMAC-SHA256, etc.) is selected.

Security artifacts present in code:

- After using a subroutine (LockFileEx, etc.) to lock an opened file, but before using a subroutine (fread, fprintf, etc.) that reads or writes to a file, the method for data authentication selected during the design phase is used to verify the authenticity of the file. The file is only used if the authenticity of the file is confirmed.

(13) ***Software Security Requirement:*** After deleting a file, if the data is too sensitive to be left on disk even in an encrypted format, the process shall write zeros to the entire hard-drive partition that contained the file. The process shall take into consideration that the procedure could also remove the operating system from disk.

Security artifacts present in design:

- Data that may potentially be too sensitive to remain on disk, even when encrypted, is determined.
- Whether deleting the operating system is an acceptable risk when deleting data that is too sensitive to remain on disk is determined.

Security artifacts present in code:

- Before deleting a file that has been determined during the design phase to be too sensitive to remain on disk even when encrypted, a subroutine (fstat) is called using the file descriptor returned during the file open operation to retrieve the device id information on the file. The device id information will then be used to determine the associated partition, and used in a subroutine (dd, etc.) that will write zeros to that entire partition.

(14) ***Software Security Requirement:*** After attempting a write to a file, the process shall ensure that the write was successful.

Security artifacts present in design:

- A method for handling failed write attempts to the filesystem is determined.

Security artifacts present in code:

- After using a subroutine (fprintf, etc.) to a write to a file, a check is done to determine if the write was successful. If the write was unsuccessful, the strategy for handling failed write attempts determined during the design phase is used.

4.2.2.5 Randomness Resources

This section presents the security artifacts from software security requirements imposed on randomness resources.

(1) ***Software Security Requirement:*** The process shall use a random number generation algorithm that is currently recommended by NIST.

Security artifacts present in design:

- A NIST recommended cryptographically secure PRNG (Hash_DRBG, etc.) is selected.

Security artifacts present in code:

- When random data is required, a call is made to a subroutine (RAND_bytes, etc.) that produces random data based on the cryptographically secure PRNG algorithm selected during the design phase.

(2) ***Software Security Requirement:*** The process shall seed the cryptographically secure PRNG before any values are generated.

Security artifacts present in code:

- Before the first call to a subroutine (RAND_bytes, etc.) that retrieves random data, a subroutine (RAND_add, etc.) is called that will seed the cryptographically secure PRNG algorithm selected during the design phase.

(3) ***Software Security Requirement:*** The process shall reseed the cryptographically secure PRNG when the NIST defined maximum number of generated values per seed for that algorithm has been reached.

Security artifacts present in design:

- The maximum number of values that can be generated per seed for the cryptographically secure PRNG algorithm in use is determined from NIST documentation.
- A process for storing the number of values generated from the current seed is determined.

Security artifacts present in code:

- Before using a subroutine (RAND_bytes, etc.) to retrieve random data, the number of values generated from the current seed is compared against the maximum number of values that can be generated per seed determined during the design phase. If the maximum number of values has been reached, then a subroutine (RAND_add, etc.) is used to reseed the cryptographically secure PRNG. The number of values generated from the current seed is then set to 0.
- After using a subroutine (RAND_bytes, etc.) to retrieve random data, the number of values generated from the current seed is incremented by 1.

(4) ***Software Security Requirement:*** The process shall limit the access to the the internal state of the cryptographically secure PRNG to the fewest users possible.

Security artifacts present in design:

- Users that will require access to the internal state of the cryptographically secure PRNG, and access permissions to give only those users access is determined.

Security artifacts present in code:

- When reading the internal state of the cryptographically secure PRNG, the access permissions are compared against what was determined during the design phase. The internal state is only used if the access permissions match the expected values.

(5) ***Software Security Requirement:*** The process shall use a process recommended by NIST to seed the cryptographically secure PRNG in use, and discard the seed value after seeding.

Security artifacts present in design:

- The process for seeding the cryptographically secure PRNG is determined from NIST requirements.

Security artifacts present in code:

- When the cryptographically secure PRNG requires seeding, the process for seeding the cryptographically secure PRNG determined during the design phase is used. The seed value is then immediately discarded after seeding.

(6) ***Software Security Requirement:*** The process shall provide the seed with enough entropy to satisfy NIST requirements for the cryptographically secure PRNG in use.

Security artifacts present in design:

- The amount of entropy needed for seeding is determined from NIST requirements for the cryptographically secure PRNG in use.

Security artifacts present in code:

- Before seeding the cryptographically secure PRNG, a call is made to read (from /dev/random, etc.) in the required amount of entropic data determined during the design phase. The retrieved entropic data is used to seed the cryptographically secure PRNG.

(7) ***Software Security Requirement:*** The process shall use entropic data to seed a cryptographically secure PRNG, and provide cryptographic algorithms or protocols that require unpredictable data values generated from the cryptographically secure PRNG.

Security artifacts present in code:

- When a cryptographic algorithm or protocol requires unpredictable data, entropy gathered from the system is not given to the cryptographic algorithm or protocol directly, and instead is used to seed a cryptographically secure PRNG. A subroutine (RAND_bytes, etc.) is called to retrieve unpredictable data from the cryptographically secure PRNG, which is given to the cryptographic algorithm or protocol.

(8) ***Software Security Requirement:*** The process shall estimate 1 bit of entropy for each character pressed on the keyboard, but only if the character is different than the previous character.

Security artifacts present in design:

- A method for collecting keyboard presses in the operating system in use is determined.

Security artifacts present in code:

- Key presses are collected using the method determined during the design phase. If the key that was pressed is the same as the previous key, the new key event is discarded. Otherwise, the key press is given to a subroutine (RAND_add, etc.) that adds entropy to the cryptographically secure PRNG, and 1 bit is given as the entropy estimation.

(9) ***Software Security Requirement:*** The process shall estimate log2(time) - 1 bits of entropy for the timing between local keyboard events, starting with the third event.

Security artifacts present in design:

- A method for collecting the time of keyboard presses in the operating system in use is determined.

Security artifacts present in code:

- The timing of key presses is collected using the method determined the design phase. If the key press is the first or second, the timing event is discarded. Otherwise, the timing event is subtracted from the timing event of the last key press, and is given to a subroutine (RAND_add, etc.) that adds entropy to the cryptographically secure PRNG. The entropy estimation given to the same subroutine is log2(time between events) - 1.

(10) ***Software Security Requirement:*** If mouse events from a user account cannot be detected from another user account, the process shall estimate 1 bit of entropy per local mouse event.

Security artifacts present in design:

- Whether a potential attacker can detect mouse events of one user account from another user account is determined. If an attacker can detect mouse events from a different account, mouse events are not used as a source of entropy.
- A method for collecting mouse events in the operating system in use is determined.

Security artifacts present in code:

- When collecting mouse event data, the events are collected using the method determined during the design phase. The mouse event data is then given to a subroutine (RAND_add, etc.) that adds entropy to the cryptographically secure PRNG, and 1 bit is given as the entropy estimation.

(11) ***Software Security Requirement:*** If a clock resolution of milliseconds or greater is available, the process shall count 0.5 bits of entropy for the timing between incoming packets, starting with with the third event.

Security artifacts present in design:

- Whether a clock resolution of milliseconds or greater is available to the process is determined during the design phase. If a clock resolution of milliseconds or greater is not available, network packet timings will not be used as a source of entropy.
- A method for collecting incoming network packet events in the operating system in use is determined.

Security artifacts present in code:

- When collecting the timing of incoming network packet events, the events are collected using the method determined during the design phase. If the network packet event was the first or second, the time event is discarded. Otherwise, the timing event is subtracted from the timing event of the last incoming network packet, and is given to a subroutine (RAND_add, etc.) that adds entropy to the cryptographically secure PRNG. The entropy estimation given to the same subroutine is 0.5.

(12) ***Software Security Requirement:*** For entropy estimates that are not from keyboard events, network packet timings, or mouse events, the process shall divide the estimate of entropy from a data set by 8.

Security artifacts present in design:

- Possible sources of entropy are determined.
- An entropy source that is not from keyboard events, network packet timings, or mouse events is evaluated to estimate the amount of entropy that the source provides. That value is then divided by 8, to get the conservative entropy estimation.

Security artifacts present in code:

- After entropic data is gathered from a source determined during the design phase, the data is given to a subroutine (RAND_add, etc.) that adds entropy to the cryptographically secure PRNG. The entropy estimation given to the same subroutine is the conservative entropy estimation value determined during the design phase.

(13) ***Software Security Requirement:*** If a user does not have to remember a password/key required for a cryptographic algorithm or protocol, the process shall use a cryptographically secure random data generator to generate the password/key.

Security artifacts present in design:

- Whether a user is required to remember a password/key required by a cryptographic algorithm or protocol is determined.
- If a user is not required to remember a password/key for a cryptographic algorithm or protocol, then a cryptographically secure random data generator (Hash_DRBG, etc.) is selected.

Security artifacts present in code:

- When generating a password/key for a cryptographic algorithm or protocol, a call is made to a subroutine (RAND_bytes,etc.) that uses the cryptographically secure PRNG selected during the design phase to generate random data.

(14) ***Software Security Requirement:*** If the user is required to remember a password for a cryptographic algorithm or protocol, the process shall use a random password generator recommended by NIST to provide passwords for users.

Security artifacts present in design:

- Whether a user is required to remember a password/key required by a cryptographic algorithm or protocol is determined.
- If a user is required to remember a password/key for a cryptographic algorithm or protocol, then a NIST recommended random password generator (APG, etc.) is selected.

Security artifacts present in code:

- When generating a password/key for a user to remember, a call is made to a subroutine that uses the random password generator selected during the design to generate a new password/key.

(15) ***Software Security Requirement:*** If a user generated password will be used in a cryptographic algorithm or protocol, the process shall use a password-key derivation function described in PKCS #5 to create a cryptographic key from a user supplied password.

Security artifacts present in design:

- Whether a user generated password will be used in a cryptographic algorithm is determined.

Security artifacts present in code:

- When converting a password for use as a key in a cryptographic algorithm or protocol, the password is given to a subroutine that implements PKCS #5. The, the value returned by the subroutine is given to the cryptographic algorithm or protocol.

(16) ***Software Security Requirement:*** The process shall use a random number generation algorithm that is currently recommended by NIST.

Security artifacts present in design:

- A NIST recommended cryptographically secure PRNG (Hash_DRBG, etc.) is selected.

Security artifacts present in code:

- When random data is required by a cryptographic algorithm or protocol, a call is made to a subroutine (RAND_bytes, etc.) that uses the cryptographically secure PRNG selected during the design phase to generate random data.

4.2.2.6 Cryptographic Algorithms and Protocols

This section presents the security artifacts from software security requirements imposed on cryptographic algorithms and protocols.

(1) ***Software Security Requirement:*** The process shall use a random number generation algorithm that is currently recommended by NIST.

Security artifacts present in design:

- A NIST recommended cryptographically secure PRNG (Hash_DRBG, etc.) is selected.

Security artifacts present in code:

- When random data is required by a cryptographic algorithm or protocol, a call is made to a subroutine (RAND_bytes, etc.) that uses the cryptographically secure PRNG selected during the design phase to generate random data.

(2) ***Software Security Requirement:*** The process shall establish secret keys between two parties using a key establishment scheme that is currently recommended by NIST.

Security artifacts present in design:

- A NIST recommended key establishment scheme (Discrete logarithm based scheme, etc.) is selected.

Security artifacts present in code:

- When a password/key needs to be established with a peer for the cryptographic algorithm or protocol to operate correctly, the key establishment scheme determined during the design phase is used.

(3) ***Software Security Requirement:*** The process shall use the current NIST recommendation for key length for the cryptographic algorithm or protocol in use.

Security artifacts present in design:

- The key length to be used with the cryptographic algorithm or protocol (256-bit, etc.) is determined, and based on NIST recommendations.

Security artifacts present in code:

- Whenever a cryptographic algorithm or protocol is used, the exact key length determined the during design phase is used.

(4) ***Software Security Requirement:*** The process shall use a hash algorithm that NIST currently recommends.

Security artifacts present in design:

- A hash algorithm to be used throughout the process (SHA-256, etc.), and currently recommended by NIST is selected.

Security artifacts present in code:

- Whenever the use of hash function is required, a call is made to a subroutine (EVP_DigestFinal_ex) that implements the hash function determined during the design phase.

(5) ***Software Security Requirement:*** Before giving data to a hash function, the process shall generate a random string of data whose length is equal to the length of the hash function output, then pad the end of the random string with zeros to equal the length of the internal block size of the hash function. The process shall concatenate this newly generated string before and after the data that was destined for the hash function, then give this large string concatenation to the hash function. The process shall then give the result of the hash to the hash function a second time, and discard the result of the first hash, only using the result of the second hash. The process shall store the random string with the hash result so the process can be duplicated with the same random string for any required verification.

Security artifacts present in design:

- The output size of the hash function in use is determined.
- The internal block size of the hash function in use is determined.

Security artifacts present in code:

- Before any data is given to a hash function, a subroutine (RAND_bytes, etc.) is called to get a string of random data equal to the length of the hash function output size determined during the design phase. Zeros are then concatenated to this string, until the entire length of the string reaches the length of the hash function internal block size determined during the design phase. This newly created string is concatenated before and after the data destined for the hash function, and then given to a subroutine (EVP_DigestFinal_ex, etc.) that implements the hash function selected during the design phase. The output from the hash function is then given to the subroutine (EVP_DigestFinal_ex, etc.) that implements the hash function, and the result of the first hash is discarded. The randomly generated data is then stored with the hash result, so it can be reused during verification.

(6) ***Software Security Requirement:*** If non-encrypted data needs to be verified for integrity and authenticity but not verified as to what party generated the data, the process shall use a message authentication algorithm that is currently recommended by NIST. The process shall keep the key used for authentication a secret.

Security artifacts present in design:

- Whether non-repudiation is required for data, where data can be proven to come from only once source, is determined.
- Whether the data will be encrypted is determined.
- If non-encrypted data needs authentication without non-repudiation, then a message authentication algorithm (HMAC-SHA256, etc.) currently recommended by NIST is selected.
- Users that will require access to the key used by the message authentication algorithm, and access permissions to give only those users access is determined.

Security artifacts present in code:

- When reading the key to be used by the message authentication algorithm, the access permissions are compared against what was determined during the design phase. The key is only used if the access permissions match the expected values.
- When providing authentication without non-repudiation to unencrypted data, the data is given to a subroutine (HMAC_Final, etc.) that computes the authentication value using the algorithm selected during the design phase.

(7) ***Software Security Requirement:*** If encrypted data needs to be verified for integrity and authenticity but not verified as to what party generated the data, the process shall use a block cipher mode when encrypting the data that supports authenticated encryption. The process shall keep the key used for authentication a secret.

Security artifacts present in design:

- Whether non-repudiation is required for data, where data can be proven to come from only once source, is determined.
- Whether the data will be encrypted is determined.
- If encrypted data needs authentication without non-repudiation, then a block cipher mode that supports authenticated encryption (CCM, etc.) currently recommended by NIST is selected.
- Users that will require access to the key used by the block cipher mode, and access permissions to give only those users access is determined.

Security artifacts present in code:

- When reading the key to be used by the block cipher mode, the access permissions are compared against what was determined during the design phase. The key is only used if the access permissions match the expected values.
- When providing authentication without non-repudiation to encrypted data, the data is given to a subroutine (EVP_DecryptFinal_ex, etc.) that uses the block cipher mode selected during the design phase.

(8) ***Software Security Requirement:*** If non-repudiation is required for encrypted or non-encrypted data to prove that the data came from only one source, the process shall sign the data with a signature algorithm currently recommended by NIST. The process shall keep any private keys used by the signature algorithm a secret.

Security artifacts present in design:

- Whether non-repudiation is required for data, where data can be proven to come from only once source, is determined.
- If encrypted or non-encrypted data needs authentication with non-repudiation, then a signature algorithm (DSA, RSA, etc.) currently recommended by NIST is selected.
- Users that will require access to the key, used by the signature algorithm, is determined, and access permissions to give only those users access is determined.

Security artifacts present in code:

- When reading the key to be used by the digital signature algorithm, the access permissions are compared against what was determined during the design phase. The key is only used if the access permissions match the expected values.
- When providing authentication for data requiring non-repudiation, the data is given to a subroutine (EVP_SignFinal, etc.) that signs the data using the signature algorithm selected during the design phase.
- When authenticating data requiring non-repudiation, the data and the signature is given to a subroutine (EVP_VerifyFinal, etc.) that will verify the signature of the data using the signature algorithm selected during the design phase.

(9) ***Software Security Requirement:*** If authentication of encrypted or unencrypted data is being used by two remote hosts, the sending process shall concatenate an increasing number value to the authentication key before generating an authentication value. The sending process shall store the number value with the authentication value. The receiving process shall use this number during authentication, and ensure this number value increases with every authentication check. The process shall ensure the data space for the number value is large enough to prevent repeat numbers.

Security artifacts present in design:

- If authentication of data is being used between two remote hosts is determined.
- A digit length that supports a number that is twice as long as the expected number of authentication requests is determined.

Security artifacts present in code:

- When a sending host is providing authentication to a remote receiver host, a value corresponding to the number of times data was given to the authentication method is concatenated to the data. The concatenated string is given to the authentication method.
- When a receiving host is authenticating data from a remote sending host, the value corresponding to the number of times data was given to the authentication method is checked. The data is only used if the number value is larger than the previous value. Then, the number value is concatenated to the data, and given to the authentication method as usual.

(10) ***Software Security Requirement:*** The process shall provide an unpredictable key to a stream cipher for keystream generation.

Security artifacts present in code:

- When a stream cipher requires a key for keystream generation, a call is made to a subroutine (RAND_bytes, etc.) that provides cryptographically secure random data.

(11) ***Software Security Requirement:*** The process shall never provide a duplicate key to a stream cipher for keystream generation.

Security artifacts present in design:

- A digit length that supports a number that is twice as long as the expected number of keystream generation operations is determined.

Security artifacts present in code:

- When calling a subroutine (RAND_bytes, etc.) for cryptographically secure random data, the amount requested is equal to the digit length determined during the design phase.

(12) ***Software Security Requirement:*** The process shall encrypt sensitive data using a block cipher that is currently recommended by NIST, using a block cipher mode currently recommended by NIST. The process shall keep the encryption key a secret.

Security artifacts present in design:

- A block cipher (AES, etc.) that is currently recommended by NIST is selected.
- A blocker cipher mode (CCM, etc.) that is currently recommended by NIST is selected.

Security artifacts present in code:

- When encrypting data, it is given to a subroutine (EVP_EncryptFinal_ex, etc.) that uses the block cipher, and block cipher mode selected during the design phase.
- When decrypting data, it is given to a subroutine (EVP_EncryptFinal_ex, etc.) that uses the block cipher, and block cipher mode selected during the design phase.

(13) ***Software Security Requirement:*** The process shall follow NIST guidelines for key management, to validate any key reported by another machine before use.

Security artifacts present in design:

- A key management scheme (X.509, etc.) recommended by NIST is selected to verify the keys reported by peers.

Security artifacts present in code:

- When a third party provides a key, it is used in the key management method selected during the design phase to verify its identity.

(14) ***Software Security Requirement:*** The process shall designate a trusted third party for key verification.

Security artifacts present in design:

- A trusted third party for key verification is selected.

Security artifacts present in code:

- When a host provides a key, it is given to the third party, selected during the design phase, for verification.

(15) ***Software Security Requirement:*** The process shall encrypt sensitive data using a block cipher that is currently recommended by NIST, using a block cipher mode currently recommended by NIST. The process shall keep the encryption key a secret.

Security artifacts present in design:

- A block cipher (AES, etc.) that is currently recommended by NIST is selected.
- A blocker cipher mode (CCM, etc.) that is currently recommended by NIST is selected.

Security artifacts present in code:

- When encrypting data, it is given to a subroutine (EVP_EncryptFinal_ex, etc.) that uses the block cipher, and block cipher mode selected during the design phase.
- When decrypting data, it is given to a subroutine (EVP_EncryptFinal_ex, etc.) that uses the block cipher, and block cipher mode selected during the design phase.

(16) ***Software Security Requirement:*** If storage is required for a password/key for later retrieval, the process shall encrypt the password/key with a second password/key, immediately discarding the original password/key. The process shall not store the second password/key with the encrypted password/key.

Security artifacts present in design:

- Whether the usage of key/password will require validation or later retrieval is determined during the design phase.

Security artifacts present in code:

- If a key/password requires storage for validation, it is given to a hashing procedure. The key/password is then discarded and the results from the secure hashing procedure are stored.

(17) ***Software Security Requirement:*** If storage is required for a password/key for later validation, the process shall give the string to a secure hash procedure. The process shall discard the password after providing it to the secure hashing procedure, only storing the results of the secure hashing procedure.

Security artifacts present in design:

- Whether the usage of key/password will require validation or later retrieval is determined during the design phase.
- A source of a second key/password is determined.

Security artifacts present in code:

- If a key/password requires storage for later retrieval, it is given to a subroutine that provides encryption (EVP_EncryptFinal_ex, etc.), and uses the second password/key determined during the design phase for encryption. The original password/key is immediately discarded, and the second/password key is not stored on the system with the encrypted password/key.

4.2.3 Verification with the Security Artifacts

Each software security requirement used in a project must be verified at the end of the design, and implementation phases using the security artifacts given in the previous section. If a security artifact is not verified, then whether the corresponding software security requirement is used correctly is unknown. The verification process differs slightly depending on the phase of development, but each only uses the security artifacts designated for the particular phase. The verification processes are as follows:

Design phase verification compares the design artifacts of every software security requirement included in the project with the design documentation. The design artifacts are a collection of decisions, and if a comparable decision cannot be found in the design documentation, then the design documentation cannot be considered complete. Once all the security artifacts found in design have been verified in the design documentation, the implementation phase can begin.

Implementation phase verification differs slightly the design phase verification, because each code artifact must be critically analyzed to determine all possible locations in code the artifact will be present. Each place in code is then compared against the code artifact. If the code does not match the code artifact, it is re-written until it can be verified. The code artifact has finished verification when all expected places in code contain the artifact. The entire implementation phase is considered complete when all security artifacts found in code have been verified.

Once the software security requirements have been verified in the design and implementation, a level of assurance is provided for correct translation of the software security requirements. However, verification does not guarantee correct inclusion of the requirements nor does it guarantee bug-free software, so the remaining full VV&T strategies presented in 4.1.1 must be used along with verification.

4.3 Summary

This chapter presents how to use VV&T with the software security requirements to ensure their proper translation into the design and implementation of a project. Only full VV&T is supported by this research, because it is the only VV&T strategy where security concerns start during requirements engineering. Validation and testing strategies provided by full VV&T are easily used with the software security requirements because the requirements were designed to be testable. Verification in VV&T is supported by the security artifacts which were generated from each software security requirement. The security artifacts support developers with limited security knowledge, by providing examples of what is expected in design and code. Lastly, a verification process that uses the security artifacts is provided for the design and implementation phases, to clearly explain how verification must occur.

Chapter 5

Support for Software Security Requirements Selection

The list of software security requirements is long, and many projects will not have the need for all the requirements since a project may only face a subset of vulnerabilities that the requirements are designed to prevent. This creates a problem for developers; unnecessary requirements increase the workload in the design and implementation as verification and validation strategies are conducted on requirements that are never used. Unfortunately, the software security requirements are targeted at developers without security knowledge, who may not have the capability of removing the unnecessary requirements correctly.

5.1 Software Security Requirements Tree

The Software Security Requirements Tree (SSRT) associates software security requirements with general software requirements. This places a condition on the use of the SSRT - a development team cannot select the necessary software security requirements until the more general, non-security related, software requirements have been generated. Once all the general software requirements have been written, they can be used to select which software security requirements are needed.

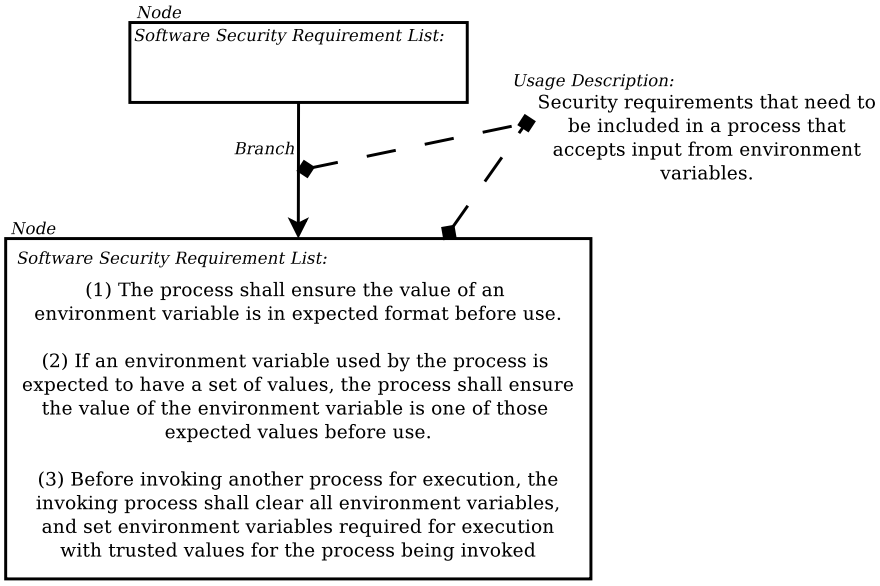


Figure 5.1: Elements of the SSRT

The SSRT contains four basic elements: nodes, branches, usage descriptions, and software security requirement lists. Figure 5.1 shows a portion of the SSRT with the elements labeled in italicized text. The software security requirement list in the top node is blank because the node contains no software security requirements.

Nodes represent the general requirements of a project. A node can connect to another node through a branch, where a parent/child relationship is created between the connected nodes. In the given figure, the node with the arrow pointing to it is the child, while the node connected at the other end is the parent. The child node represents a subset of the general requirements contained in the parent node. In this example, the parent node is input, while the child node is a subset of input, more specifically, input from an environment variable. A node can have multiple children nodes, but only one parent node.

Branches connect nodes in the SSRT. A single branch is connected to two nodes, one of which is a child, and the other a parent. When using the SSRT, the branches must be traversed from the parent node, to the child node. In figure 5.1, the child node has an arrow pointing to it, while the node connected on the other end is the parent.

Usage descriptions are associated with a branch and the branch's child node, which is shown in the provided example with dashed arrows. The usage description clearly states in positive terms what general requirements the child node represents. The simultaneous association with the branch and child node may seem unnecessary since a branch can only be connected to one child node, but the distinction was made for the different scenarios the structure faces. When creating or navigating the SSRT, logically it makes sense to associate the usage description with the branch. However, when using the prototype SSRT tool, described in 5.1.3, it is more logical to associate the usage description with the child node.

Software security requirement lists are contained within a node. When a node repre-

sents general requirements contained in a project, the software security requirements in the associated list must be used by the project in development.

5.1.1 Creating the SSRT Structure

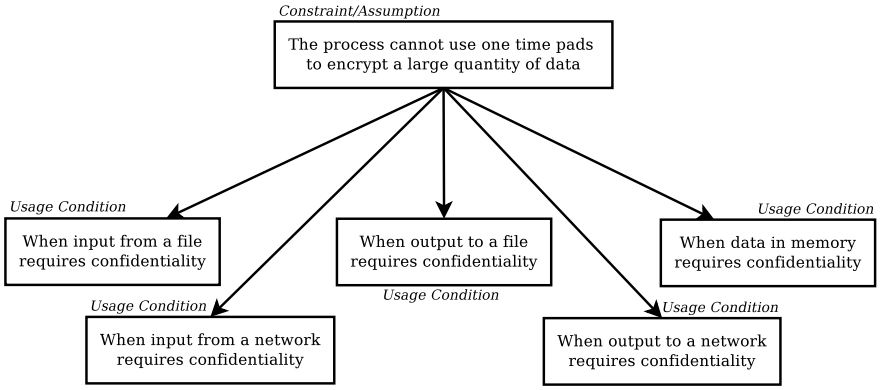
The elements of the SSRT are used to create a tree structure based on the constraints/assumptions. The tree connects general requirements directly to constraints/assumptions, which then have associated software security requirements. Therefore, the SSRT provides a direct relationship between general requirements and software security requirements. The structure of the SSRT is created using an algorithm, so that general software requirements are correctly linked to needed software security requirements.

Figure 5.2 shows the SSRT creation algorithm, which operates on each constraint/assumption, and decomposes them into *usage conditions*. The decomposition into usage conditions is located on line 3 of the creation algorithm, and is the critical step at linking general requirements to software security requirements. The usage conditions specify when the constraint/assumption can be violated, which is the only time the associated software security requirements are necessary in a project. An example constraint/assumption decomposition is shown in figure 5.3(a). The example starts with a constraint involving data confidentiality, which is evaluated to determine all scenarios, or usage conditions, where the data confidentiality constraint can be violated. In this example, there are five usage conditions that may result in the constraint being violated.

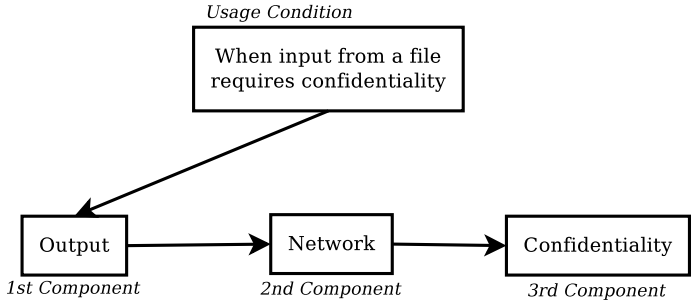
```
 1: Create root
 2: for each constraintAssumption from list.constraintAssumptions do
 3:    list.conditions ⇐ constraintAssumption.usageConditions
 4:    for each condition from list.conditions do
 5:       queue.components ⇐ condition.componentQueueStartingWithMostBroad
 6:       currentNode ⇐ root
 7:       currentComponentText ⇐ string.empty
 8:       currentComponent ⇐ queue.components.begin
 9:       while queue.components.notEmpty do
10:          queue.components.removeFirst
11:          if currentNode.list.branches.component contains component then
12:             currentNode ⇐ branch.childNode
13:          else
14:             Create newNode
15:             Create newBranch
16:             newBranch.childNode ⇐ newNode
17:             currentComponentText ⇐ currentComponentText + '.' + currentComponent
18:             newBranch.descriptor ⇐ currentComponentText
19:             Add to currentNode.list.branches ⇐ newBranch
20:             currentNode ⇐ newNode
21:          end if
22:          currentComponent ⇐ queue.components.next
23:       end while
24:       add constraintAssumption to currentNode.list.constraintAssumptions
25:    end for
26: end for
```

Figure 5.2: The SSRT Creation Algorithm

The SSRT creation algorithm then continues on line 5, to decompose usage conditions into *components*, which are ordered in a queue from most broad to most specific. One usage condition in the provided example is decomposed into components, as shown in 5.3(b). The logic in the example is that output to a network requiring confidentiality is only possible with output to a network, which is only possible with any form of output.



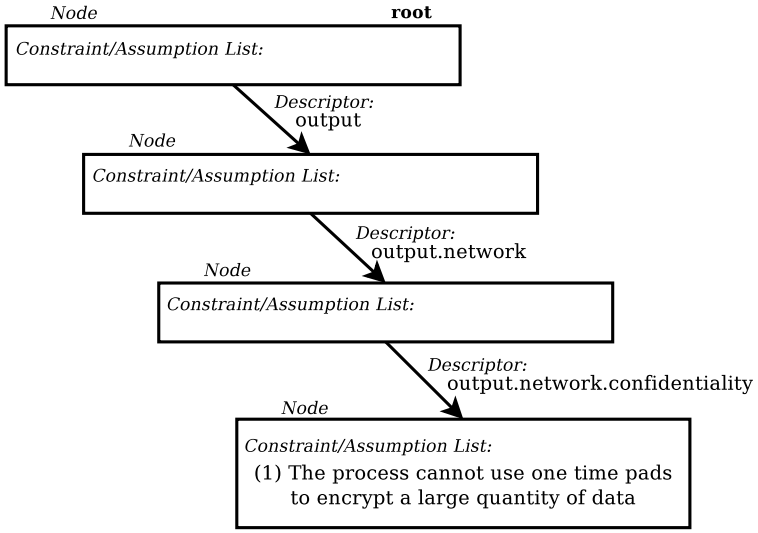
(a) Constraint decomposed into usage conditions



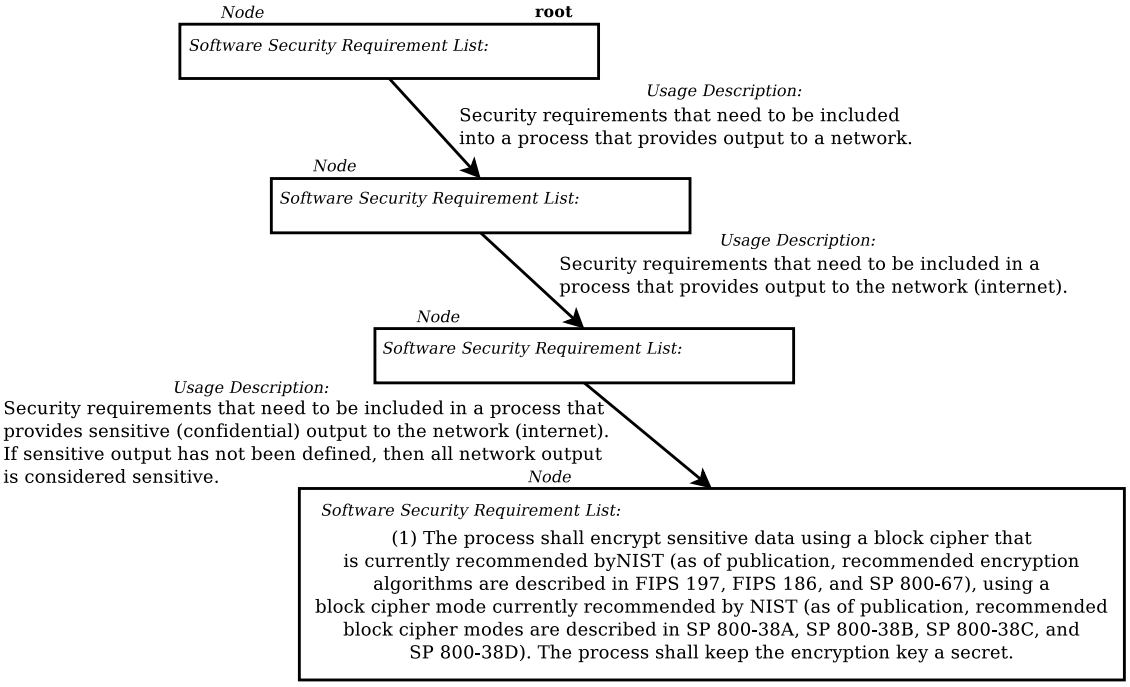
(b) Usage condition decomposed into components

Figure 5.3: Example of Critical Steps in SSRT Creation Algorithm

The remaining portion of the SSRT creation algorithm uses the component queue to determine a location in the tree. The constraint/assumption that led to the component queue is then added to that location in the tree, and parts of the component queue are attached to the branches as a *descriptor* along the path. Figure 5.4(a) depicts this process by taking the component queue shown in 5.3(b), and placing it into a tree as if it were the only component queue used so far. The tree is initialized with one node, the root node, and when a constraint/assumption has no usage conditions, it will be added to that node.



(a) Components from 5.3(b) inserted into tree using the creation algorithm



(b) Tree structure from (a) has usage description and software security requirement lists filled in

Figure 5.4: Example of Critical Steps in SSRT Creation Algorithm

After completing the creation algorithm on all constraints/assumptions, a tree structure is produced. Each node in the tree may have constraints/assumptions associated with it, and may have children nodes connected through branches. Each branch must have a usage description, but the creation algorithm has produced descriptors instead. The next step is converting those descriptors into a positive usage description, that provide details as to when this branch is used. The next step is converting the constraints/assumptions in a

node to lists of software security requirements associated to those constraints/assumptions. The conversion steps of the continuing example are shown in figure 5.4(b), which is the same as figure 5.4(a) except with usage descriptions and a software security requirement list. The final step is considering whether the usage description utilizes any potential information security requirements. If it does, an additional statement must be added to the branch usage description to specify that the branch must be used in all cases of the information security requirement absence. Currently, this is only occurs with sensitive data. If a project does not define sensitive data, then all data must be considered sensitive. Once all the components and constraints/assumption have been converted to usage descriptions and software security requirement lists, the tree structure is ready for use, and can be used to gather software security requirements as described in 5.1.2.

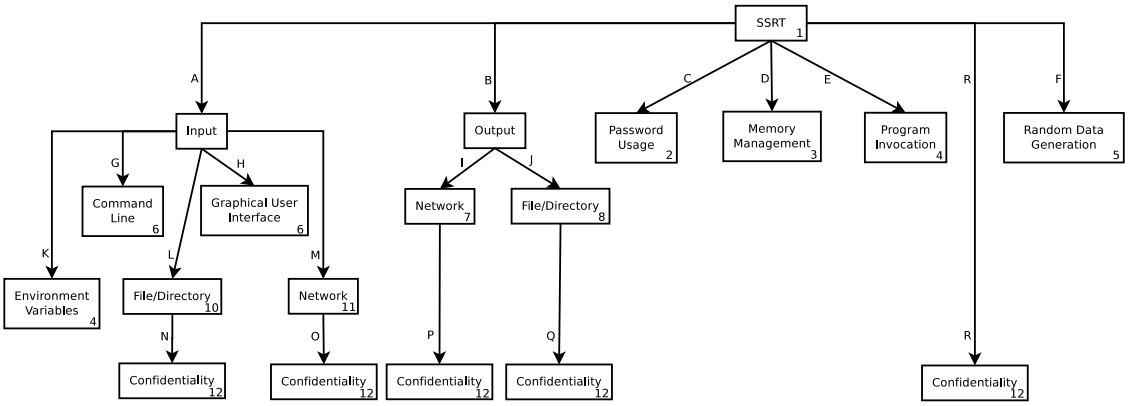


Figure 5.5: The Structure of the SSRT

The tree structure that resulted from the SSRT creation algorithm is the SSRT, shown in figure 5.5. The letters in the figure correspond to a branch usage description, while the numbers correspond to software security requirement lists. Both the branch usage descriptions, and the lists of software security requirements are presented in greater detail in appendix C.

5.1.2 Using the SSRT

The SSRT structure presented in the previous section is used to gather the necessary software security requirements for a project. When gathering software security requirements, each node in the SSRT can have two states, selected and unselected. Every node, except for the root node, is initially in the unselected state. The root node is selected because it represents general requirements that are inherent in every project. An algorithm for using the SSRT is shown in figure 5.6, and starts by reviewing the usage description of each branch connected to the root node. If a usage description associated with a branch matches any general requirement, then the state of the child node on the branch changes to selected. The process is recursive, so all selected nodes must have their branch usage descriptions reviewed to determine if their children must also be selected.

Once no more nodes can be selected, the software security requirement lists associated with each selected node must be gathered. The gathered requirements must be included with the general requirements for the project.

```
1: BEGIN HERE
2: for each node in list.nodes do
3:    node ⇐ unselected
4: end for
5: list.nodes.root ⇐ selected
6: call function searchDepthFirst(list.nodes.root)
7: securityRequirements ⇐ call function gatherDepthFirst(list.nodes.root, list.empty)
8: add securityRequirements to project
```

```
1: function searchDepthFirst(node)
2: for each branch to node do
3:    if any generalRequirement == branch.description then
4:       branch.childNode ⇐ selected
5:       call function searchDepthFirst(branch.childNode)
6:    end if
7: end for
```

```
1: function gatherDepthFirst(node, list.gatheredRequirements)
2: if node == selected then
3:    for each requirement in node.list.requirements do
4:       add to list.gatheredRequirements ⇐ requirement
5:    end for
6:    for each branch to node do
7:       call function gatherDepthFirst(branch.childNode, list.gatheredRequirements)
8:    end for
9: end if
10: return  list.gatheredRequirements
```

Figure 5.6: The SSRT Usage Algorithm

5.1.3 Prototype SSRT Tool

The SSRT is only effective when the usage algorithm is executed correctly, so a prototype tool has been created to help enforce the correct usage of the SSRT. Unfortunately, the tool cannot make all the decisions for a developer, because determining when to select a node requires the branch usage description to be compared against the general requirements. However, once the nodes are selected, the prototype tool can automatically generate the entire list of necessary software security requirements. The tool also prevents mistakes on

node selection, by preventing a child node from being selected unless all parent nodes are also selected.

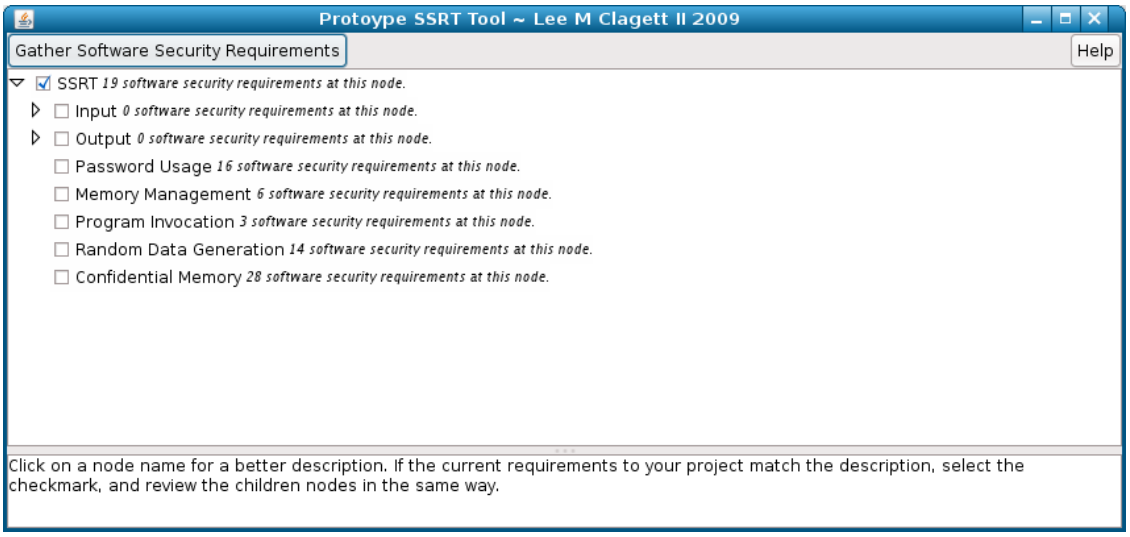
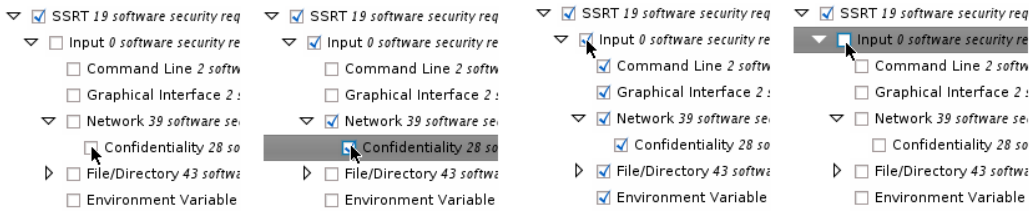


Figure 5.7: Initial State of the Prototype SSRT Tool

The prototype SSRT tool is written in Java, and uses the Java Swing library to provide a graphical interface to the user. The user interface is completely cross-platform and available for execution on any operating system supporting the full Java Runtime Environment (JRE). Launching the program requires the installation of the JRE, which is available from Sun Microssystems [52]. When the prototype tool is executed, a directory style tree is shown to the user, and represents the SSRT. Figure 5.7 is a screenshot of the initial state of the prototype tool, which shows the root node, seven children nodes to the root, and a bottom text are containing basic usage instructions. Each checkbox corresponds to a node in the SSRT, and is selected when the corresponding branch usage description matches a general requirement. The root node has no usage description, and cannot be unselected because it is always selected in the SSRT usage algorithm.

Each node has a short descriptive name shown next to its checkbox, and is based on the usage description. A mouse click on the short descriptive name reveals the entire usage description in the bottom text area. This information is used by a developer to determine if that particular node matches any general requirements, and if it does, the checkbox at that node is selected. The expand button is then given a mouse click, which is either a plus sign (+) or an arrow depending on the operating system. In the provided figure, the operating system utilizes an arrow, which is directly to the left of every checkbox (node). Clicking the expand button will reveal more nodes whose usage descriptions are compared against the general requirements.



(a) Left image shows the tree before the click, and the right image is after the click. Notice that a single selection to a child automatically selects all parent nodes

(b) Left image shows the tree before the click, the right image is after the click. Notice that a single de-selection to a parent node automatically de-selects all child nodes

Figure 5.8: Correct SSRT Usage Enforcement by the Prototype SSRT Tool

The prototype tool has several features that streamline the node selection of the SSRT. Anytime a checkbox is given a click, one of two operations occur. If the checkbox is being selected, all parent nodes up the structure are also automatically selected. If the checkbox is being unselected, all children nodes down the structure are unselected. Figure 5.8 gives a graphical depiction of the two operations. The operations were designed into the prototype tool to enforce the correct usage of the SSRT; a child node cannot be selected unless it's parent node is selected.

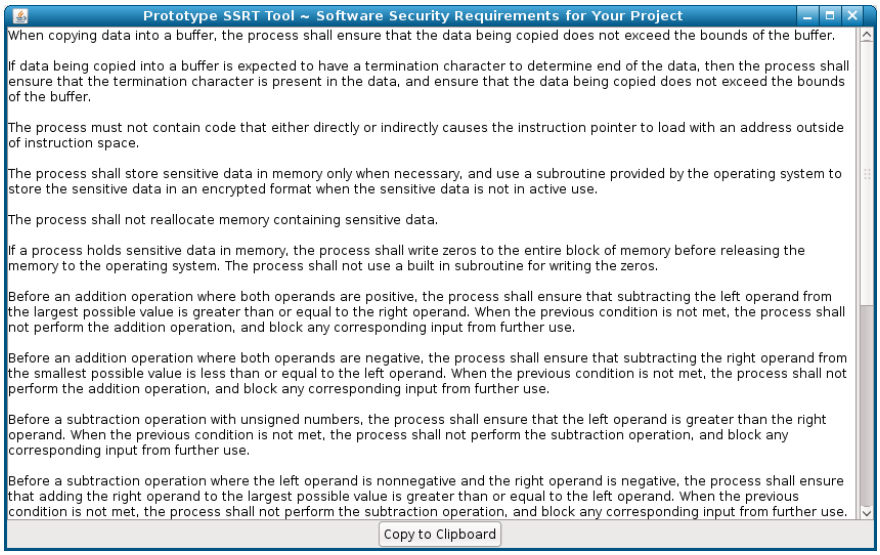


Figure 5.9: Software Security Requirements Gathered by the Prototype SSRT Tool

Once a user has completed node selection, a button in the prototype tool labeled "Gather Software Security Requirements" provides an easy method for software security requirements gathering. The button is shown in the top left of figure 5.7, and when pressed, the prototype tool gathers the software security requirements associated with each of the selected nodes, removes duplicates so each requirement only appears once, and presents a new window with

the requirements as shown in figure 5.9. The window displays a text area of all the software security requirements, which can be copied to the operating system clipboard, for pasting into other documents as desired.

Appendix D contains the entire source code for the Prototype SSRT Tool.

5.2 Summary

The inclusion of unnecessary software security requirements will inflate the length of development. The problem is further complicated because a developer lacking security knowledge may be unable to remove the unnecessary software security requirements. The SSRT solves both of these issues, by using the general requirements to narrow the scope of the software security requirements. Since the SSRT uses the general requirements of a project, a developer without security knowledge is capable of removing the unnecessary software security requirements. A prototype tool was created to enforce the correct use of the SSRT, and automatically gather required software security requirements based on input from the user. Lastly, the process that generated the SSRT is provided, if any future work needs to add a software security requirement.

Chapter 6

Conclusions and Future Work

This research presents an approach for software vulnerability prevention, that can be used by developers with limited security knowledge. The approach provides security requirements for use in development lifecycles, and support material for full access-driven VV&T strategies. The approach has three main components: the software security requirements, the security artifacts, and the SSRT. The software security requirements were reviewed by software engineers and security experts, and determined to be: (a) testable and verifiable, (b) clear, concise and non-ambiguous, (c) implementable by software engineers without security knowledge, (d) usable in modern development lifecycles, and (e) capable of preventing modern software vulnerabilities when used correctly in development. The security artifacts are used in verification strategies to ensure that the software security requirements are properly translated into the design and implementation. Proper translation ensures correct usage of the software security requirements, and therefore whether the vulnerabilities represented by the constraints/assumptions in the taxonomy have been successfully prevented. Finally, the SSRT provides support for software security requirement selection, which uses project characteristics to narrow the scope of the requirements. A graphical interface for the SSRT is provided through a prototype Java tool, to support the identification and selection of appropriate software security requirements.

The approach in this research has several advantages to secure software development. The first advantage is that the approach starts during requirements engineering, at the beginning of the development lifecycle. Approaches that start during implementation or testing have started concerns too late, and provide less time for critically evaluating the system. This research is further separated from past work because each component has been carefully designed for use with limited support from a security expert. Additionally, both of these critical advantages were achieved without modification to the development lifecycle, making this approach suitable for projects currently using a classic approach to software development.

There is a chance that a future vulnerability will be discovered, that is not covered by any of the constraints/assumptions in the taxonomy of software vulnerabilities. If a new constraint/assumption is required, this approach cannot prevent against that vulnerability. Fortunately, the work by Bazaz outlines how to generate a new constraint/assumption, should the need exist [13]. This research then provides the necessary processes and algorithms

to generate software security requirements, security artifacts, and an updated SSRT from the new constraint/assumption.

6.1 Contributions

The components of this research provide new contributions to secure software development, and new future areas of research. The contributions are described below.

- Security requirements are essential for preventing vulnerabilities in software, however no list of security requirements currently exists. This research provides a list of requirements for security, called the *software security requirements*. This research further describes a *generation process* for any future software security requirements that are necessary.

  Software Security Requirements: The taxonomy of software vulnerabilities uses the object model of computing to identify software vulnerabilities by constraints and assumptions imposed on computer system resources. The software security requirements provided in this research are designed to prevent the violation of all the constraints/assumptions in the taxonomy. Correct usage of the software security requirements will mitigate the risk of software vulnerabilities represented in the taxonomy. The requirements are also testable/verifiable, usable in software development lifecycles, and implementable by software engineers with limited security experience.

  Generation Process: Future vulnerabilities may be discovered, requiring an update to the taxonomy of software vulnerabilities. Any change to the taxonomy of software vulnerabilities will require new software security requirements to reflect the changes. This research describes a process for generating software security requirements from a constraint/assumption. The process will produce software security requirements that prevent the constraint/assumption from being violated, while remaining clear, concise, and testable/verifiable.

- Software requirements are only meaningful when they have been correctly translated into the design and implementation phases of development. If a software security requirement does not achieve proper translation into the later stages, the result could be a vulnerability. This research provides an assurance for correct translation of the software security requirements through the use of security artifacts, which are divided into two types: *design artifacts*, and *code artifacts*. Artifacts are products of the development process. The artifacts produced during requirements engineering are the actual software requirements.

  Design Artifacts: The design phase requires critical architectural decisions that affect the later stages of development. If a software security requirement is translated into the design phase improperly, the implementation will not be

correct. An assurance for a correct design is provided by the design artifacts. When a project has verified that the actual product from the design matches the design artifacts, then there is a good chance that the software security requirements have been correctly translated.

Code Artifacts: During the implementation phase, the requirements and the design are utilized to produce computer code that satisfies the goals set forth in the requirements. Any of the software security requirements can be translated incorrectly into code, and introduce a vulnerability. The code artifacts presented by this research prevent incorrect translation of the software security requirements into code. When all code artifacts have been verified in the actual code produced, then there is a good chance that the software security requirements have been correctly implemented.

- The entire list of software security requirements is lengthy, and some projects may only need a portion of the requirements. This research supplies the *SSRT*, which provides a process for selecting only the necessary software security requirements for a project. Furthermore, a graphical interface for the SSRT is provided through a *prototype Java tool*, to support the identification and selection of appropriate software security requirements.

  SSRT: The SSRT is a tree structure that can be used to select necessary software security requirements using a project's functional, but non-security related requirements. Since the SSRT leverages the non-security related requirements, a software developer with limited security knowledge can utilize the SSRT.

  Prototype Java Tool: A provided type Java tool is provided to support the usage of the SSRT. The tool provides a quicker method for software security requirements selection, and prohibits incorrect usage of the SSRT.

- The software security requirements have several targeted goals that must be met for absolute success. The requirements are carefully designed with the goals in mind, but to substantiate the goals, this research provides: a *software engineers survey*, and a *security experts survey*.

  Software Engineers Survey: Software engineers review of the software security requirements that substantiates the following claims: (a) the requirements are testable/verifiable, (b) the requirements are clear, concise, and non-ambiguous, (c) the requirements can be used in development lifecycles, and (c) the requirements can be implemented by developers with limited security knowledge.

  Security Experts Survey: Security experts review of the software security requirements that substantiates the following claims: (a) the requirements are clear, concise, and non-ambiguous, (b) use of the requirements prevents all modern software vulnerabilities. While the security experts were more apprehensive at claiming complete success at preventing all software vulnerabilities due to scope issues, their feedback showed that the software security requirements do mitigate the risk of potential software vulnerabilities.

6.2 Future Work

Research for secure software development does not end with the contributions of this work, and the material can be taken into several directions or evaluated further. Future possibilities related to this research are discussed below.

6.2.1 Evaluating Software Security Requirements in a New Project

The software security requirements are designed for use in full VV&T when security concerns start during requirements engineering. The same requirements were given to software engineers to substantiate several claims, including whether a developer without security knowledge could implement them. The software engineers gave the software security requirements high marks overall, but currently the requirements have never been used in a project. Likewise, the security artifacts and the SSRT are designed to provide help to developers with limited security knowledge, but neither has been used in such an environment. Future work could take the software security requirements, the security artifacts, and the SSRT and evaluate their use in a new development project.

6.2.2 Software Security Requirements in Partial VV&T

This research only provided support for full VV&T, but the software security requirements, and the security artifacts could be used at the end of development in a partial VV&T strategy. Partial VV&T reviews the design and implementation documentation, but does not influence the initial production of those documents. The approach would be less desirable since security concerns start so late, but could still be valuable at evaluating existing software. Primarily, research into partial VV&T will determine how to incorporate the security artifacts into a verification, and correction process. The correction process will be an approach that provides details on how to modify the design and implementation to achieve security artifact verification.

6.2.3 Evaluating the SSRT

The SSRT may not be the only approach to removing unnecessary software security requirements. Alternative methods can be explored and compared against the SSRT for efficiency, and effectiveness. In particular, new approaches will be evaluated to determine whether adding new software security requirements will be easier or more difficult.

6.2.4 Extending New Constraint/Assumption

A vulnerability discovered in the future may not be covered by the current list of constraints/assumptions. Future work could create constraints/assumptions for any new vul-

nerabilities, using the process provided by Bazaz [13]. Then, the process and algorithms in this research can be used to generate software security requirements, security artifacts, and produce an updated SSRT. The approach would remain the same, but the materials supporting the approach would be updated for modern attacks.

6.3 Summary

Software security is important, and will continue to be important as the number of vulnerabilities continue to increase. Approaches for preventing vulnerabilities are still in their infancy, but security researchers agree that the best prevention methods start at the beginning of development. This research provides such an approach, by supplying a full list of requirements for security that mitigate the risk of modern software vulnerabilities. No prior work has attempted to supply a list of security requirements, or provide an approach that limits the necessity of a security expert. While vulnerability free software cannot be guaranteed, the approach is unique, and provides several new possibilities for vulnerability prevention and further research opportunities.

Appendix A

Survey Format for the Software Security Requirements

A survey was provided to software engineers and security experts so that the claims from the software security requirements could be substantiated. The surveys contained statements linked to a single goal, and the subject matter experts provided a response on a scale from 1 to 10. A higher score reflects a statement that is more true, and therefore better achieves the associated goal. Each survey was carefully constructed to gain a useful evaluation from the subject matter experts through a web browser.

A.1 Survey for Software Engineers

The survey for software engineers evaluated four targeted goals of the software security requirements: (1) whether they are clear, concise, and non-ambiguous, (2) whether they are implementable by software engineers, (3) whether they are incorporable into existing lifecycles, and (4) whether they are testable/verifiable. All of the goals had to be evaluated for each individual software security requirement, so the survey contained four statements for each requirement. The software engineers used a series of radio buttons to indicate how true they felt the statements to be. Figure A.1 shows the first software security requirement, the four statements, and the four response areas. The four statements under the software security requirement in the figure are used for every software security requirement in the software engineers survey.

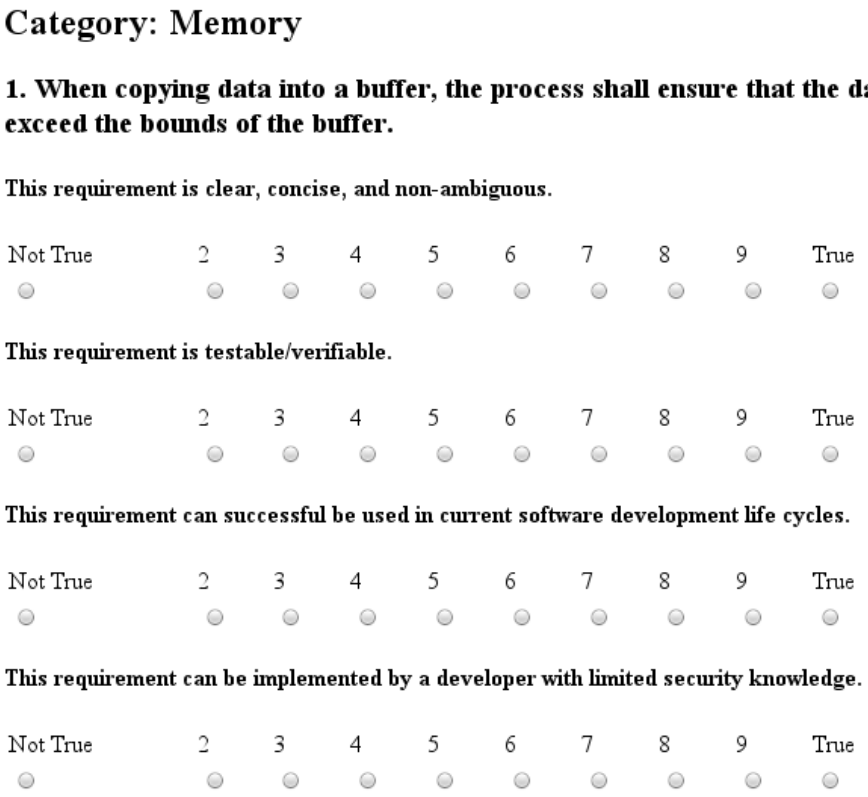


Figure A.1: Software Engineers Survey Example

A.2 Survey for Security Experts

The survey for security experts evaluated three targeted goals of the software security requirements: (1) whether they are clear, concise, and non-ambiguous, (2) whether they prevent the associated constraint/assumption from being violated, and (3) whether they prevent all software vulnerabilities. The first goal had to be evaluated for each individual software security requirement, the second goal had to be evaluated for each constraint/assumption, and the third goal had to be evaluated for each computer resource category. This lead to a corresponding statement for each software security requirement, constraint/assumption, and computer resource category. The software engineers used a series of radio buttons to indicate how true they felt the statements to be. Figure A.1 shows the first constraint/assumption, the associated software security requirements, the three statements, and the three response areas. Those statements are used for every software security requirement, constraint/assumption, and computer resource category respectively. The resource category is shown separate from the other statements because it was provided after all constraints/assumption in each category.

Category: Memory

Constraint/Assumption #1. Data accepted as input to a process and assigned to a buffer must occupy and modify only specific locations allocated in the buffer.

Requirements to prevent violation:

- **Requirement: When copying data into a buffer, the process shall ensure that the data being copied does not exceed the bounds of the buffer.**

  This requirement is clear, concise, and non-ambiguous.

  Not True 2 3 4 5 6 7 8 9 True

- **Requirement: If data being copied into a buffer is expected to have a termination character to determine end of the data, then the process shall ensure that the termination character is present in the data, and ensure that the data being copied does not exceed the bounds of the buffer.**

  This requirement is clear, concise, and non-ambiguous.

  Not True 2 3 4 5 6 7 8 9 True

These requirements sufficiently prevent constraint/assumption #1 from being violated.

Not True 2 3 4 5 6 7 8 9 True

(a) The response statement for each software security requirement, and each constraint/assumption

The use of all the requirements listed in this category (Memory) sufficiently prevent the vulnerabilities present in modern software in this category (Memory)?

Not True 2 3 4 5 6 7 8 9 True

(b) The response statement for each computer resource category

Figure A.2: Security Experts Survey Example

Appendix B

Survey Results for the Software Security Requirements

A survey was provided to software engineers and security experts so that the claims from the software security requirements could be substantiated. The surveys contained statements linked to a single goal, and the subject matter experts provided a response on a scale from 1 to 10. A higher score reflects a statement that is more true, and therefore better achieves the associated goal. Each survey was carefully constructed to gain a useful evaluation from the subject matter experts through a web browser. This appendix contains the average values from each response opportunity.

B.1 Survey Results from Software Engineers

The survey for software engineers evaluated four targeted goals of the software security requirements: (1) whether they are clear, concise, and non-ambiguous, (2) whether they are implementable by software engineers, (3) whether they are incorporable into existing lifecycles, and (4) whether they are testable/verifiable. All of the goals had to be evaluated for each individual software security requirement, so the survey contained four statements for each requirement. The average response value for each software security requirement is provided here, and organized by computer resource category. Some of the software security requirements are listed twice because they were given to the software engineers twice - once for each time a constraint/assumption needed the requirement. The survey was conducted in this fashion so that the security experts responses on a per constraint/assumption basis could always be linked to software engineers responses.

B.1.1 Memory

(1) **Software Security Requirement:** When copying data into a buffer, the process shall ensure that the data being copied does not exceed the bounds of the buffer.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 8.25

- **Goal:** Implementable = 9

(2) **Software Security Requirement:** If data being copied into a buffer is expected to have a termination character to determine end of the data, then the process shall ensure that the termination character is present in the data, and ensure that the data being copied does not exceed the bounds of the buffer.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 8
- **Goal:** Implementable = 8.25

(3) **Software Security Requirement:** The process must not contain code that either directly or indirectly causes the instruction pointer to load with an address outside of instruction space.

- **Goal:** Concise = 8.5
- **Goal:** Testable/Verfiable = 6.75
- **Goal:** Incorporable into Lifecycles = 7.25
- **Goal:** Implementable = 6

(4) **Software Security Requirement:** The process shall ensure the value of an environment variable is in expected format before use.

- **Goal:** Concise = 9.5
- **Goal:** Testable/Verfiable = 9.5
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(5) **Software Security Requirement:** If an environment variable used by the process is expected to have a set of values, the process shall ensure the value of the environment variable is one of those expected values before use.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 9.5
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 9.5

(6) **Software Security Requirement:** Before invoking another process for execution, the invoking process shall clear all environment variables, and set environment variables required for execution with trusted values for the process being invoked.

- **Goal:** Concise = 9.25

- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 9.25

(7) **Software Security Requirement:** After a process makes a request for memory, the process shall check to see that the memory was properly allocated, and only use memory that has been successfully allocated.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 8.5
- **Goal:** Incorporable into Lifecycles = 8.75
- **Goal:** Implementable = 8.75

(8) **Software Security Requirement:** The process shall store sensitive data in memory only when necessary, and use a subroutine provided by the operating system to store the sensitive data in an encrypted format when the sensitive data is not in active use.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 8.25
- **Goal:** Implementable = 7.75

(9) **Software Security Requirement:** The process shall not reallocate memory containing sensitive data.

- **Goal:** Concise = 9.75
- **Goal:** Testable/Verfiable = 8.75
- **Goal:** Incorporable into Lifecycles = 8.75
- **Goal:** Implementable = 7.5

(10) **Software Security Requirement:** If a process holds sensitive data in memory, the process shall write zeros to the entire block of memory before releasing the memory to the operating system. The process shall not use a built in subroutine for writing the zeros.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 9.75
- **Goal:** Incorporable into Lifecycles = 9.25
- **Goal:** Implementable = 8.25

(11) **Software Security Requirement:** After a pointer to a memory location on the heap is deallocated, the process shall set that pointer's value to NULL.

- **Goal:** Concise = 9.5
- **Goal:** Testable/Verfiable = 9.75
- **Goal:** Incorporable into Lifecycles = 9.5
- **Goal:** Implementable = 9.5

(12) **Software Security Requirement:** Before dereferencing a pointer, the process shall ensure that the pointer's value is not set to NULL.

- **Goal:** Concise = 9.5
- **Goal:** Testable/Verfiable = 9.5
- **Goal:** Incorporable into Lifecycles = 9.75
- **Goal:** Implementable = 9.25

(13) **Software Security Requirement:** The process shall only set a pointer's value to an address located on the stack, if the lifetime of the stack variable will end after the lifetime of the pointer.

- **Goal:** Concise = 8.25
- **Goal:** Testable/Verfiable = 7.5
- **Goal:** Incorporable into Lifecycles = 7.5
- **Goal:** Implementable = 6.75

(14) **Software Security Requirement:** The process shall never make a request to the operating system for zero bytes of memory.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(15) **Software Security Requirement:** The process shall ensure that all addresses placed in a pointer variable are to memory locations other than its own.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 9

(16) **Software Security Requirement:** The process shall ensure the value of string input is in expected format before use.

- **Goal:** Concise = 9.5

- **Goal:** Testable/Verfiable = 9.5
- **Goal:** Incorporable into Lifecycles = 9.5
- **Goal:** Implementable = 9.5

(17) **Software Security Requirement:** If an input has an expected set of values, the process shall ensure the value of the input is one of those expected values before use.

- **Goal:** Concise = 9.5
- **Goal:** Testable/Verfiable = 9.5
- **Goal:** Incorporable into Lifecycles = 9.5
- **Goal:** Implementable = 9.5

(18) **Software Security Requirement:** Before an addition operation where both operands are positive, the process shall ensure that subtracting the left operand from the largest possible value is greater than or equal to the right operand. When the previous condition is not met, the process shall not perform the addition operation, and block any corresponding input from further use.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 9.5
- **Goal:** Implementable = 9.25

(19) **Software Security Requirement:** Before an addition operation where both operands are negative, the process shall ensure that subtracting the right operand from the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the addition operation, and block any corresponding input from further use.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 9.5
- **Goal:** Implementable = 9.25

(20) **Software Security Requirement:** Before a subtraction operation with unsigned numbers, the process shall ensure that the left operand is greater than the right operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 9.5
- **Goal:** Incorporable into Lifecycles = 9.5

- **Goal:** Implementable = 9.5

(21) **Software Security Requirement:** Before a subtraction operation where the left operand is nonnegative and the right operand is negative, the process shall ensure that adding the right operand to the largest possible value is greater than or equal to the left operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.

- **Goal:** Concise = 8.5
- **Goal:** Testable/Verfiable = 9.25
- **Goal:** Incorporable into Lifecycles = 9
- **Goal:** Implementable = 8.75

(22) **Software Security Requirement:** Before a subtraction operation where the left operand is negative and the right operand is positive, the process shall ensure that adding the right operand to the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.

- **Goal:** Concise = 8.5
- **Goal:** Testable/Verfiable = 9.25
- **Goal:** Incorporable into Lifecycles = 9
- **Goal:** Implementable = 8.75

(23) **Software Security Requirement:** Before a multiplication operation where both operands have the same sign, the process shall ensure that dividing the right operand by the largest possible value is greater than or equal to the left operand. When the previous condition is not met, the process shall not perform the multiplication operation, and block any corresponding input from further use.

- **Goal:** Concise = 8.25
- **Goal:** Testable/Verfiable = 9.25
- **Goal:** Incorporable into Lifecycles = 9
- **Goal:** Implementable = 8.75

(24) **Software Security Requirement:** Before a multiplication operation where operands have different signs, the process shall ensure that dividing the right operand by the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the multiplication operation, and block any corresponding input from further use.

- **Goal:** Concise = 8.25

- **Goal:** Testable/Verfiable = 9.25
- **Goal:** Incorporable into Lifecycles = 9
- **Goal:** Implementable = 8.75

(25) **Software Security Requirement:** The process shall ensure that a division operation never contains the largest negative value in the numerator, with a -1 in the denominator. When the previous condition is not met, the process shall not perform the division operation, and block any corresponding input from use.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 9.75
- **Goal:** Incorporable into Lifecycles = 9.25
- **Goal:** Implementable = 9.5

(26) **Software Security Requirement:** Before assigning a new value to a variable used as an index to a buffer, the process shall ensure the new value is within the buffers bounds.

- **Goal:** Concise = 9.5
- **Goal:** Testable/Verfiable = 9.75
- **Goal:** Incorporable into Lifecycles = 9.75
- **Goal:** Implementable = 9.5

(27) **Software Security Requirement:** Before assigning a new value to a variable used to hold the length or quantity of an object, the process shall ensure that the new value is nonnegative.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(28) **Software Security Requirement:** When copying data into a buffer, the process shall ensure that the data being copied does not exceed the bounds of the buffer.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(29) **Software Security Requirement:** If data being copied into a buffer is expected to have a termination character to determine end of the data, then the process shall ensure that the termination character is present in the data, and ensure that the data being copied does not exceed the bounds of the buffer.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(30) **Software Security Requirement:** The process shall not store sensitive data in static memory.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 7.5
- **Goal:** Implementable = 6.25

B.1.2 Networking

(1) **Software Security Requirement:** When receiving encrypted data over a network, the process shall decrypt and authenticate all data before its use, and reject data that does not authenticate or decrypt properly.

- **Goal:** Concise = 9.5
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 7.25

(2) **Software Security Requirement:** When receiving unencrypted data over a network from another host, the process shall authenticate all data before its use, and reject data that does not authenticate properly.

- **Goal:** Concise = 8.75
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 8.5
- **Goal:** Implementable = 6.5

(3) **Software Security Requirement:** The process shall authenticate the remote host to verify its identity before sending or accepting any data from that host.

- **Goal:** Concise = 9.5

- **Goal:** Testable/Verfiable = 9.5
- **Goal:** Incorporable into Lifecycles = 9.5
- **Goal:** Implementable = 7.25

(4) **Software Security Requirement:** The process shall ensure data received from a remote host is in expected format before use.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(5) **Software Security Requirement:** The process shall only send data in the expected format.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(6) **Software Security Requirement:** If a process is sending sensitive data to another host on a network, the process shall encrypt and provide authentication for the data.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 9.25
- **Goal:** Implementable = 6

(7) **Software Security Requirement:** When sending data over a network to another host, the process shall provide authentication for the data.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 9
- **Goal:** Implementable = 6.75

(8) **Software Security Requirement:** The process shall check to see if the desired port is available for listening before waiting for connections on that port.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10

- **Goal:** Implementable = 9.5

(9) **Software Security Requirement:** The process shall check to see if a connection to a remote host was accepted before sending data to that host.

- **Goal:** Concise = 8
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 8.75
- **Goal:** Implementable = 10

(10) **Software Security Requirement:** The process shall send all numerical data over a network in network byte order.

- **Goal:** Concise = 7.5
- **Goal:** Testable/Verfiable = 9.75
- **Goal:** Incorporable into Lifecycles = 9.75
- **Goal:** Implementable = 8.5

(11) **Software Security Requirement:** The process shall expect all numerical data received over a network to be in network byte order.

- **Goal:** Concise = 6.75
- **Goal:** Testable/Verfiable = 9.25
- **Goal:** Incorporable into Lifecycles = 9.75
- **Goal:** Implementable = 8.5

B.1.3 Filesystem

(1) **Software Security Requirement:** When creating a new file/directory, the process shall set the access permissions so the fewest number of users possible have access. The process shall set the access permissions using the file descriptor and not the filename.

- **Goal:** Concise = 8.25
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 9.75
- **Goal:** Implementable = 7.5

(2) **Software Security Requirement:** Before using a file/directory, the process shall check the ownership and the access permissions of the file/directory, and only use files/directories with expected ownership and access permissions.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 9.25
- **Goal:** Implementable = 8.25

(3) **Software Security Requirement:** When creating a file, the process shall ensure the path and name are unique.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(4) **Software Security Requirement:** If the process does not need to open files/directories through links, the process shall use a file open subroutine that blocks the opening of files/directories through links.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 7.25

(5) **Software Security Requirement:** If the process needs to open files/directories through links, the process shall first check the access permissions of the link itself. Then, the process shall open the file, and use the file descriptor to check the access permissions of the file. If the access permissions of the file and link do not match, the process shall not use the file.

- **Goal:** Concise = 8.75
- **Goal:** Testable/Verfiable = 9.25
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 9.75

(6) **Software Security Requirement:** The process shall ensure that data contained in a file is in expected format before use.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(7) **Software Security Requirement:** If a file used by the process is expected to have a set of data, the process shall ensure that the expected data is in that file before use.

- **Goal:** Concise = 9.5
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(8) **Software Security Requirement:** After opening a file, but before reading or writing to that file, the process shall request the operating system lock the file, using the file descriptor returned by the file open operation.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 9.25

(9) **Software Security Requirement:** Before closing a file, but after all reading and writing operations have been completed on that file, the process shall request the operating system unlock the file, using the file descriptor given to the previous lock file operation.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 9.25

(10) **Software Security Requirement:** If writing sensitive data to a file, the process shall encrypt and provide authentication for the data.

- **Goal:** Concise = 8.5
- **Goal:** Testable/Verfiable = 9.5
- **Goal:** Incorporable into Lifecycles = 9.5
- **Goal:** Implementable = 6.25

(11) **Software Security Requirement:** The process shall provide authentication for unencrypted data in files.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 9.75
- **Goal:** Incorporable into Lifecycles = 9.5

- **Goal:** Implementable = 6.5

(12) **Software Security Requirement:** The process shall only use data from files that have verified authenticity.

- **Goal:** Concise = 9.75
- **Goal:** Testable/Verfiable = 9.25
- **Goal:** Incorporable into Lifecycles = 9.25
- **Goal:** Implementable = 6.5

(13) **Software Security Requirement:** After deleting a file, if the data is too sensitive to be left on disk even in an encrypted format, the process shall write zeros to the entire hard-drive partition that contained the file. The process shall take into consideration that the procedure could also remove the operating system from disk.

- **Goal:** Concise = 9.5
- **Goal:** Testable/Verfiable = 8.75
- **Goal:** Incorporable into Lifecycles = 8
- **Goal:** Implementable = 5.75

(14) **Software Security Requirement:** After attempting a write to a file, the process shall ensure that the write was successful.

- **Goal:** Concise = 10
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 10

(15) **Software Security Requirement:** If writing sensitive data to a file, the process shall encrypt and provide authentication for the data.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 10
- **Goal:** Incorporable into Lifecycles = 10
- **Goal:** Implementable = 6.75

(16) **Software Security Requirement:** The process shall provide authentication for unencrypted data in files.

- **Goal:** Concise = 8.5
- **Goal:** Testable/Verfiable = 9.5
- **Goal:** Incorporable into Lifecycles = 9.75

- **Goal:** Implementable = 6.75

(17) **Software Security Requirement:** The process shall only use data from files that have verified authenticity.

- **Goal:** Concise = 9.25
- **Goal:** Testable/Verfiable = 9.25
- **Goal:** Incorporable into Lifecycles = 9.25
- **Goal:** Implementable = 7.25

B.1.4 Randomness

(1) **Software Security Requirement:** The process shall use a random number generation algorithm that is currently recommended by NIST.

- **Goal:** Concise = 7.6667
- **Goal:** Testable/Verfiable = 6.6667
- **Goal:** Incorporable into Lifecycles = 8.6667
- **Goal:** Implementable = 6

(2) **Software Security Requirement:** The process shall seed the cryptographically secure PRNG before any values are generated.

- **Goal:** Concise = 7.3333
- **Goal:** Testable/Verfiable = 7.6667
- **Goal:** Incorporable into Lifecycles = 8
- **Goal:** Implementable = 5.6667

(3) **Software Security Requirement:** The process shall reseed the cryptographically secure PRNG when the NIST defined maximum number of generated values per seed for that algorithm has been reached.

- **Goal:** Concise = 8
- **Goal:** Testable/Verfiable = 7.6667
- **Goal:** Incorporable into Lifecycles = 8.3333
- **Goal:** Implementable = 6

(4) **Software Security Requirement:** The process shall limit the access to the the internal state of the cryptographically secure PRNG to the fewest users possible.

- **Goal:** Concise = 6.6667
- **Goal:** Testable/Verfiable = 6

- **Goal:** Incorporable into Lifecycles = 9
- **Goal:** Implementable = 5.6667

(5) **Software Security Requirement:** The process shall use a process recommended by NIST to seed the cryptographically secure PRNG in use, and discard the seed value after seeding.

- **Goal:** Concise = 7.3333
- **Goal:** Testable/Verfiable = 8.6667
- **Goal:** Incorporable into Lifecycles = 9.3333
- **Goal:** Implementable = 6.3333

(6) **Software Security Requirement:** The process shall provide the seed with enough entropy to satisfy NIST requirements for the cryptographically secure PRNG in use.

- **Goal:** Concise = 6.3333
- **Goal:** Testable/Verfiable = 4.3333
- **Goal:** Incorporable into Lifecycles = 7.6667
- **Goal:** Implementable = 2.6667

(7) **Software Security Requirement:** The process shall use entropic data to seed a cryptographically secure PRNG, and provide cryptographic algorithms or protocols that require unpredictable data values generated from the cryptographically secure PRNG.

- **Goal:** Concise = 5.6667
- **Goal:** Testable/Verfiable = 7.6667
- **Goal:** Incorporable into Lifecycles = 8.3333
- **Goal:** Implementable = 2.6667

(8) **Software Security Requirement:** The process shall estimate 1 bit of entropy for each character pressed on the keyboard, but only if the character is different than the previous character.

- **Goal:** Concise = 4.6667
- **Goal:** Testable/Verfiable = 6.6667
- **Goal:** Incorporable into Lifecycles = 7.3333
- **Goal:** Implementable = 6

(9) **Software Security Requirement:** The process shall estimate log2(time) - 1 bits of entropy for the timing between local keyboard events, starting with the third event.

- **Goal:** Concise = 4
- **Goal:** Testable/Verfiable = 6
- **Goal:** Incorporable into Lifecycles = 5.6667
- **Goal:** Implementable = 4.3333

(10) **Software Security Requirement:** If mouse events from a user account cannot be detected from another user account, the process shall estimate 1 bit of entropy per local mouse event.

- **Goal:** Concise = 4.3333
- **Goal:** Testable/Verfiable = 5.6667
- **Goal:** Incorporable into Lifecycles = 5
- **Goal:** Implementable = 4

(11) **Software Security Requirement:** If a clock resolution of milliseconds or greater is available, the process shall count 0.5 bits of entropy for the timing between incoming packets, starting with with the third event.

- **Goal:** Concise = 6.6667
- **Goal:** Testable/Verfiable = 6.6667
- **Goal:** Incorporable into Lifecycles = 7
- **Goal:** Implementable = 5.6667

(12) **Software Security Requirement:** For entropy estimates that are not from keyboard events, network packet timings, or mouse events, the process shall divide the estimate of entropy from a data set by 8.

- **Goal:** Concise = 6
- **Goal:** Testable/Verfiable = 8.6667
- **Goal:** Incorporable into Lifecycles = 7.6667
- **Goal:** Implementable = 5.3333

(13) **Software Security Requirement:** If a user does not have to remember a password/key required for a cryptographic algorithm or protocol, the process shall use a cryptographically secure random data generator to generate the password/key.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 9.3333
- **Goal:** Incorporable into Lifecycles = 9
- **Goal:** Implementable = 7

(14) **Software Security Requirement:** If the user is required to remember a password for a cryptographic algorithm or protocol, the process shall use a random password generator recommended by NIST to provide passwords for users.

- **Goal:** Concise = 9.3333
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 7.6667
- **Goal:** Implementable = 7

(15) **Software Security Requirement:** If a user generated password will be used in a cryptographic algorithm or protocol, the process shall use a password-key derivation function described in PKCS #5 to create a cryptographic key from a user supplied password.

- **Goal:** Concise = 7.3333
- **Goal:** Testable/Verfiable = 8
- **Goal:** Incorporable into Lifecycles = 8
- **Goal:** Implementable = 5.3333

(16) **Software Security Requirement:** The process shall use a random number generation algorithm that is currently recommended by NIST.

- **Goal:** Concise = 8.6667
- **Goal:** Testable/Verfiable = 6.3333
- **Goal:** Incorporable into Lifecycles = 8
- **Goal:** Implementable = 6.3333

(17) **Software Security Requirement:** The process shall use a random number generation algorithm that is currently recommended by NIST.

- **Goal:** Concise = 8.6667
- **Goal:** Testable/Verfiable = 6.3333
- **Goal:** Incorporable into Lifecycles = 8
- **Goal:** Implementable = 6.6667

B.1.5 Cryptographic Algorithms and Protocols

(1) **Software Security Requirement:** The process shall use a random number generation algorithm that is currently recommended by NIST.

- **Goal:** Concise = 8.3333
- **Goal:** Testable/Verfiable = 6.6667

- **Goal:** Incorporable into Lifecycles = 8.6667
- **Goal:** Implementable = 6.6667

(2) **Software Security Requirement:** The process shall establish secret keys between two parties using a key establishment scheme that is currently recommended by NIST.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 9.3333
- **Goal:** Incorporable into Lifecycles = 9.3333
- **Goal:** Implementable = 6.3333

(3) **Software Security Requirement:** The process shall use the current NIST recommendation for key length for the cryptographic algorithm or protocol in use.

- **Goal:** Concise = 9.3333
- **Goal:** Testable/Verfiable = 9.3333
- **Goal:** Incorporable into Lifecycles = 9.3333
- **Goal:** Implementable = 8

(4) **Software Security Requirement:** The process shall use a hash algorithm that NIST currently recommends.

- **Goal:** Concise = 8.6667
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 9.3333
- **Goal:** Implementable = 5

(5) **Software Security Requirement:** Before giving data to a hash function, the process shall generate a random string of data whose length is equal to the length of the hash function output, then pad the end of the random string with zeros to equal the length of the internal block size of the hash function. The process shall concatenate this newly generated string before and after the data that was destined for the hash function, then give this large string concatenation to the hash function. The process shall then give the result of the hash to the hash function a second time, and discard the result of the first hash, only using the result of the second hash. The process shall store the random string with the hash result so the process can be duplicated with the same random string for any required verification.

- **Goal:** Concise = 7
- **Goal:** Testable/Verfiable = 8.3333
- **Goal:** Incorporable into Lifecycles = 8.3333

- **Goal:** Implementable = 4.6667

(6) **Software Security Requirement:** If non-encrypted data needs to be verified for integrity and authenticity but not verified as to what party generated the data, the process shall use a message authentication algorithm that is currently recommended by NIST. The process shall keep the key used for authentication a secret.

- **Goal:** Concise = 7.3333
- **Goal:** Testable/Verfiable = 8
- **Goal:** Incorporable into Lifecycles = 8.3333
- **Goal:** Implementable = 5

(7) **Software Security Requirement:** If encrypted data needs to be verified for integrity and authenticity but not verified as to what party generated the data, the process shall use a block cipher mode when encrypting the data that supports authenticated encryption. The process shall keep the key used for authentication a secret.

- **Goal:** Concise = 7.3333
- **Goal:** Testable/Verfiable = 8
- **Goal:** Incorporable into Lifecycles = 8.3333
- **Goal:** Implementable = 4.6667

(8) **Software Security Requirement:** If non-repudiation is required for encrypted or non-encrypted data to prove that the data came from only one source, the process shall sign the data with a signature algorithm currently recommended by NIST. The process shall keep any private keys used by the signature algorithm a secret.

- **Goal:** Concise = 8.3333
- **Goal:** Testable/Verfiable = 8.3333
- **Goal:** Incorporable into Lifecycles = 8.3333
- **Goal:** Implementable = 5.3333

(9) **Software Security Requirement:** If authentication of encrypted or unencrypted data is being used by two remote hosts, the sending process shall concatenate an increasing number value to the authentication key before generating an authentication value. The sending process shall store the number value with the authentication value. The receiving process shall use this number during authentication, and ensure this number value increases with every authentication check. The process shall ensure the data space for the number value is large enough to prevent repeat numbers.

- **Goal:** Concise = 6.6667
- **Goal:** Testable/Verfiable = 7.6667

- **Goal:** Incorporable into Lifecycles = 7
- **Goal:** Implementable = 5.3333

(10) **Software Security Requirement:** The process shall provide an unpredictable key to a stream cipher for keystream generation.

- **Goal:** Concise = 7
- **Goal:** Testable/Verfiable = 5.6667
- **Goal:** Incorporable into Lifecycles = 8.3333
- **Goal:** Implementable = 4

(11) **Software Security Requirement:** The process shall never provide a duplicate key to a stream cipher for keystream generation.

- **Goal:** Concise = 9.3333
- **Goal:** Testable/Verfiable = 5
- **Goal:** Incorporable into Lifecycles = 6.3333
- **Goal:** Implementable = 4.6667

(12) **Software Security Requirement:** The process shall encrypt sensitive data using a block cipher that is currently recommended by NIST, using a block cipher mode currently recommended by NIST. The process shall keep the encryption key a secret.

- **Goal:** Concise = 9.3333
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 8.3333
- **Goal:** Implementable = 4.6667

(13) **Software Security Requirement:** The process shall follow NIST guidelines for key management, to validate any key reported by another machine before use.

- **Goal:** Concise = 8.3333
- **Goal:** Testable/Verfiable = 7.6667
- **Goal:** Incorporable into Lifecycles = 8.3333
- **Goal:** Implementable = 5

(14) **Software Security Requirement:** The process shall designate a trusted third party for key verification.

- **Goal:** Concise = 9
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 9

- **Goal:** Implementable = 5.6667

(15) **Software Security Requirement:** The process shall encrypt sensitive data using a block cipher that is currently recommended by NIST, using a block cipher mode currently recommended by NIST. The process shall keep the encryption key a secret.

- **Goal:** Concise = 9.3333
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 8.3333
- **Goal:** Implementable = 4.6667

(16) **Software Security Requirement:** If storage is required for a password/key for later retrieval, the process shall encrypt the password/key with a second password/key, immediately discarding the original password/key. The process shall not store the second password/key with the encrypted password/key.

- **Goal:** Concise = 7
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 9.3333
- **Goal:** Implementable = 6.6667

(17) **Software Security Requirement:** If storage is required for a password/key for later validation, the process shall give the string to a secure hash procedure. The process shall discard the password after providing it to the secure hashing procedure, only storing the results of the secure hashing procedure.

- **Goal:** Concise = 7.3333
- **Goal:** Testable/Verfiable = 9
- **Goal:** Incorporable into Lifecycles = 9.3333
- **Goal:** Implementable = 7.3333

B.2 Survey Results from Security Experts

The survey for security experts evaluated three targeted goals of the software security requirements: (1) whether they are clear, concise, and non-ambiguous, (2) whether they prevent the associated constraint/assumption from being violated, and (3) whether they prevent all software vulnerabilities. The first goal had to be evaluated for each individual software security requirement, the second goal had to be evaluated for each constraint/assumption, and the third goal had to be evaluated for each computer resource category. This lead to a corresponding statement for each software security requirement, constraint/assumption, and computer resource category. The average response value for each constraint/assumption, software security requirement, and resource category is provided here, and organized by computer resource category.

B.2.1 Memory

- **Goal:** Prevent Vulnerabilities = 6.3333

(1) **Constraint/Assumption:** Data accepted as input to a process and assigned to a buffer must occupy and modify only specific locations allocated in the buffer.

- **Goal:** Prevent Violation of Constraint/Assumption = 6
- **Software Security Requirement:** When copying data into a buffer, the process shall ensure that the data being copied does not exceed the bounds of the buffer.
  - **Goal:** Conciseness = 9.25
- **Software Security Requirement:** If data being copied into a buffer is expected to have a termination character to determine end of the data, then the process shall ensure that the termination character is present in the data, and ensure that the data being copied does not exceed the bounds of the buffer.
  - **Goal:** Conciseness = 9.25

(2) **Constraint/Assumption:** The process will not interpret data present on the dynamic memory as executable code.

- **Goal:** Prevent Violation of Constraint/Assumption = 4.6667
- **Software Security Requirement:** The process must not contain code that either directly or indirectly causes the instruction pointer to load with an address outside of instruction space.
  - **Goal:** Conciseness = 9

(3) **Constraint/Assumption:** Environment variables being used by the process have expected format and values.

- **Goal:** Prevent Violation of Constraint/Assumption = 8
- **Software Security Requirement:** The process shall ensure the value of an environment variable is in expected format before use.
  - **Goal:** Conciseness = 8.5
- **Software Security Requirement:** If an environment variable used by the process is expected to have a set of values, the process shall ensure the value of the environment variable is one of those expected values before use.
  - **Goal:** Conciseness = 8.5
- **Software Security Requirement:** Before invoking another process for execution, the invoking process shall clear all environment variables, and set environment variables required for execution with trusted values for the process being invoked.

    * **Goal:** Conciseness = 8.5

(4) **Constraint/Assumption:** The process will be provided with the dynamic memory that it requests.

- **Goal:** Prevent Violation of Constraint/Assumption = 7
- **Software Security Requirement:** After a process makes a request for memory, the process shall check to see that the memory was properly allocated, and only use memory that has been successfully allocated.
    * **Goal:** Conciseness = 9.5

(5) **Constraint/Assumption:** Data present on the dynamic memory cannot be observed while the process is in execution.

- **Goal:** Prevent Violation of Constraint/Assumption = 6.3333
- **Software Security Requirement:** The process shall store sensitive data in memory only when necessary, and use a subroutine provided by the operating system to store the sensitive data in an encrypted format when the sensitive data is not in active use.
    * **Goal:** Conciseness = 9.25
- **Software Security Requirement:** The process shall not reallocate memory containing sensitive data.
    * **Goal:** Conciseness = 9.25

(6) **Constraint/Assumption:** Data owned by the process and stored on the dynamic memory cannot be accessed after the process frees the memory.

- **Goal:** Prevent Violation of Constraint/Assumption = 10
- **Software Security Requirement:** If a process holds sensitive data in memory, the process shall write zeros to the entire block of memory before releasing the memory to the operating system. The process shall not use a built in subroutine for writing the zeros.
    * **Goal:** Conciseness = 9.25

(7) **Constraint/Assumption:** A pointer variable being used by the process references a legal memory location.

- **Goal:** Prevent Violation of Constraint/Assumption = 7
- **Software Security Requirement:** After a pointer to a memory location on the heap is deallocated, the process shall set that pointer's value to NULL.
    * **Goal:** Conciseness = 9
- **Software Security Requirement:** Before dereferencing a pointer, the process shall ensure that the pointer's value is not set to NULL.

    - **Goal:** Conciseness = 9
- **Software Security Requirement:** The process shall only set a pointer's value to an address located on the stack, if the lifetime of the stack variable will end after the lifetime of the pointer.
    - **Goal:** Conciseness = 9

(8) **Constraint/Assumption:** A memory pointer returned by the underlying operating system does not point to zero bytes of memory.

- **Goal:** Prevent Violation of Constraint/Assumption = 7
- **Software Security Requirement:** The process shall never make a request to the operating system for zero bytes of memory.
    - **Goal:** Conciseness = 9.25

(9) **Constraint/Assumption:** A pointer variable being used by the process cannot reference itself.

- **Goal:** Prevent Violation of Constraint/Assumption = 10
- **Software Security Requirement:** The process shall ensure that all addresses placed in a pointer variable are to memory locations other than its own.
    - **Goal:** Conciseness = 9.75

(10) **Constraint/Assumption:** Data accepted by the process must not be interpreted as a format string by the I/O routines.

- **Goal:** Prevent Violation of Constraint/Assumption = 8
- **Software Security Requirement:** The process shall ensure the value of string input is in expected format before use.
    - **Goal:** Conciseness = 10
- **Software Security Requirement:** If an input has an expected set of values, the process shall ensure the value of the input is one of those expected values before use.
    - **Goal:** Conciseness = 10

(11) **Constraint/Assumption:** The value of the integer/expression (signed AMP unsigned) accepted/calculated by the process cannot be greater (less) than the maximum (minimum) value that can be stored in the integer variable.

- **Goal:** Prevent Violation of Constraint/Assumption = 9.3333
- **Software Security Requirement:** Before an addition operation where both operands are positive, the process shall ensure that subtracting the left operand from the largest possible value is greater than or equal to the right operand. When the previous condition is not met, the process shall not perform the addition operation, and block any corresponding input from further use.

    * **Goal:** Conciseness = 9.5

- **Software Security Requirement:** Before an addition operation where both operands are negative, the process shall ensure that subtracting the right operand from the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the addition operation, and block any corresponding input from further use.

    * **Goal:** Conciseness = 9.5

- **Software Security Requirement:** Before a subtraction operation with unsigned numbers, the process shall ensure that the left operand is greater than the right operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.

    * **Goal:** Conciseness = 9.5

- **Software Security Requirement:** Before a subtraction operation where the left operand is nonnegative and the right operand is negative, the process shall ensure that adding the right operand to the largest possible value is greater than or equal to the left operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.

    * **Goal:** Conciseness = 9.5

- **Software Security Requirement:** Before a subtraction operation where the left operand is negative and the right operand is positive, the process shall ensure that adding the right operand to the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the subtraction operation, and block any corresponding input from further use.

    * **Goal:** Conciseness = 9.5

- **Software Security Requirement:** Before a multiplication operation where both operands have the same sign, the process shall ensure that dividing the right operand by the largest possible value is greater than or equal to the left operand. When the previous condition is not met, the process shall not perform the multiplication operation, and block any corresponding input from further use.

    * **Goal:** Conciseness = 9.5

- **Software Security Requirement:** Before a multiplication operation where operands have different signs, the process shall ensure that dividing the right operand by the smallest possible value is less than or equal to the left operand. When the previous condition is not met, the process shall not perform the multiplication operation, and block any corresponding input from further use.

    * **Goal:** Conciseness = 9.5

- **Software Security Requirement:** The process shall ensure that a division operation never contains the largest negative value in the numerator, with a -1 in the denominator. When the previous condition is not met, the process shall not perform the division operation, and block any corresponding input from use.
  - **Goal:** Conciseness = 9.5

(12) **Constraint/Assumption:** An integer variable/expression used by the process as the index to a buffer must only hold values that allow it access to the memory locations assigned to the buffer.

- **Goal:** Prevent Violation of Constraint/Assumption = 10
- **Software Security Requirement:** Before assigning a new value to a variable used as an index to a buffer, the process shall ensure the new value is within the buffers bounds.
  - **Goal:** Conciseness = 9.5

(13) **Constraint/Assumption:** An integer variable/expression used by the process to indicate length/quantity of any object must not hold negative values.

- **Goal:** Prevent Violation of Constraint/Assumption = 10
- **Software Security Requirement:** Before assigning a new value to a variable used to hold the length or quantity of an object, the process shall ensure that the new value is nonnegative.
  - **Goal:** Conciseness = 8.25

(14) **Constraint/Assumption:** Data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer on the static memory.

- **Goal:** Prevent Violation of Constraint/Assumption = 6.6667
- **Software Security Requirement:** When copying data into a buffer, the process shall ensure that the data being copied does not exceed the bounds of the buffer.
  - **Goal:** Conciseness = 10
- **Software Security Requirement:** If data being copied into a buffer is expected to have a termination character to determine end of the data, then the process shall ensure that the termination character is present in the data, and ensure that the data being copied does not exceed the bounds of the buffer.
  - **Goal:** Conciseness = 10

(15) **Constraint/Assumption:** Data held on the static memory cannot be observed while the process is in execution.

- **Goal:** Prevent Violation of Constraint/Assumption = 3.6667
- **Software Security Requirement:** The process shall not store sensitive data in static memory.
  - **Goal:** Conciseness = 10

B.2.2 Networking

- **Goal:** Prevent Vulnerabilities = 5

(1) **Constraint/Assumption:** The data received by the software process through network interface is neither read nor modified by anyone other than the intended recipient.

- **Goal:** Prevent Violation of Constraint/Assumption = 6.6667
- **Software Security Requirement:** When receiving encrypted data over a network, the process shall decrypt and authenticate all data before its use, and reject data that does not authenticate or decrypt properly.
  - **Goal:** Conciseness = 9.5
- **Software Security Requirement:** When receiving unencrypted data over a network from another host, the process shall authenticate all data before its use, and reject data that does not authenticate properly.
  - **Goal:** Conciseness = 9.5

(2) **Constraint/Assumption:** The data received by the software process through the network interface is from a legitimate client or peer or server and has expected format and length.

- **Goal:** Prevent Violation of Constraint/Assumption = 7
- **Software Security Requirement:** The process shall authenticate the remote host to verify its identity before sending or accepting any data from that host.
  - **Goal:** Conciseness = 9.5
- **Software Security Requirement:** The process shall ensure data received from a remote host is in expected format before use.
  - **Goal:** Conciseness = 9.5
- **Software Security Requirement:** The process shall only send data in the expected format.
  - **Goal:** Conciseness = 9.5

(3) **Constraint/Assumption:** The data sent by the software process via the network interface will not be read/modified before it reaches its destination.

- **Goal:** Prevent Violation of Constraint/Assumption = 7
- **Software Security Requirement:** If a process is sending sensitive data to another host on a network, the process shall encrypt and provide authentication for the data.
  - **Goal:** Conciseness = 9
- **Software Security Requirement:** When sending data over a network to another host, the process shall provide authentication for the data.
  - **Goal:** Conciseness = 9

(4) **Constraint/Assumption:** The software process will be able to utilize the network interface to send and receive data

- **Goal:** Prevent Violation of Constraint/Assumption = 7
- **Software Security Requirement:** The process shall check to see if the desired port is available for listening before waiting for connections on that port.
  - **Goal:** Conciseness = 9
- **Software Security Requirement:** The process shall check to see if a connection to a remote host was accepted before sending data to that host.
  - **Goal:** Conciseness = 9

(5) **Constraint/Assumption:** The byte order of numerical data accepted from the network interface is the same as that of the host machine

- **Goal:** Prevent Violation of Constraint/Assumption = 7
- **Software Security Requirement:** The process shall send all numerical data over a network in network byte order.
  - **Goal:** Conciseness = 9
- **Software Security Requirement:** The process shall expect all numerical data received over a network to be in network byte order.
  - **Goal:** Conciseness = 9

B.2.3 Filesystem

- **Goal:** Prevent Vulnerabilities = 5.6667

(1) **Constraint/Assumption:** Access permissions assigned to newly created files/directories are such that only the required principals have access to them.

- **Goal:** Prevent Violation of Constraint/Assumption = 6.3333

- **Software Security Requirement:** When creating a new file/directory, the process shall set the access permissions so the fewest number of users possible have access. The process shall set the access permissions using the file descriptor and not the filename.
  - **Goal:** Conciseness = 8.5

(2) **Constraint/Assumption:** Access permissions of the files/directories being used by the process are such that only the required principals have access to them.

- **Goal:** Prevent Violation of Constraint/Assumption = 6.6667
- **Software Security Requirement:** Before using a file/directory, the process shall check the ownership and the access permissions of the file/directory, and only use files/directories with expected ownership and access permissions.
  - **Goal:** Conciseness = 8.5

(3) **Constraint/Assumption:** A file being created by the process does not have the same name as an already existing file.

- **Goal:** Prevent Violation of Constraint/Assumption = 9.6667
- **Software Security Requirement:** When creating a file, the process shall ensure the path and name are unique.
  - **Goal:** Conciseness = 8.25

(4) **Constraint/Assumption:** A filename (including path) being used by the process is not a link that points to another file for which the user executing the process does not have the required access permissions.

- **Goal:** Prevent Violation of Constraint/Assumption = 7.6667
- **Software Security Requirement:** If the process does not need to open files/directories through links, the process shall use a file open subroutine that blocks the opening of files/directories through links.
  - **Goal:** Conciseness = 8.25
- **Software Security Requirement:** If the process needs to open files/directories through links, the process shall first check the access permissions of the link itself. Then, the process shall open the file, and use the file descriptor to check the access permissions of the file. If the access permissions of the file and link do not match, the process shall not use the file.
  - **Goal:** Conciseness = 8.25

(5) **Constraint/Assumption:** A file created/populated by a principal other than the process, and then being used by the process, will have expected format and data.

- **Goal:** Prevent Violation of Constraint/Assumption = 9.3333

- **Software Security Requirement:** The process shall ensure that data contained in a file is in expected format before use.
  - **Goal:** Conciseness = 9.25
- **Software Security Requirement:** If a file used by the process is expected to have a set of data, the process shall ensure that the expected data is in that file before use.
  - **Goal:** Conciseness = 9.25

(6) **Constraint/Assumption:** A file being used by a process cannot be observed/-modified/replaced by any other process while the initial process is in execution.

- **Goal:** Prevent Violation of Constraint/Assumption = 8
- **Software Security Requirement:** After opening a file, but before reading or writing to that file, the process shall request the operating system lock the file, using the file descriptor returned by the file open operation.
  - **Goal:** Conciseness = 9.5
- **Software Security Requirement:** Before closing a file, but after all reading and writing operations have been completed on that file, the process shall request the operating system unlock the file, using the file descriptor given to the previous lock file operation.
  - **Goal:** Conciseness = 9.5

(7) **Constraint/Assumption:** A file/directory being used by the process and stored on the filesystem (information used by the process over multiple runs) cannot be observed/modified/replaced in-between these runs.

- **Goal:** Prevent Violation of Constraint/Assumption = 6.6667
- **Software Security Requirement:** If writing sensitive data to a file, the process shall encrypt and provide authentication for the data.
  - **Goal:** Conciseness = 10
- **Software Security Requirement:** The process shall provide authentication for unencrypted data in files.
  - **Goal:** Conciseness = 10
- **Software Security Requirement:** The process shall only use data from files that have verified authenticity.
  - **Goal:** Conciseness = 10

(8) **Constraint/Assumption:** Data held by files owned/used by the process must not be accessible after the process deletes them.

- **Goal:** Prevent Violation of Constraint/Assumption = 6.6667

- **Software Security Requirement:** After deleting a file, if the data is too sensitive to be left on disk even in an encrypted format, the process shall write zeros to the entire hard-drive partition that contained the file. The process shall take into consideration that the procedure could also remove the operating system from disk.
  - **Goal:** Conciseness = 10

(9) **Constraint/Assumption:** The process must be provided with the filesystem space that it requests.

- **Goal:** Prevent Violation of Constraint/Assumption = 7
- **Software Security Requirement:** After attempting a write to a file, the process shall ensure that the write was successful.
  - **Goal:** Conciseness = 10

(10) **Constraint/Assumption:** A file having proprietary or obscure format cannot be understood or modified.

- **Goal:** Prevent Violation of Constraint/Assumption = 4
- **Software Security Requirement:** If writing sensitive data to a file, the process shall encrypt and provide authentication for the data.
  - **Goal:** Conciseness = 9
- **Software Security Requirement:** The process shall provide authentication for unencrypted data in files.
  - **Goal:** Conciseness = 9
- **Software Security Requirement:** The process shall only use data from files that have verified authenticity.
  - **Goal:** Conciseness = 9

B.2.4 Randomness

- **Goal:** Prevent Vulnerabilities = 8.6667

(1) **Constraint/Assumption:** The series of random data being produced by the PRNG is unpredictable (assuming unpredictable seed).

- **Goal:** Prevent Violation of Constraint/Assumption = 9.3333
- **Software Security Requirement:** The process shall use a random number generation algorithm that is currently recommended by NIST.
  - **Goal:** Conciseness = 9.5

- **Software Security Requirement:** The process shall seed the cryptographically secure PRNG before any values are generated.
  - **Goal:** Conciseness = 9.5
- **Software Security Requirement:** The process shall reseed the cryptographically secure PRNG when the NIST defined maximum number of generated values per seed for that algorithm has been reached.
  - **Goal:** Conciseness = 9.5
- **Software Security Requirement:** The process shall limit the access to the the internal state of the cryptographically secure PRNG to the fewest users possible.
  - **Goal:** Conciseness = 9.5

(2) **Constraint/Assumption:** The seed being used by the PRNG is unpredictable.

- **Goal:** Prevent Violation of Constraint/Assumption = 9.6667
- **Software Security Requirement:** The process shall use a process recommended by NIST to seed the cryptographically secure PRNG in use, and discard the seed value after seeding.
  - **Goal:** Conciseness = 8.75
- **Software Security Requirement:** The process shall provide the seed with enough entropy to satisfy NIST requirements for the cryptographically secure PRNG in use.
  - **Goal:** Conciseness = 8.75

(3) **Constraint/Assumption:** The process will have easy access to entropic data on a computer system.

- **Goal:** Prevent Violation of Constraint/Assumption = 8
- **Software Security Requirement:** The process shall use entropic data to seed a cryptographically secure PRNG, and provide cryptographic algorithms or protocols that require unpredictable data values generated from the cryptographically secure PRNG.
  - **Goal:** Conciseness = 9.5

(4) **Constraint/Assumption:** The process will be able to accurately estimate entropy of a data set.

- **Goal:** Prevent Violation of Constraint/Assumption = 6.3333
- **Software Security Requirement:** The process shall estimate 1 bit of entropy for each character pressed on the keyboard, but only if the character is different than the previous character.
  - **Goal:** Conciseness = 10

- **Software Security Requirement:** The process shall estimate log2(time) - 1 bits of entropy for the timing between local keyboard events, starting with the third event.
  - **Goal:** Conciseness = 10
- **Software Security Requirement:** If mouse events from a user account cannot be detected from another user account, the process shall estimate 1 bit of entropy per local mouse event.
  - **Goal:** Conciseness = 10
- **Software Security Requirement:** If a clock resolution of milliseconds or greater is available, the process shall count 0.5 bits of entropy for the timing between incoming packets, starting with with the third event.
  - **Goal:** Conciseness = 10
- **Software Security Requirement:** For entropy estimates that are not from keyboard events, network packet timings, or mouse events, the process shall divide the estimate of entropy from a data set by 8.
  - **Goal:** Conciseness = 10

(5) **Constraint/Assumption:** User selected passwords/keys will have a sufficient amount of entropy.

- **Goal:** Prevent Violation of Constraint/Assumption = 9.3333
- **Software Security Requirement:** If a user does not have to remember a password/key required for a cryptographic algorithm or protocol, the process shall use a cryptographically secure random data generator to generate the password/key.
  - **Goal:** Conciseness = 10
- **Software Security Requirement:** If the user is required to remember a password for a cryptographic algorithm or protocol, the process shall use a random password generator recommended by NIST to provide passwords for users.
  - **Goal:** Conciseness = 10
- **Software Security Requirement:** If a user generated password will be used in a cryptographic algorithm or protocol, the process shall use a password-key derivation function described in PKCS #5 to create a cryptographic key from a user supplied password.
  - **Goal:** Conciseness = 10

(6) **Constraint/Assumption:** If two different seeds are provided to the PRNG, it is computationally infeasible to produce the same series of data both times.

- **Goal:** Prevent Violation of Constraint/Assumption = 9.6667

- **Software Security Requirement:** The process shall use a random number generation algorithm that is currently recommended by NIST.
  - **Goal:** Conciseness = 10

(7) **Constraint/Assumption:** Given that the PRNG is continuously producing random data, it is computationally infeasible to produce the same sequence of random data after some time.

- **Goal:** Prevent Violation of Constraint/Assumption = 10
- **Software Security Requirement:** The process shall use a random number generation algorithm that is currently recommended by NIST.
  - **Goal:** Conciseness = 9

B.2.5 Cryptographic Algorithms and Protocols

- **Goal:** Prevent Vulnerabilities = 6.6667

(1) **Constraint/Assumption:** Random data being used by the cryptographic algorithm/protocol is unpredictable.

- **Goal:** Prevent Violation of Constraint/Assumption = 10
- **Software Security Requirement:** The process shall use a random number generation algorithm that is currently recommended by NIST.
  - **Goal:** Conciseness = 9
- **Software Security Requirement:** The process shall establish secret keys between two parties using a key establishment scheme that is currently recommended by NIST.
  - **Goal:** Conciseness = 9

(2) **Constraint/Assumption:** The length of the key being used by the cryptographic algorithm or protocol is sufficient.

- **Goal:** Prevent Violation of Constraint/Assumption = 10
- **Software Security Requirement:** The process shall use the current NIST recommendation for key length for the cryptographic algorithm or protocol in use.
  - **Goal:** Conciseness = 8

(3) **Constraint/Assumption:** The hashing algorithm will not produce same hash for two different inputs.

- **Goal:** Prevent Violation of Constraint/Assumption = 7

- **Software Security Requirement:** The process shall use a hash algorithm that NIST currently recommends.
  - **Goal:** Conciseness = 7.5
- **Software Security Requirement:** Before giving data to a hash function, the process shall generate a random string of data whose length is equal to the length of the hash function output, then pad the end of the random string with zeros to equal the length of the internal block size of the hash function. The process shall concatenate this newly generated string before and after the data that was destined for the hash function, then give this large string concatenation to the hash function. The process shall then give the result of the hash to the hash function a second time, and discard the result of the first hash, only using the result of the second hash. The process shall store the random string with the hash result so the process can be duplicated with the same random string for any required verification.
  - **Goal:** Conciseness = 7.5

(4) **Constraint/Assumption:** The process cannot use encryption to ensure data integrity or data authentication.

- **Goal:** Prevent Violation of Constraint/Assumption = 7
- **Software Security Requirement:** If non-encrypted data needs to be verified for integrity and authenticity but not verified as to what party generated the data, the process shall use a message authentication algorithm that is currently recommended by NIST. The process shall keep the key used for authentication a secret.
  - **Goal:** Conciseness = 6.75
- **Software Security Requirement:** If encrypted data needs to be verified for integrity and authenticity but not verified as to what party generated the data, the process shall use a block cipher mode when encrypting the data that supports authenticated encryption. The process shall keep the key used for authentication a secret.
  - **Goal:** Conciseness = 6.75
- **Software Security Requirement:** If non-repudiation is required for encrypted or non-encrypted data to prove that the data came from only one source, the process shall sign the data with a signature algorithm currently recommended by NIST. The process shall keep any private keys used by the signature algorithm a secret.
  - **Goal:** Conciseness = 6.75
- **Software Security Requirement:** If authentication of encrypted or unencrypted data is being used by two remote hosts, the sending process shall concatenate an increasing number value to the authentication key before generating

an authentication value. The sending process shall store the number value with the authentication value. The receiving process shall use this number during authentication, and ensure this number value increases with every authentication check. The process shall ensure the data space for the number value is large enough to prevent repeat numbers.

    * **Goal:** Conciseness = 6.75

(5) **Constraint/Assumption:** The process cannot use a key more than once for a stream cipher.

- **Goal:** Prevent Violation of Constraint/Assumption = 9.6667
- **Software Security Requirement:** The process shall provide an unpredictable key to a stream cipher for keystream generation.
    * **Goal:** Conciseness = 8.25
- **Software Security Requirement:** The process shall never provide a duplicate key to a stream cipher for keystream generation.
    * **Goal:** Conciseness = 8.25

(6) **Constraint/Assumption:** The process cannot use one time pads to encrypt large quantity of data.

- **Goal:** Prevent Violation of Constraint/Assumption = 8.3333
- **Software Security Requirement:** The process shall encrypt sensitive data using a block cipher that is currently recommended by NIST, using a block cipher mode currently recommended by NIST. The process shall keep the encryption key a secret.
    * **Goal:** Conciseness = 9.25

(7) **Constraint/Assumption:** The process cannot use keys that are self reported by a client or a server.

- **Goal:** Prevent Violation of Constraint/Assumption = 6.6667
- **Software Security Requirement:** The process shall follow NIST guidelines for key management, to validate any key reported by another machine before use.
    * **Goal:** Conciseness = 9.25
- **Software Security Requirement:** The process shall designate a trusted third party for key verification.
    * **Goal:** Conciseness = 9.25

(8) **Constraint/Assumption:** The process cannot use obfuscation instead of encryption to ensure confidentiality.

- **Goal:** Prevent Violation of Constraint/Assumption = 6.6667
- **Software Security Requirement:** The process shall encrypt sensitive data using a block cipher that is currently recommended by NIST, using a block cipher mode currently recommended by NIST. The process shall keep the encryption key a secret.
  - **Goal:** Conciseness = 9

(9) **Constraint/Assumption:** The process cannot store keys/passwords in clear text.

- **Goal:** Prevent Violation of Constraint/Assumption = 10
- **Software Security Requirement:** If storage is required for a password/key for later retrieval, the process shall encrypt the password/key with a second password/key, immediately discarding the original password/key. The process shall not store the second password/key with the encrypted password/key.
  - **Goal:** Conciseness = 8.75
- **Software Security Requirement:** If storage is required for a password/key for later validation, the process shall give the string to a secure hash procedure. The process shall discard the password after providing it to the secure hashing procedure, only storing the results of the secure hashing procedure.
  - **Goal:** Conciseness = 8.75

Appendix C

Contents of the SSRT Structure

The structure for the SSRT is provided in figure C.1. The letters in the figure correspond to the description usage statements for a branch, and the numbers correspond to a list of associated software security requirements required at that branch.

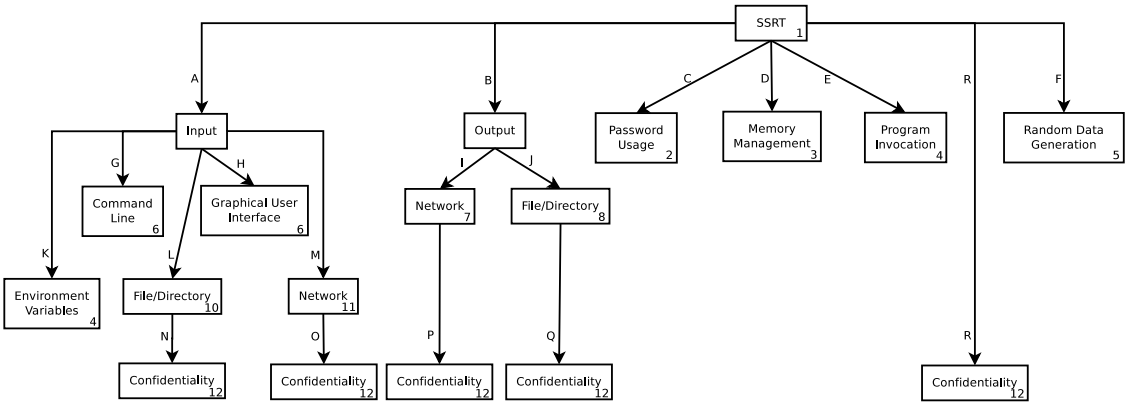


Figure C.1: The Structure of the SSRT

C.1 Branch Descriptions

Each bullet in this section contains a letter that corresponds to a branch in the SSRT, as shown in figure C.1. The usage description statements associated with each branch is outlined below.

- ***Description:*** **A** Security requirements that need to be included in a process that accepts any kind of input from users or any other process. The input could be obtained through a graphical interface, command line, filesystem read, networking, or environment variable.
- ***Description:*** **B** Security requirements that need to be included in a process that provides output to a network, or filesystem.

- ***Description:*** **C** Security requirements that need to be included in a process that accepts, collects, stores, or uses passwords.
- ***Description:*** **D** Security requirements that need to be included in a process that handles its own memory management. This includes systems that directly request, and return memory to the operating system.
- ***Description:*** **E** Security requirements that need to be included in a process that invokes (executes) another process.
- ***Description:*** **F** Security requirements that need to be included in a process that uses random data (numbers).
- ***Description:*** **G** Security requirements that need to be included in a process that accepts input from a command line.
- ***Description:*** **H** Security requirements that need to be included in a process that accepts input from a graphical interface.
- ***Description:*** **I** Security requirements that need to be included in a process that provides output to the network (internet).
- ***Description:*** **J** Security requirements that need to be included in a process that provides output to a file/directory.
- ***Description:*** **K** Security requirements that need to be included in a process that accepts input from environment variables.
- ***Description:*** **L** Security requirements that need to be included in a process that accepts input from a file/directory.
- ***Description:*** **M** Security requirements that need to be included in a process that accepts input from the network (internet).
- ***Description:*** **N** Security requirements that need to be included in a process that accepts sensitive (confidential) input from a file/directory. If sensitive input has not been defined, then all file/directory input is considered sensitive.
- ***Description:*** **O** Security requirements that need to be included in a process that accepts sensitive (confidential) input from the network (internet). If sensitive input has not been defined, then all network input is considered sensitive.
- ***Description:*** **P** Security requirements that need to be included in a process that provides sensitive (confidential) output to the network (internet). If sensitive output has not been defined, then all network output is considered sensitive.
- ***Description:*** **Q** Security requirements that need to be included in a process that provides sensitive (confidential) output to a file/directory. If sensitive output has not been defined, then all file/directory output is considered sensitive.

- ***Description:*** **R** Security requirements that need to be included in a process that contains sensitive (confidential) data in memory. If sensitive data has not been defined, then all data is considered sensitive.

C.2 Software Security Requirements Lists

Each bullet in this section contains a number that corresponds to a node in the SSRT, as shown in figure C.1. The software security requirements associated with each node are outlined below. While general requirements ultimately map to software security requirements, they are first mapped to constraints/assumptions. Since the list of software security requirement is longer, the constraints/assumptions are listed here for brevity, and represent the software security requirements. The prototype tool discussed in 5.1.3 uses the full software security requirements, since they are what would be of most use to a developer.

- ***Requirements List:*** **1**

  Dynamic Memory #1 Dynamic Memory #2 Dynamic Memory #5
  Dynamic Memory #6 Dynamic Memory #11 Dynamic Memory #12
  Dynamic Memory #13 Static Memory #1 Static Memory #2

- ***Requirements List:*** **2**

  Randomness Resources #1
  Cryptographic Resources and Algorithms #2
  Cryptographic Algorithms and Protocols #3
  Cryptographic Algorithms and Protocols #4
  Cryptographic Algorithms and Protocols #5
  Cryptographic Algorithms and Protocols #6
  Cryptographic Algorithms and Protocols #8
  Cryptographic Algorithms and Protocols #9

- ***Requirements List:*** **3**

  Dynamic Memory #4 Dynamic Memory #7
  Dynamic Memory #8 Dynamic Memory #9

- ***Requirements List:*** **4**

  Dynamic Memory #3

- ***Requirements List:*** **5**

  Randomness Resources #1 Randomness Resources #2
  Randomness Resources #3 Randomness Resources #4
  Randomness Resources #6 Randomness Resources #7

- ***Requirements List:*** **6**

  Dynamic Memory #10

- ***Requirements List:*** **7**

  Network Interface #2
  Network Interface #3
  Network Interface #4
  Network Interface #5
  Cryptographic Algorithms and Protocols #1
  Cryptographic Algorithms and Protocols #2
  Cryptographic Algorithms and Protocols #3
  Cryptographic Algorithms and Protocols #4
  Cryptographic Algorithms and Protocols #7
  Cryptographic Algorithms and Protocols #9
  Randomness Resources #1
  Randomness Resources #2
  Randomness Resources #3
  Randomness Resources #4
  Randomness Resources #5
  Randomness Resources #6
  Randomness Resources #7

- ***Requirements List:*** **8**

  Filesystem #4
  Filesystem #5
  Filesystem #6
  Filesystem #7
  Filesystem #10
  Cryptographic Algorithms and Protocols #1
  Cryptographic Algorithms and Protocols #2
  Cryptographic Algorithms and Protocols #3
  Cryptographic Algorithms and Protocols #4
  Cryptographic Algorithms and Protocols #7
  Cryptographic Algorithms and Protocols #9
  Randomness Resources #1
  Randomness Resources #2
  Randomness Resources #3
  Randomness Resources #4
  Randomness Resources #5
  Randomness Resources #6
  Randomness Resources #7

- ***Requirements List:*** **9**

  Dynamic Memory #3

- ***Requirements List:*** **10**

  Filesystem #4
  Filesystem #5
  Filesystem #6
  Filesystem #7
  Filesystem #8
  Cryptographic Algorithms and Protocols #1
  Cryptographic Algorithms and Protocols #2
  Cryptographic Algorithms and Protocols #3
  Cryptographic Algorithms and Protocols #4
  Cryptographic Algorithms and Protocols #7
  Cryptographic Algorithms and Protocols #9
  Randomness Resources #1
  Randomness Resources #2
  Randomness Resources #3
  Randomness Resources #4
  Randomness Resources #5
  Randomness Resources #5
  Randomness Resources #6
  Randomness Resources #7

- ***Requirements List:*** **12**

Network Interface #1	Randomness Resources #1
Network Interface #2	Randomness Resources #2
Network Interface #4	Randomness Resources #3
Network Interface #5	Randomness Resources #4
Cryptographic Algorithms and Protocols #1	Randomness Resources #5
Cryptographic Algorithms and Protocols #2	Randomness Resources #6
Cryptographic Algorithms and Protocols #3	Randomness Resources #7
Cryptographic Algorithms and Protocols #4	
Cryptographic Algorithms and Protocols #7	
Cryptographic Algorithms and Protocols #9	

- ***Requirements List:*** **12**

Cryptographic Algorithms and Protocols #1	Randomness Resources #1
Cryptographic Algorithms and Protocols #2	Randomness Resources #2
Cryptographic Algorithms and Protocols #3	Randomness Resources #3
Cryptographic Algorithms and Protocols #5	Randomness Resources #4
Cryptographic Algorithms and Protocols #6	Randomness Resources #5
Cryptographic Algorithms and Protocols #8	Randomness Resources #6
Cryptographic Algorithms and Protocols #9	Randomness Resources #7

Appendix D

Code for the Prototype SSRT Tool

All the code for the developed prototype SSRT tool is provided in this appendix. The prototype was written in Java, and the code was broken up into multiple files. The name of each file, and the code in the file is provided here.

Listing D.1: SSRTProgram.java

```
package edu.vt.cirt;

import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ComponentEvent;
import java.awt.event.ComponentListener;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.security.InvalidParameterException;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTextArea;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;


/**************************************************************************
 * Class for the main execution of the program. Constructor will create the
 * main window. Unrecoverable exceptions (other than GUI setup exceptions)
 * are shown via alert boxes to the user with information as to why the event
 * was unrecoverable.
 *
 * @author Lee M. Clagett II (lclagett@vt.edu)
 */
public class SSRTProgram extends JFrame
{
        // Where to read the XML files from. Default is current working directory
        final static private String[] defaultFileLocations = {
                "ssrt.xml", // Default SSRT XML location
```

```
        "requirements.xml", // Default Requirements XML location
        "help.xml"}; // Help XML file location

// Required by Java for serialization (not used)
private static final long serialVersionUID = 1L;

private SSRT m_requirementsTree;
private JSplitPane m_splitPane;
private JTextArea m_descriptionWindow;

private JButton m_gatherRequirementsButton;
private JButton m_helpButton;

private String m_helpFileText;


/**************************************************************************
 * Constructor( SSRT )
 *
 * Create a new SSRTProgram window. Sets up up all elements of the
 * window, and required event listeners for actions between elements.
 *
 * @param requirementsTree The copy of the SSRT to be used for this session
 * @exception NullPointerException if requirementsTree or helpFileText is null
 * @exception InvalidParameterException if helpFileText is blank string
 */
public SSRTProgram(SSRT requirementsTree, String helpFileText)
{
        super("Protoype SSRT Tool ~ Lee M Clagett II 2009");

        // *** ERROR CHECK PARAMETERS ***
        if(requirementsTree == null || helpFileText == null)
                throw new NullPointerException("DCSSRTProgram::Constructor cannot be
                     given a null argument.");

        if(helpFileText.equals(""))
                throw new InvalidParameterException("DCSSRTProgram::Constructor
                    cannot be given a blank string for the help file location.");
        // END ERROR CHECK PARAMETERS

        m_helpFileText = helpFileText;

        m_requirementsTree = requirementsTree;
        m_requirementsTree.addTreeSelectionListener(dcssrtListener);

        createWindowComponents();

        // Last necessary parameters before showing window
        addComponentListener(resizeListener);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);

        setDefaultWindowSize();
}


/**************************************************************************
 * Anonymous class to detect whenever the tree was clicked. Updates the
 * bottom window to show the usage description of the node that was
 * clicked.
 */
private TreeSelectionListener dcssrtListener = new TreeSelectionListener()
{
        public void valueChanged(TreeSelectionEvent e)
        {
```

```
                // *** ERROR CHECK PRE-CONDITIONS
                assert(m_requirementsTree != null);
                assert(m_descriptionWindow != null);
                // END ERROR CHECK PRE-CONDITIONS

                Object selected = m_requirementsTree.getLastSelectedPathComponent();

                // Make sure that it wasn't somehow DE-selected, or the root node
                if(selected == null || m_requirementsTree.getModel().getRoot() ==
                    selected)
                        return;

                m_descriptionWindow.setText(((Requirement)selected).getText());
        }
};

/*************************************************************************
 * Anonymous class to detect button presses in the main window. The only
 * two buttons in the main window are:
 * "Gather Software Security Requirements", and "Help".
 */
private ActionListener mainWindowButtonListener = new ActionListener()
{
        public void actionPerformed(ActionEvent event)
        {
                // *** ERROR CHECK PRE-CONDITIONS
                assert(m_requirementsTree != null);
                assert(m_helpFileText != null);
                assert(!m_helpFileText.equals(""));
                // END ERROR CHECK PRE-CONDITIONS

                if(event == null)
                        return;

                // Try to catch when a blank string is given to the requirements
                    window
                try
                {
                        if(event.getSource() == m_gatherRequirementsButton)
                                new SecurityRequirementsWindow(m_requirementsTree.
                                    getSecurityRequirementsText());

                        else if(event.getSource() == m_helpButton)
                                        new HelpWindow(m_helpFileText);
                }
                catch(InvalidParameterException exception)
                {
                        JOptionPane.showMessageDialog(m_requirementsTree, "An error
                            ocurred when trying to gather the requirements (no
                            requirements to give).", "Requirements Gathering Error",
                             JOptionPane.ERROR_MESSAGE);
                }
        }
};

/*************************************************************************
 * Anonymous class to detect window resizes. Needed to correctly set the
 * two pane divider, which gets screwed up during resizes
 */
private ComponentListener resizeListener = new ComponentListener()
{
        public void componentHidden(ComponentEvent e) {}

        public void componentMoved(ComponentEvent e) {}

        public void componentResized(ComponentEvent e)
```

```
        {
                // *** ERROR CHECK PRE-CONDITIONS
                assert(m_splitPane != null);
                // END ERROR CHECK PRE-CONDITIONS

                m_splitPane.setDividerLocation(0.85);
        }

        public void componentShown(ComponentEvent e) {}
};

/**************************************************************************
 * createWindowComponents()
 *
 * Helper method that contains the creation of all the window elements.
 */
private void createWindowComponents()
{
        // Create the Vertical Split Pane
        m_splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
        setContentPane(m_splitPane);

        // Set the DCSSRT in the Top Pane
        JPanel topPanel = new JPanel();
        topPanel.setLayout(new BoxLayout(topPanel, BoxLayout.Y_AXIS));
        JPanel topBottomPanel = new JPanel();
        topBottomPanel.setLayout(new BoxLayout(topBottomPanel, BoxLayout.X_AXIS));
        m_gatherRequirementsButton = new JButton("Gather Software Security
            Requirements");
        m_gatherRequirementsButton.addActionListener(mainWindowButtonListener);
        topBottomPanel.add(m_gatherRequirementsButton);
        topBottomPanel.add(Box.createHorizontalGlue());
        m_helpButton = new JButton("Help");
        m_helpButton.addActionListener(mainWindowButtonListener);
        topBottomPanel.add(m_helpButton);

        topPanel.add(topBottomPanel);
        topPanel.add(new JScrollPane(m_requirementsTree));

        m_splitPane.setLeftComponent(topPanel);

        // Create a text area, and set in the bottom pane
        m_descriptionWindow = new JTextArea();
        m_descriptionWindow.setEditable(false);
        m_descriptionWindow.setLineWrap(true);
        m_descriptionWindow.setWrapStyleWord(true);
        m_descriptionWindow.setText("Click on a node name for a better description.
            If the current requirements to your project match the description,
            select the checkmark, and review the children nodes in the same way.");
        m_splitPane.setRightComponent(new JScrollPane(m_descriptionWindow));
}

/**************************************************************************
 * setDefaultWindowSize()
 *
 * Helper method to set the window size based on the screen resolution.
 * The window is 2/3rds the screen size in the x direction, and 1/2 the
 * direction in the y direction. The window is automatically centered.
 */
private void setDefaultWindowSize()
{
        // Next line may throw an exception, let program fail if they are
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();

        // Window is 2/3rds the screen size in x direction, 1/2 the y direction
        int windowWidth = (int)((0.667 * screenSize.getWidth()));
```

```
        int windowHeight = (int)((0.5 * screenSize.getHeight()));
        setSize(windowWidth, windowHeight);

        // Window starts at 1/6th into the x screen and 1/4 the y screen
        // (1/4 and 1/2 the window size directions)
        setLocation((int)(windowWidth * 0.25), (int)(windowHeight * 0.5));
}

/**************************************************************************
 * printUsage()
 *
 * Helper method for printing the usage of the program from the
 * command line.
 */
private static void printUsage()
{
        System.out.print("Command line arguments: [SSRT XML File] [Requirements XML
            File] [Help XML File]");
}


/**************************************************************************
 * Main( String[] )
 *
 * Three optional command line arguments can be given, all of which are
 * file locations. They are in order: (1) the SSRT XML file,
 * (2) requirements XML file, and (3) help xml file.
 *
 * @param args Arguments from the command line, should be zero (no one will be used)
 * @throws UnsupportedLookAndFeelException If the Java cannot get the
     SystemLookAndFeel, or the Java look and feel. Crash if we have no look or feel
     (no GUI), we need GUI
 * @throws IllegalAccessException If some class in LookAndFeel isn't accessible for
     creation, let 'er crash we need GUI
 * @throws InstantiationException If an instance of the look and feel could not be
     created. Unrecoverable from this program's standpoint, let 'er crash
 * @throws ClassNotFoundException If the LookAndFeel class could not be found.
     Should not happen when getting system look ... famous last words
 */
public static void main(String[] args) throws ClassNotFoundException,
    InstantiationException, IllegalAccessException, UnsupportedLookAndFeelException
{
        // ** ERROR CHECK ARGUMENTS ***
        if(args.length > 3 || args == null)
        {
                printUsage();
                System.exit(-1);
        }
        // END ERROR CHECK ARGUMENTS

        // Copy default files, and update to a new location if one from command line
        String[] fileLocations = defaultFileLocations;
        for(int i = 0; i < args.length; i++)
                fileLocations[i] = args[i];

        try
        {
                // Set the UI stuff first, so it it fails, we don't even try to show
                     alert messages
                UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
                setDefaultLookAndFeelDecorated(true);

                SSRT tree = new SSRT(fileLocations[0], fileLocations[1]);
                HelpParser tempHelperParser = new HelpParser(fileLocations[2]);

                new SSRTProgram(tree, tempHelperParser.getHelpText());
```

```
                }
                catch (FileNotFoundException e) // When either of the input files cannot be
                    read ..
                {
                        JOptionPane.showMessageDialog(null, e.getMessage(), "File Input
                            Error", JOptionPane.ERROR_MESSAGE);
                        printUsage();
                }
                catch (IOException e) // Some sort of I/O problem during file reading ..
                {
                        JOptionPane.showMessageDialog(null, e.getMessage(), "General I/O
                            Exception", JOptionPane.ERROR_MESSAGE);
                }
                catch (SAXException e) // Bad XML format, or bad DCSRRT/Requirements
                    document format. Will tell user which.
                {
                        JOptionPane.showMessageDialog(null, e.getMessage(), "File Format
                            Error", JOptionPane.ERROR_MESSAGE);
                        e.printStackTrace();
                }
                catch (ParserConfigurationException e)
                {
                        JOptionPane.showMessageDialog(null, "An error ocurred preventing a
                            document parser from being created.\nThe program cannot operate
                            without the information from the parsed XML files.", "Document
                            Parser", JOptionPane.ERROR_MESSAGE);
                }
        }
}
```

Listing D.2: SecurityRequirementsWindow.java

```
package edu.vt.cirt;

import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.datatransfer.Clipboard;
import java.awt.datatransfer.StringSelection;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.security.InvalidParameterException;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;


/**
 * This shows the software security requirements in a new window, in a text
 * box. A button at the bottom copies all the requirements to the clipboard.
 *
 * @author Lee M. Clagett II (lclagett@vt.edu)
 */
public class SecurityRequirementsWindow extends JFrame
{
        // Required by Java for serialization (not used)
        private static final long serialVersionUID = 1L;

        String m_requirements;

        /**************************************************************************
         * Constructor( String )
         *
```

```
 * Creates a new security requirements window and dislays it above all
 * other windows. The string parameter is used in the textbox, and should
 * be software security requirements to display to the user.
 *
 * @param requirements Software security requirements to display in the window to
     the user
 */
public SecurityRequirementsWindow(String requirements)
{
        super("Prototype SSRT Tool ~ Software Security Requirements for Your Project
            ");

        // *** ERROR CHECK PARAMETERS
        if(requirements == null)
                throw new NullPointerException("Constructor cannot be given a null
                    argument");

        if(requirements.equals(""))
                throw new InvalidParameterException("Constructor cannot be given an
                    empty string");
        // END ERROR CHECK PARAMETERS

        m_requirements = requirements;

        createWindowComponents();
        pack();
        setVisible(true);

        setDefaultWindowSize();
}

/**************************************************************************
 * Anonymous class to listen for when the copy clipboard button is clicked.
 * When it is, the software security requirements text in the window
 * will be in the clipboard.
 */
private ActionListener copyClipboardListener = new ActionListener()
{
        public void actionPerformed(ActionEvent e)
        {
                // *** ERROR CHECK PRE-CONDITIONS ***
                assert(m_requirements != null);
                assert(!m_requirements.equals(""));
                // END ERROR CHECK PRE-CONDITIONS

                // Next line may throw an exception, let program fail if they are
                Clipboard systemClipboard = Toolkit.getDefaultToolkit().
                    getSystemClipboard();
                StringSelection temp = new StringSelection(m_requirements);

                systemClipboard.setContents(temp, temp); // Don't care if we lose
                    ownership of clipboard
        }
};

/**************************************************************************
 * createWindowComponents()
 *
 * Creates all the JComponents in this window.
 */
private void createWindowComponents()
{
        // *** ERROR CHECK PRE-CONDITIONS ***
        assert(m_requirements != null);
        assert(!m_requirements.equals(""));
        // END ERROR CHECK PRE-CONDITIONS
```

```
                JPanel windowPanel = new JPanel();
                windowPanel.setLayout(new BoxLayout(windowPanel, BoxLayout.Y_AXIS));

                JTextArea requirementTextArea = new JTextArea(m_requirements);
                requirementTextArea.setWrapStyleWord(true);
                requirementTextArea.setLineWrap(true);
                requirementTextArea.setEditable(false);
                windowPanel.add(new JScrollPane(requirementTextArea));

                JPanel buttonPanel = new JPanel();
                buttonPanel.setLayout(new BoxLayout(buttonPanel, BoxLayout.X_AXIS));

                JButton copyClipboard = new JButton("Copy to Clipboard");
                copyClipboard.addActionListener(copyClipboardListener);

                buttonPanel.add(Box.createHorizontalGlue());
                buttonPanel.add(copyClipboard);
                buttonPanel.add(Box.createHorizontalGlue());

                windowPanel.add(buttonPanel);

                setContentPane(windowPanel);
        }

        /**************************************************************************
         * Helper method to set the window size based on the screen resolution.
         * The window is 2/3rds the screen size in the x direction, and 2/3rds the
         * direction in the y direction. The window is automatically centered.
         */
        private void setDefaultWindowSize()
        {
                // Next line may throw an exception, let program fail if they are
                Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();

                // Window is 2/3rds the screen size in each direction
                int windowWidth = (int)((0.667 * screenSize.getWidth()));
                int windowHeight = (int)((0.667 * screenSize.getHeight()));
                setSize(windowWidth, windowHeight);

                // Window starts at 1/6th into the screen (1/4 the window size)
                setLocation((int)(windowWidth * 0.25), (int)(windowHeight * 0.25));
        }
}
```

Listing D.3: HelpWindow.java

```
package edu.vt.cirt;

import java.awt.Dimension;
import java.awt.Toolkit;
import java.security.InvalidParameterException;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;


/**************************************************************************
 * Basic window that only has a text area for displaying help information.
 * Closing this window will not terminate the program.
 *
 * @author Lee M. Clagett II (lclagett@vt.edu)
 */
public class HelpWindow extends JFrame
{
        // Required by Java for serialization (not used)
```

```
private static final long serialVersionUID = 1L;

String m_helpText;

/**************************************************************************
 * Constructor( String )
 *
 * @param helpText The string that will be displayed as help
 *
 * @exception NullPointerException if the argument is null
 * @exception InvalidParameterException if the argument contains a blank string
 */
public HelpWindow(String helpText)
{
        super("Prototype SSRT Tool ~ Help");

        // *** ERROR CHECK PARAMETERS
        if(helpText == null)
                throw new NullPointerException("Constructor cannot be given a null
                    argument");

        if(helpText.equals(""))
                throw new InvalidParameterException("Constructor cannot be given an
                    empty string");
        // END ERROR CHECK PARAMETERS

        m_helpText = helpText;

        createWindowComponents();
        pack();
        setVisible(true);

        setDefaultWindowSize();
}

/**************************************************************************
 * createWindowComponents()
 *
 * Helper method that creates all UI components for the window.
 */
private void createWindowComponents()
{
        // *** ERROR CHECK PRE-CONDITIONS ***
        assert(m_helpText != null);
        assert(!m_helpText.equals(""));
        // END ERROR CHECK PRE-CONDITIONS

        JTextArea requirementTextArea = new JTextArea(m_helpText);
        requirementTextArea.setWrapStyleWord(true);
        requirementTextArea.setLineWrap(true);
        requirementTextArea.setEditable(false);
        add(new JScrollPane(requirementTextArea));
}

/**************************************************************************
 * setDefaultWindowSize()
 *
 * Helper method for setting the default window size and location. The
 * window is 2/3rds the screen size in both x and y directions. The window
 * will be centered.
 */
private void setDefaultWindowSize()
{
        // Next line may throw an exception, let program fail if they are
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
```

```
                // Window is 2/3rds the screen size in each direction
                int windowWidth = (int)((0.667 * screenSize.getWidth()));
                int windowHeight = (int)((0.667 * screenSize.getHeight()));
                setSize(windowWidth, windowHeight);

                // Window starts at 1/6th into the screen (1/4 the window size)
                setLocation((int)(windowWidth * 0.25), (int)(windowHeight * 0.25));
        }
}
```

Listing D.4: SSRT.java

```
package edu.vt.cirt;

import java.awt.Component;
import java.awt.event.MouseEvent;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.EventObject;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import javax.swing.JTree;
import javax.swing.event.CellEditorListener;
import javax.swing.tree.DefaultTreeCellRenderer;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreeCellEditor;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;


/**************************************************************************
 * Represents the actual SSRT. Given a dcssrt filename, and the requirements
 * filename, it will parse those values and use the provided information.
 * Handles all draw operations on the tree, the "editing" capabilities, and
 * requirements gathering on selecting nodes.
 *
 * @author Lee M. Clagett II (lclagett@vt.edu)
 */
public class SSRT extends JTree
{
        // Required by Java for serialization (not used)
        private static final long serialVersionUID = 1L;

        private FunctionalRequirement m_rootRequirement;


        /**********************************************************************
         * Constructor( String, String )
         *
         * Create the SSRT with the required ssrt and requirements xml
         * file names.
         *
         * @param ssrtFilename Location of the ssrt xml file
         * @param requirementsFilename Location of the requirements xml file
         *
         * @exception NullPointerException if either argument is null
         * @throws FileNotFoundException If the ssrt and requirements xml file cannot be
             found
         * @throws IOException If there is an issue while reading the ssrt and requirements
             file
         * @throws SAXException If the xml files do not meet their specification
         * @throws ParserConfigurationException If there is an issue with setting up the
             parser
         */
```

```
public SSRT(String ssrtFilename, String requirementsFilename) throws
    FileNotFoundException, IOException, SAXException, ParserConfigurationException
{
        // *** ERROR CHECK PARAMETERS ***
        if(ssrtFilename == null || requirementsFilename == null)
                throw new NullPointerException("Parameters to DCSRRT constructor
                    cannot be null");
        // END ERROR CHECK PARAMETERS

        SSRTParser theParser = new SSRTParser(ssrtFilename, new RequirementsParser(
            requirementsFilename));

        m_rootRequirement = theParser.getRootFunctionalRequirement();
        m_rootRequirement.addTreeRepaintListenerIteratively(m_ac_repaintListener);
        m_rootRequirement.setChecked(true);
        setModel(new DefaultTreeModel(m_rootRequirement)); // Set the tree model as
            the root requirement

        setCellRenderer(m_ac_requirementRenderer);
        setCellEditor(m_ac_functionalRequirementEditor);
        setEditable(true);
}

/**********************************************************************
 * Anonymous class for the tree renderer
 */
private DefaultTreeCellRenderer m_ac_requirementRenderer = new
    DefaultTreeCellRenderer()
{
        // Required by Java for serialization (not used)
        private static final long serialVersionUID = 1L;

        /**********************************************************************
         * Returns the UI component associated with the requirement
         */
        public Component getTreeCellRendererComponent(JTree tree, Object value,
            boolean selected, boolean expanded, boolean leaf, int row, boolean
            hasFocus)
        {
                // If null or if the root node, return the default cell renderer
                if(value == null || !(value instanceof Requirement))
                {
                        super.getTreeCellRendererComponent(tree, value, selected,
                            expanded, leaf, row, hasFocus);
                        return this;
                }

                // We have a Requirement Node !!
                return ((Requirement)value).getUIComponent();
        }
};

/**********************************************************************
 * Anonymous class for the tree editor
 */
private TreeCellEditor m_ac_functionalRequirementEditor = new TreeCellEditor()
{
        /**********************************************************************
         * Returns the same Component as the treeCellRenderer
         */
        public Component getTreeCellEditorComponent(JTree tree, Object value,
            boolean isSelected, boolean expanded, boolean leaf, int row)
        {
                if(tree == null)
                        return null;
```

```
                return tree.getCellRenderer().getTreeCellRendererComponent(tree,
                    value, true, expanded, leaf, row, true);
        }

        public void addCellEditorListener(CellEditorListener l) {}

        public void cancelCellEditing() {}

        public Object getCellEditorValue() { return null; }

        /**************************************************************************
         * @return True if not the root node
         */
        public boolean isCellEditable(EventObject anEvent)
        {
                // Expecting mouse events, not correct if we don't get it
                if(!(anEvent instanceof MouseEvent))
                        return false;

                MouseEvent click = (MouseEvent)anEvent;
                FunctionalRequirement clickNode = (FunctionalRequirement)
                    getPathForLocation(click.getX(), click.getY()).
                    getLastPathComponent();

                // Everything but the root is editable
                return clickNode != m_rootRequirement;
        }

        public void removeCellEditorListener(CellEditorListener l) {}

        public boolean shouldSelectCell(EventObject anEvent) { return true; }

        public boolean stopCellEditing() { return false; }
};

/**************************************************************************
 * Anonymous class to monitor when a request has been made for a tree
 * repaint. Immediately repaints the tree.
 */
private TreeRepaintListener m_ac_repaintListener = new TreeRepaintListener()
{
        public void treeRepaintRequested()
        {
                invalidate();
                repaint();
        }
};

/**************************************************************************
 * getSecurityRequirements()
 *
 * @return List of SecurityRequirements that have been selected
 */
public List<SecurityRequirement> getSecurityRequirements()
{
        return getSecurityRequirementsHelper();
}

/**************************************************************************
 * getSecurityRequirementsText()
 *
 * @return String of SecurityRequirements that have been selected, with a blank line
      between requirements
 */
public String getSecurityRequirementsText()
{
```

```
        String returnText = "";

        List<SecurityRequirement> requirementsList = getSecurityRequirementsHelper()
            ;
        for(SecurityRequirement secRequirement : requirementsList)
        {
                returnText += secRequirement.getRequirementText() + "\n\n";
        }

        return returnText;
}

/**************************************************************************
 * getSecurityRequirementsHelper()
 *
 * Combines code for getSecurityRequirements() and
 * getSecurityRequirementsText()
 *
 * @return List of SecurityRequirements that have been selected
 */
private List<SecurityRequirement> getSecurityRequirementsHelper()
{
        // *** ERROR CHECK PRE-CONDITIONS ***
        assert(m_rootRequirement != null);
        // END ERROR CHECK PRE-CONDITIONS

        List<SecurityRequirement> requirementsList = new LinkedList<
            SecurityRequirement>();

        // Do not check to see if root requirement is checked, its requirements MUST
             be included!
        m_rootRequirement.getSelectedSecurityRequirements(requirementsList);
        return removeDuplicateSecurityRequirements(requirementsList);
}

/**************************************************************************
 * removeDuplicateRequirements()
 *
 * Removes duplicate requirements from the list provided. This method will
 * modify the original list, and return the same list.
 *
 * @param securityRequirements The list of requirements that need duplicates removed
     . This list will be modified when duplicates are found.
 * @return The provided list with the duplicates removed
 */
private List<SecurityRequirement> removeDuplicateSecurityRequirements(List<
    SecurityRequirement> securityRequirements)
{
        // *** ERROR CHECK PARAMETERS
        assert(securityRequirements != null);
        // END ERROR CHECK PARAMETERS

        HashMap<String, SecurityRequirement> freshRequirements = new HashMap<String,
             SecurityRequirement>();

        ListIterator<SecurityRequirement> itey = securityRequirements.listIterator()
            ;
        while(itey.hasNext())
        {
                SecurityRequirement current = itey.next();

                // Found a duplicate so remove it
                if(freshRequirements.containsKey(current.getRequirementText()))
                        itey.remove();
                else
```

```
                                freshRequirements.put(current.getRequirementText(), current)
                                    ;
                }

                return securityRequirements;
        }
}
```

Listing D.5: TreeRepaintListener.java

```
package edu.vt.cirt;

import java.util.EventListener;


/**************************************************************************
 * Interface for child nodes in the tree, so they can request that the entire
 * tree structure be repainted when something changes.
 *
 * @author Lee M. Clagett II (lclagett@vt.edu)
 */
public interface TreeRepaintListener extends EventListener
{
        /**********************************************************************
         * treeRepaintRequested()
         *
         * Any implementor should repaint the tree structure when this method is
         * called.
         */
        public void treeRepaintRequested();

}
```

Listing D.6: CustomParser.java

```
package edu.vt.cirt;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.xml.sax.SAXException;


/**************************************************************************
 * An abstract class for parsing. This takes a filename as a string, and tries
 * to open the file. The constructor will immediately throw any exceptions
 * should there be a file I/O issue, or parsing issue.
 *
 * @author Lee M. Clagett II (lclagett@vt.edu)
 */
public abstract class CustomParser
{
        protected Document m_document;


        /**********************************************************************
         * Constructor( String )
         *
```

```
     * Creates a custom parser class. Throws exceptions if there is an issue
     * with the given filename.
     *
     * @param filename Path to the XML to open
     *
     * @exception NullPointerException If filename is null
     * @throws SAXException If there is a parsing issue with the given file
     * @throws FileNotFoundException If there the filename provided cannot be opened
     * @throws IOException If there is an issue while reading the file
     * @throws ParserConfigurationException If there is an issue with setting up the
         parser
     */
    public CustomParser(String filename) throws FileNotFoundException, SAXException,
        IOException, ParserConfigurationException
    {
            // *** ERROR CHECK PARAMETERS ***
            if(filename == null)
                    throw new NullPointerException("Constructor cannot be given a null
                        argument");

            File requirementsFile = new File(filename);

            if(!requirementsFile.canRead())
                    throw new FileNotFoundException("XML file error. The given filename:
                         " + filename + " does not exist, or cannot be opened.");
            // ERROR CHECK PARAMETERS

            // Create the XML parser provided by Java
            m_document = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse
                (requirementsFile);
    }

    /**************************************************************************
     * getAttributes( Node, List<String>, boolean )
     *
     * Will return a map containing the attributes of the given node. The
     * boolean flag determined whether the class should thrown an exception if
     * the number of attributes in the file does not match the given attributes
     * passed as a parameter.
     *
     * @param theNode The node to get attributes from
     * @param expectedAttributes List of attributes expected to be in the file for this
         node
     * @param strictEnforcement True iff an exception should be thrown when expected
         attributes do not match actual attributes
     * @return Map containing the attribute values as a node
     *
     * @throws SAXException If expected attributes does not match actual attributes (
         when strictEnforcement). If expected attribute is not found.
     */
    protected Map<String, Node> getAttributes(Node theNode, List<String>
        expectedAttributes, boolean strictEnforcement) throws SAXException
    {
            // *** ERROR CHECK PARAMETERS ***
            assert(theNode != null);
            assert(expectedAttributes != null);
            assert(expectedAttributes.size() > 0);
            // END ERROR CHECK PARAMETERS

            // ***ERROR CHECK XML FILE ***
            if(strictEnforcement && theNode.getAttributes().getLength() !=
                expectedAttributes.size())
                    throw new SAXException("In tag: " + theNode.getNodeName() + "
                        expected " + expectedAttributes.size() + " but received " +
                        theNode.getAttributes().getLength());
            // END ERROR CHECK XML FILE
```

```
                Map<String, Node> returning = new HashMap<String, Node>();
                for(String attribute : expectedAttributes)
                {
                        Node findNode = (Node)theNode.getAttributes().getNamedItem(attribute
                            );

                        // *** ERROR CHECK XML FILE ***
                        if(findNode == null)
                                throw new SAXException("Expected attribute named: " +
                                    attribute + " in tag: " + theNode.getNodeName());
                        // END ERROR CHECK XML FILE

                        returning.put(attribute, findNode);
                }

                return returning;
        }
}
```

Listing D.7: SSRTParser.java

```
package edu.vt.cirt;

import java.io.IOException;
import java.util.LinkedList;
import java.util.Map;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;


/*****************************************************************************
 * Parser for the SSRT xml file. Does strict error checking, throwing
 * exceptions if the specification is not met.
 *
 * @author Lee M. Clagett II (lclagett@vt.edu)
 */
public class SSRTParser extends CustomParser
{
        private RequirementsParser m_requirementsParser;


        /*************************************************************************
         * Constructor( String, RequirementsParser )
         *
         * @param filename Location of the SSRT filename
         * @param requirementsParser Requirements parser that will retrieve requirements
             associated with the SSRT
         *
         * @throws SAXException If there is a basic XML parsing issue, or if DCSSRT is not
             the name of the root node
         * @throws IOException If there is a basic file reading error with the SSRT file
         * @throws ParserConfigurationException If there is an issue with setting up the
             parser
         */
        public SSRTParser(String filename, RequirementsParser requirementsParser) throws
            SAXException, IOException, ParserConfigurationException
        {
                super(filename);

                // *** ERROR CHECK PARAMETERS ***
                if(requirementsParser == null)
                        throw new NullPointerException("");
```

```
        // END ERROR CHECK PARAMETERS

        // *** ERROR CHECK XML file ***
        Element rootNode = m_document.getDocumentElement();

        if(!rootNode.getNodeName().equals("DCSSRT"))
                throw new SAXException("DCSSRT XML file error. The root node must be
                    \"DCSSRT\".");
        // END ERROR CHECK XML file

        m_requirementsParser = requirementsParser;
}

/**********************************************************************
 * getRootFunctionalRequirement()
 *
 * Returns the root node of the SSRT, which is guaranteed to be a
 * FunctionalRequirement (called a node in the thesis). Every functional
 * requirement automatically has its children functional requirements, and
 * associated security requirements attached, including the root node.
 *
 * @return The root of the SSRT
 *
 * @throws SAXException If the provided SSRT file does not meet the specifications
 */
public FunctionalRequirement getRootFunctionalRequirement() throws SAXException
{
        // *** ERROR CHECK PRE-CONDITIONS ***
        assert(m_document != null);
        // END ERROR CHECK PRE-CONDITIONS

        return digNodeAndCreateFunctionalRequirement(m_document.getDocumentElement()
            , "SSRT", "Requirements of this node must be included with every project
            .");
}

/**********************************************************************
 * functionalRequirementHelper( Node )
 *
 * Turns the given node from the XML file, into a FunctionalRequirement
 * with its children and security requirements attached.
 *
 * @param requirement A FunctionalRequirement (node) from the XML file
 * @return FunctionalRequirement object based on this node
 *
 * @throws SAXException If the provided SSRT file does not meet the specifications
 */
private FunctionalRequirement functionalRequirementHelper(Node requirement) throws
    SAXException
{
        // *** ERROR CHECK PARAMETERS ***
        assert(requirement != null);
        assert(requirement.getNodeType() == Node.ELEMENT_NODE);
        assert(requirement.getNodeName().equals("Node"));
        // END ERROR CHECK PARAMETERS

        LinkedList<String> expectedAttributes = new LinkedList<String>();
        expectedAttributes.add("name");
        expectedAttributes.add("description");

        Map<String, Node> nodeAttributes = getAttributes(requirement,
            expectedAttributes, true);

        return digNodeAndCreateFunctionalRequirement(requirement, ((Node)
            nodeAttributes.get("name")).getFirstChild().getNodeValue(), ((Node)
            nodeAttributes.get("description")).getFirstChild().getNodeValue());
```

```
}

/**********************************************************************
 * digNodeAndCreateFunctionalRequirement( Node, String, String )
 *
 * Turns the given node from the XML file, into a FunctionalRequirement
 * with its children and security requirements attached. The difference is
 * the attributes describing the name and description will not be looked
 * up, it will take that information as a parameter instead. Created to
 * fix the fact that the root will not have a name and description.
 *
 * @param requirement A FunctionalRequirement (node) from the XML file
 * @param name The name of the FunctionalRequirement (node)
 * @param description The usage description of the FunctionalRequirement (node)
 * @return FunctionalObject based on this node
 *
 * @throws SAXException If the provided SSRT file does not meet the specifications
 */
private FunctionalRequirement digNodeAndCreateFunctionalRequirement(Node requirement
    , String name, String description) throws SAXException
{
        // *** ERROR CHECK PARAMETERS ***
        assert(requirement != null);
        assert(name != null);
        assert(!name.equals(""));
        assert(description != null);
        assert(!description.equals(""));
        // END ERROR CHECK PARAMETERS

        FunctionalRequirement temp = new FunctionalRequirement(name, description);

        NodeList children = requirement.getChildNodes();
        for(int i = 0; i < children.getLength(); i++)
        {
                Node currentNode = children.item(i);

                // Get only Element tags
                if(currentNode.getNodeType() == Node.ELEMENT_NODE)
                {
                        // If another branch, recursively go another layer deep
                        if(currentNode.getNodeName().equals("Node"))
                                temp.addRequirement(functionalRequirementHelper(
                                    currentNode));

                        // If a ending security requirement, gather the relevant
                            information
                        else if(currentNode.getNodeName().equals("Requirement"))
                                gatherSecurityRequirements(temp, currentNode);

                        // *** ERROR CHECK XML FILE ***
                        else // Branch tag should only contain another Branch tag or
                             a Requirement tag
                                throw new SAXException("DCSRRT XML file error.
                                    Invalid node name: " + currentNode.getNodeName()
                                     + ". Only valid types are \"Branch\", and \"
                                    Requirement\".");
                        // END ERROR CHECK XML FILE
                }
        }

        return temp;
}

/**********************************************************************
 * gatherSecurityRequirement( FunctionalRequirement, Node )
 *
```

```
         * Gets and creates all security requirements at the given node, and puts
         * them into the given FunctionalRequirement.
         *
         *
         * @param fillWithSecurityRequirements The functional requirement to associate with
           security requirements
         * @param requirementNode The node from the XML to search for security requirements
         *
         * @throws SAXException If the provided SSRT file does not meet the specifications
         */
        private void gatherSecurityRequirements(FunctionalRequirement
            fillWithSecurityRequirements, Node requirementNode) throws SAXException
        {
                // ERROR CHECK PARAMETERS
                assert(m_requirementsParser != null);
                assert(requirementNode != null);
                assert(requirementNode.getNodeType() == Node.ELEMENT_NODE);
                assert(requirementNode.getNodeName().equals("Requirement"));
                // END ERROR CHECK PARAMETERS

                LinkedList<String> expectedAttributes = new LinkedList<String>();
                expectedAttributes.add("type");
                expectedAttributes.add("number");

                Map<String, Node> attributes = getAttributes(requirementNode,
                    expectedAttributes, true);

                try
                {
                        for(SecurityRequirement secRequirement : m_requirementsParser.
                            getMatchingSecurityRequirements(((Node)attributes.get("type")).
                            getNodeValue(), Integer.parseInt((((Node)attributes.get("number"
                            )).getNodeValue()))))
                        {
                                fillWithSecurityRequirements.addRequirement(secRequirement);
                        }
                }
                catch(NumberFormatException e)
                {
                        throw new SAXException("DCSSRT XML file. The number attribute for a
                            Requirement must be a number.");
                }
        }
}
```

Listing D.8: RequirementsParser.java

```
package edu.vt.cirt;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.security.InvalidParameterException;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;


/**
 * Extension of the Custom Parser to grab software security requirements
 * associated to a specific constraints/assumptions. The file is not
 * automatically error checked - an error is only provided when there is an
 * issue with the requested constraint/assumption.
```

```
 *
 * @author Lee M. Clagett II (lclagett@vt.edu)
 */
public class RequirementsParser extends CustomParser
{
        /**************************************************************************
         * Constructor( String )
         *
         * @param filename Name of the requirements XML file
         *
         * @exception (From CustomParser) NullPointerException if the argument is null
         * @throws FileNotFoundException if the filename in the argument does not exist
         * @throws SAXException If the root node isn't Help
         * @throws IOException If there is an error when reading the XML file
         * @throws ParserConfigurationException If the parser configuration is not set
             correctly
         */
        public RequirementsParser(String filename) throws SAXException, IOException,
            ParserConfigurationException
        {
                super(filename);

                // *** ERROR CHECK XML File ***
                if(!m_document.getDocumentElement().getNodeName().equals("Requirements"))
                        throw new SAXException("Requirements XML file error. The root node
                            of the XML file must be \"Requirements\".");
                // ERROR CHECK XML FIle
        }

        /**************************************************************************
         * getMatchingSecurityRequirements( String, int )
         *
         * Given a constraint/assumption type, and number within that type, the
         * associated software security requirements are returned in a list.
         *
         * @param type The constraint/assumption type (memory, networking, etc.)
         * @param number The constraint/assumption number within the type
         * @return A list of software security requirements associated with the specified
             constraint/assumption
         *
         * @throws NullPointerException if the string parameter is null
         * @throws InvalidParameterException If the type attribute is blank, or the number
             attribute is less than 1
         * @throws SAXException If there is no constraint/assumption with the given type
             name, or if a child node to the root isn't name Type
         */
        public List<SecurityRequirement> getMatchingSecurityRequirements(String type, int
            number) throws SAXException
        {
                // *** ERROR CHECK PARAMETERS ***
                if(type == null)
                        throw new NullPointerException("Method cannot be given a null
                            argument.");

                if(type.equals(""))
                        throw new InvalidParameterException("The type attribute for a
                            requirement cannot be \"\". (blank)");

                if(number < 1)
                        throw new InvalidParameterException("The number attribute for a
                            requirement cannot be less than 1.");
                // END ERROR CHECK PARAMETERS

                // *** ERROR CHECK PRE-CONDITIONS ***
                assert(m_document != null);
                // END ERROR CHECK PRE-CONDITIONS
```

```
        NodeList types = m_document.getDocumentElement().getChildNodes();

        // Go through each type in the requirements; Memory, Networking, etc.
        for(int i = 0; i < types.getLength(); i++)
        {
                if(types.item(i).getNodeType() == Node.ELEMENT_NODE)
                {
                        // *** ERROR CHECK XML FILE ***
                        if(!types.item(i).getNodeName().equals("Type"))
                                throw new SAXException("Requirements XML file error.
                                    Children nodes to the root node must be named
                                    \"Type\".");
                        // END ERROR CHECK XML FILE

                        // If is the requirement type we are looking for
                        if(isMatchingRequirementType(types.item(i), type))
                                return goIntoMatchingTypeForSecurityRequirement(
                                    types.item(i), type, number);
                }
        }

        // If we reached here its an error because the security
        // requirement was not found in other XML file
        throw new SAXException("DCSSRT/Requirements XML file error.\nCould not find
            a Requirement in the requirement XML file,\nmatching the type named: " +
             type + " from the DCSSRT XML file");
}

/**************************************************************************
 * goIntoMatchingTypeForSecurityRequirement( Node, String, int )
 *
 * The node passed as a parameter must be a "Type" node. This method will
 * then search for the corresponding constraint/assumption based on the
 * integer given. The type name is given purely to store the name in the
 * SecurityRequirement object later - a choice made for simplicity.
 *
 * @param typeNode A node from the XML file - must be a "Type" node
 * @param type The type name of the Constraint/Assumption
 * @param number The constraint/assumption within the type
 * @return A list of software security requirements associated with the specified
     constraint/assumption
 *
 * @throws SAXException If a child to the node given as a parameter isn't Constraint
 */
private List<SecurityRequirement> goIntoMatchingTypeForSecurityRequirement(Node
    typeNode, String type, int number) throws SAXException
{
        // *** ERROR CHECK PARAMETERS ***
        assert(typeNode != null);
        assert(type != null);
        assert(!type.equals(""));
        assert(number > 0);
        // END ERROR CHECK PARAMETERS

        NodeList constraints = typeNode.getChildNodes();

        // Go through each constraint in this type of matching requirement
        for(int j = 0; j < constraints.getLength(); j++)
        {
                if(constraints.item(j).getNodeType() == Node.ELEMENT_NODE)
                {
                        // *** ERROR CHECK XML FILE ***
                        if(!constraints.item(j).getNodeName().equals("Constraint"))
```

```
                                throw new SAXException("Requirements XML file error.
                                    Child nodes to a \"Type\" node must be called
                                    \"Constraint\".");
                            // END ERROR CHECK XML FILE

                            // If is the constraint number we are looking for in this
                                type
                            if(isMatchingConstraintNumber(constraints.item(j), number))
                                return goIntoMatchingNumberForSecurityRequirement(
                                    constraints.item(j), type, number);
                    }
            }

            throw new SAXException("DCSSRT/Requirements XML file error.\nCould not find
                a security requirement in the requirements XML file,\nmatching the
                number " + number + " in the type named " + type + " from the DCSSRT xml
                 file.");
    }

    /**************************************************************************
     * goIntoMatchingNumberForSecurityRequirement( Node, type, number )
     *
     * The node passed as a parameter must be a "Constraint" node. This method
     * will then search for the corresponding software security requirements
     * based on the integer given. The type name, and number is given purely to
     * store the name in the SecurityRequirement object - a choice made for
     * simplicity.
     *
     * @param numberNode A node from the XML file - must be a "Constraint" node
     * @param type The type name of the Constraint/Assumption
     * @param number The constraint/assumption within the type
     * @return A list of software security requirements associated with the specified
         node
     *
     * @throws SAXException If the Node doesn't have the attribute name "text"
     */
    private List<SecurityRequirement> goIntoMatchingNumberForSecurityRequirement(Node
        numberNode, String type, int number) throws SAXException
    {
            // *** ERROR CHECK PARAMETERS ***
            assert(numberNode != null);
            assert(type != null);
            assert(!type.equals(""));
            assert(number > 0);
            // END ERROR CHECK PARAMETERS

            List<String> expectedAttributes = new LinkedList<String>();
            expectedAttributes.add("text");

            List<SecurityRequirement> returnList = new LinkedList<SecurityRequirement>()
                ;

            NodeList securityRequirements = numberNode.getChildNodes();

            // Go through each requirement for the constraint, filling up the list of
                security requirements
            for(int k = 0; k < securityRequirements.getLength(); k++)
            {
                    Node currentRequirement = securityRequirements.item(k);

                    if(currentRequirement.getNodeType() == Node.ELEMENT_NODE &&
                        currentRequirement.getNodeName().equals("Requirement"))
                    {
                            Map<String, Node> nodeAttributes = getAttributes(
                                currentRequirement, expectedAttributes, true);
```

```
                        returnList.add(new SecurityRequirement(type + " Requirement
                            #" + number, currentRequirement.getFirstChild().
                            getNodeValue(), ((Node)nodeAttributes.get("text")).
                            getFirstChild().getNodeValue()));
                }
        }

        return returnList;
}

/**********************************************************************
 * isMatchingRequirementType( Node, String )
 *
 * Determines if the node passed as a parameter has the matching name given
 * as a string parameter.
 *
 * @param type The node to check for a match
 * @param typeName The name of the type that we are looking for
 * @return True iff the type name matches the name of the type node
 *
 * @throws SAXException If the node does not have an attribute named name, or has an
      additional parameters
 */
private boolean isMatchingRequirementType(Node type, String typeName) throws
    SAXException
{
        // *** ERROR CHECK PARAMETERS ***
        assert(type != null);
        assert(typeName != null);
        assert(type.getNodeType() == Node.ELEMENT_NODE);
        assert(type.getNodeName().equals("Constraint"));
        // END ERROR CHECK PARAMETERS

        LinkedList<String> expectedAttributes = new LinkedList<String>();
        expectedAttributes.add("name");

        Map<String, Node> attributes = getAttributes(type, expectedAttributes, true)
            ;

        return ((Node)attributes.get("name")).getNodeValue().equals(typeName);
}

/**********************************************************************
 * isMatchingConstraintNumber( Node, int )
 *
 * Determines if the node passed as a parameter is the matching number
 * given as an int parameter.
 *
 * @param type The type node to check for a match
 * @param number The number of the constraint that we are looking for
 * @return True iff the number int matches the number of the constraint node
 *
 * @throws SAXException If the node does not have an attributes named number and
      value or has an attribute named something other than those two
 */
private boolean isMatchingConstraintNumber(Node type, int number) throws
    SAXException
{
        // *** ERROR CHECK PARAMETERS ***
        assert(type != null);
        assert(number > 0);
        // END ERROR CHECK PARAMETERS

        LinkedList<String> expectedAttributes = new LinkedList<String>();
        expectedAttributes.add("number");
        expectedAttributes.add("value");
```

```
                Map<String, Node> attributes = getAttributes(type, expectedAttributes, true)
                    ;

                return ((Node)attributes.get("number")).getNodeValue().equals(String.valueOf
                    (number));
        }
}
```

Listing D.9: HelpParser.java

```
package edu.vt.cirt;

import java.io.FileNotFoundException;
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;


/**************************************************************************
 * Extension of the custom parser to retrieve information from the help file.
 * This class throws exceptions if the help file format is not strictly
 * followed.
 *
 * @author Lee M. Clagett II (lclagett@vt.edu)
 */
public class HelpParser extends CustomParser
{
        /**********************************************************************
         * Constructor( String )
         *
         * @param filename Name of the help XML file
         *
         * @exception (From CustomParser) NullPointerException if the argument is null
         * @throws FileNotFoundException if the filename in the argument does not exist
         * @throws SAXException If the root node isn't Help
         * @throws IOException If there is an error when reading the XML file
         * @throws ParserConfigurationException If the parser configuration is not set
             correctly
         */
        public HelpParser(String filename) throws FileNotFoundException, SAXException,
            IOException, ParserConfigurationException
        {
                super(filename);

                if(!(m_document.getDocumentElement().getNodeName().equals("Help")))
                        throw new SAXException("The root element in the help file must be
                            named help");
        }

        /**********************************************************************
         * getHelpText()
         *
         * @return String of the help window text.
         *
         * @throws SAXException If the root node is empty, and contains no text
         */
        String getHelpText() throws SAXException
        {
                String output = m_document.getDocumentElement().getTextContent();
                if(output.equals(""))
                        throw new SAXException("Help XML Error. Help node cannot be empty.")
                            ;

                return m_document.getDocumentElement().getTextContent();
        }
```

```
}
```

Listing D.10: Requirement.java

```
package edu.vt.cirt;

import java.awt.Component;
import javax.swing.tree.DefaultMutableTreeNode;


/**************************************************************************
 * Base class for FunctionalRequirements (nodes), and SecurityRequirements
 * (Software Security Requirements). The base class allowed for heavy code
 * re-use, and a more simple design.
 *
 * @author Lee M. Clagett (lclagett@vt.edu)
 */
public abstract class Requirement extends DefaultMutableTreeNode
{
        // Required by Java for serialization (not used)
        private static final long serialVersionUID = 1L;

        private String  m_name;
        private String  m_description;
        private Requirement m_parent;

        /**********************************************************************
         * Constructor(String, String)
         *
         * @param name String of the text that will be displayed in the tree UI
         * @param description String of the text that will be displayed in the bottom of the
             UI when the requirement is selected
         *
         * @exception NullPointerException is thrown if name or description is null
         * @exception IllegalArgumentException is thrown if name or description is a blank
             string
         */
        public Requirement(String name, String description)
        {
                super(name);

                // *** ERROR CHECK PARAMETERS ***
                if(name == null || description == null)
                        throw new NullPointerException("Both name and text arguments CANNOT
                            be null");

                if(name.equals("") || description.equals(""))
                        throw new IllegalArgumentException("Both name and text arguments
                            CANNOT be blank");
                // END ERROR CHECK PARAMETERS

                m_name = name;
                m_description = description;
                m_parent = null;
        }

        /**********************************************************************
         * getUIComponent()
         *
         * @return The UI component associated with the derived class, that should be drawn
             in the tree structure
         */
        public abstract Component getUIComponent();

        /**********************************************************************
         * getName()
```

```
 *
 * @return String of the text that should be displayed in the tree structure of the
     DCSSRT.
 */
public String getName()
{
        // *** ERROR CHECK PRE-CONDITIONS ***
        assert(m_name != null);
        assert(!m_name.equals(""));
        // ERROR CHECK PRE-CONDITIONS

        return m_name;
}

/**************************************************************************
 * setName( String )
 *
 * @param newName The new name of the Requirement
 */
protected void setName(String newName)
{
        // *** ERROR CHECK PARAMETERS ***
        assert(newName != null);
        assert(!newName.equals(""));
        // END ERROR CHECK PARAMETERS

        m_name = newName;
        setUserObject(newName);
}

/**************************************************************************
 * getText()
 *
 * @return String of the text that should be displayed in the bottom window of the
     app.
 */
public String getText()
{
        // *** ERROR CHECK PRE-CONDITIONS ***
        assert(m_name != null);
        assert(!m_name.equals(""));
        // END ERROR CHECK PRE-CONDITIONS

        return m_description;
}

/**************************************************************************
 * setParent( Requirement )
 *
 * FunctionalRequirements can have children nodes, but either type of node
 * can have a parent (and will except for the root). In terms of SSRT
 * nomenclature, the parent of a SecurityRequirement (software security
 * requirement) is in fact the associated FunctionalRequirement (node). The
 * parent of a FunctionalRequirement (node) is of course, its true parent.
 *
 * @param newParent The parent or associated nodes to this Requirement
 */
protected void setParent(Requirement newParent)
{
        // *** ERROR CHECK PARAMETERS ***
        if(newParent == null)
                throw new NullPointerException("Method cannot be given a null value
                    as a parameter");
        // END ERROR CHECK PARAMETERS

        m_parent = newParent;
```

```
        }

        /**************************************************************************
         * getParent()
         *
         * FunctionalRequirements can have children nodes, but either type of node
         * can have a parent (and will except for the root). In terms of SSRT
         * nomenclature, the parent of a SecurityRequirement (software security
         * requirement) is in fact the associated FunctionalRequirement (node). The
         * parent of a FunctionalRequirement (node) is of course, its true parent.
         *
         * @return Parent or associated node to this requirement. Is null if no parent
             exists (only root has this property).
         */
        public Requirement getParent()
        {
                return m_parent;
        }
}
```

Listing D.11: FunctionalRequirement.java

```
package edu.vt.cirt;

import java.awt.Color;
import java.awt.Component;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.LinkedList;
import java.util.List;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JCheckBox;
import javax.swing.JLabel;
import javax.swing.JPanel;


/**************************************************************************
 * In the written thesis, this would represents what is called a "node". The
 * class was named to FunctionalRequirement to better fit into the design (both
 * the Software Security Requirements and the Nodes are based from the same
 * abstract class).
 *
 * @author Lee M. Clagett II (lclagett@vt.edu)
 */
public class FunctionalRequirement extends Requirement
{
        // Required by Java
        private static final long serialVersionUID = 1L;

        private List<Requirement> m_childRequirements;
        private JPanel m_thePanel;
        private JCheckBox m_checkbox;
        private JLabel m_numRequirementsLabel;

        private int m_securityRequirementsCount;

        private List<TreeRepaintListener> m_repaintListeners;

        /**************************************************************************
         * Constructor(String, String)
         *
         * @param name String of the text that will be displayed in the tree UI
         * @param description String of the text that will be displayed in the bottom of the
             UI when the requirement is selected
```

```
 *
 * @exception (From Requirement) NullPointerException is thrown if name or
     description is null
 * @exception (From Requirement) IllegalArgumentException is thrown if name or
     description is a blank string
 */
public FunctionalRequirement(String name, String text)
{
        super(name, text);

        m_childRequirements = new LinkedList<Requirement>();

        m_thePanel = new JPanel();
        m_thePanel.setBackground(new Color(1, 1, 1, 1));
        m_thePanel.setLayout(new BoxLayout(m_thePanel, BoxLayout.X_AXIS));

        m_checkbox = new JCheckBox();
        m_checkbox.setEnabled(true);
        m_checkbox.addActionListener(m_ac_checkboxListener);
        m_thePanel.add(m_checkbox);

        m_thePanel.add(new JLabel(name));
        m_numRequirementsLabel = new JLabel(" 0 software security requirements at
            this node");
        m_numRequirementsLabel.setFont(new Font("Default", Font.ITALIC, 10));
        m_thePanel.add(m_numRequirementsLabel);

        m_thePanel.add(Box.createHorizontalGlue());

        m_securityRequirementsCount = 0;
        m_repaintListeners = new LinkedList<TreeRepaintListener>();
}

/**********************************************************************
 * Anonymous class that detects when the checkbox held by this class
 * changes state. This enforces the correct SSRT usage: (1) all parent
 * nodes are selected when this checkbox is selected, and (2) all children
 * nodes are deslected when this checkbox is deselected.
 *
 * In reality this should be a static method, so there is
 * only one instance of this class, instead of an instance for each
 * FunctionalRequirement (node). But then a custom checkbox would be
 * required for a pointer back to this class, so yeah forget that.
 */
private ActionListener m_ac_checkboxListener = new ActionListener()
{
        public void actionPerformed(ActionEvent e)
        {
                // If not a JCheckBox or if the CheckBox is now selected, ignore the
                     event
                if( e == null || ! (e.getSource() instanceof JCheckBox))
                        return;

                if(((JCheckBox)e.getSource()).isSelected())
                        selectAllParents();
                else
                        deselectAllChildren();

                callTreeRepaintListeners();
        }
};


/**********************************************************************
 * isChecked()
 *
```

```
     * @return True iff the requirement is checked in the GUI
     */
    public boolean isChecked()
    {
            // *** ERROR CHECK PRE-CONDITIONS ***
            assert(m_checkbox != null);
            // END ERROR CHECK PRE-CONDITIONS

            return m_checkbox.isSelected();
    }

    public void setChecked(boolean value)
    {
            // *** ERROR CHECK PRE-CONDITIONS ***
            assert(m_checkbox != null);
            // END ERROR CHECK PRE-CONDITIONS

            m_checkbox.setSelected(value);
    }


    /**************************************************************************
     * addRequirement(Requirement)
     *
     * Adds the given requirement to the list of child requirements. A
     * functional requirement will have any number of children requirements,
     * which can be other functional requirements, or security requirements.
     *
     * @parameter addThis Requirement that will be added to the list of children
         requirements to this requirement
     *
     * @exception NullPointerException if the parameter is null
     * @throws ArithmeticException if more than Integer.MAX_VALUE SecurityRequirement
         objects are added
     */
    public void addRequirement(Requirement addThis)
    {
            // *** ERROR CHECK PRE-CONDITIONS ***
            assert(m_childRequirements != null);
            assert(m_numRequirementsLabel != null);
            assert(m_securityRequirementsCount >= 0);
            // END ERROR CHECK PRE-CONDITIONS

            // *** ERROR CHECK PARAMETERS ***
            if(addThis == null)
                    throw new NullPointerException("Parameter currentList cannot be null
                        ");
            // END ERROR CHECK PARAMETERS

            m_childRequirements.add(addThis); // Add to our list copy

            // Add only FunctionalReqirements to the tree GUI
            if(addThis instanceof FunctionalRequirement)
            {
                    add(addThis);
            }
            // Keep track of the number of SecurityRequirements associated with this
                FunctionalRequirement
            else if(addThis instanceof SecurityRequirement)
            {
                    // *** ERROR CHECK PARAMETERS ***
                    if(m_securityRequirementsCount == Integer.MAX_VALUE)
                            throw new ArithmeticException("Integer Overflow Exception! A
                                single node in the SSRT cannot have more than " +
                                Integer.MAX_VALUE + " SecurityRequirements.");
                    // END ERROR CHECK PARAMETERS
```

```
                m_securityRequirementsCount++;
        }

        addThis.setParent(this); // Set the parent of the incoming node
        m_numRequirementsLabel.setText(" " + m_securityRequirementsCount + "
            software security requirements at this node.");
}


/**************************************************************************
 * getUIComponent()
 *
 * Returns the UI component associated with this FunctionalRequirement.
 * Should be displayed in the tree cell.
 *
 * @return UI component to display in the tree cell, associated with the functional
     requirement.
 */
@Override
public Component getUIComponent()
{
        // *** ERROR CHECK PRE-CONDITIONS ***
        assert(m_thePanel != null);
        // END ERROR CHECK PRE-CONDITIONS

        return m_thePanel;
}

/**************************************************************************
 * addTreeRepaintListenerIteratively( TreeRepaintListener )
 *
 * Adds a tree repaint listener to this class for when a node changes, and
 * must be repainted. The listener will be added to all children of this
 * class.
 *
 * @param newListener for detecting when the tree needs repainting
 *
 * @exception NullPointerException If the parameter is null
 */
public void addTreeRepaintListenerIteratively(TreeRepaintListener newListener)
{
        // *** ERROR CHECK PARAMETERS ***
        if(newListener == null)
                throw new NullPointerException("Method cannot be given a null
                    argument.");
        // END ERROR CHECK PARAMETERS

        // *** ERROR CHECK PRE-CONDITIONS ***
        assert(m_checkbox != null);
        //  END ERROR CHECK PRE-CONDITIONS

        m_repaintListeners.add(newListener);

        // Add the same listener to all the children
        for(Requirement requirement : m_childRequirements)
        {
                if(requirement instanceof FunctionalRequirement)
                        ((FunctionalRequirement)requirement).
                            addTreeRepaintListenerIteratively(newListener);
        }
}

/**************************************************************************
 * getSelectedSecurityRequirements()
 *
```

```
     * @return List of SecurityRequirment objects that are children to this requirement,
          and currently selected in the UI
     */
    public List<SecurityRequirement> getSelectedSecurityRequirements()
    {
            return getSelectedSecurityRequirementsHelper(new LinkedList<
                SecurityRequirement>());
    }

    /**************************************************************************
     * getSelectedSecurityRequirements(List<SecurityRequirement>)
     *
     * Same as getSelectedSecurityRequirements() except the selected security
     * requirements will be added to the list provided as a parameter
     *
     * @exception InvalidParameter exception is thrown if parameter is null
     */
    public void getSelectedSecurityRequirements(List<SecurityRequirement> currentList)
    {
            // *** ERROR CHECK PARAMETERS ***
            if(currentList == null)
                    throw new NullPointerException("Parameter currentList cannot be null
                        ");
            // END ERROR CHECK PARAMETERS

            getSelectedSecurityRequirementsHelper(currentList);
    }

    /**************************************************************************
     * getSelectedSecurityRequirementsHelper( List<SecurityRequirement> )
     *
     * Helper function for determining which of the child security requirements
     * have been selected. The selected security requirements will be added to
     * the list that has been passed as a parameter.
     *
     * @param currentList List that will have selected security requirements added to it
     */
    private List<SecurityRequirement> getSelectedSecurityRequirementsHelper(List<
        SecurityRequirement> currentList)
    {
            // *** ERROR CHECK PRE-CONDITIONS ***
            assert(currentList != null);
            // END ERROR CHECK PRE-CONDITIONS

            // *** ERROR CHECK PARAMETERS ***
            assert(m_childRequirements != null);
            //  END ERROR-CHECK PARAMETERS

            for( Requirement requirement : m_childRequirements)
            {
                    if(requirement instanceof FunctionalRequirement && ((
                        FunctionalRequirement)requirement).isChecked())
                            ((FunctionalRequirement)requirement).
                                getSelectedSecurityRequirementsHelper(currentList);
                    else if(requirement instanceof SecurityRequirement)
                            currentList.add((SecurityRequirement)requirement);
            }

            return currentList;
    }

    /**************************************************************************
     * deselectAllChildren()
     *
     * Deselects the checkboxes of all children nodes. Then calls this method
     * on all the children, so that every child down the tree structure will be
```

```
         * deselected.
         */
        private void deselectAllChildren()
        {
                // *** ERROR CHECK PRE-CONDITIONS ***
                assert(m_childRequirements != null);
                assert(m_checkbox != null);
                // END ERROR CHECK PRE-CONDITIONS

                for(Requirement requirement : m_childRequirements)
                {
                        if(requirement instanceof FunctionalRequirement)
                        {
                                ((FunctionalRequirement)requirement).m_checkbox.setSelected(
                                    false);
                                ((FunctionalRequirement)requirement).deselectAllChildren();
                        }
                }
        }

        /**************************************************************************
         * selectAllParents()
         *
         * Selects the checkboxes of all parent nodes. Then calls this method on
         * the parent, so that every parent up the tree structure will be selected.
         */
        private void selectAllParents()
        {
                Requirement parentRequirement = getParent();

                // If no parent exists, stop, or
                // if parent is not FunctionalRequirement (not possible, but check anyway)
                if(parentRequirement == null || !(parentRequirement instanceof
                    FunctionalRequirement))
                        return;

                ((FunctionalRequirement)parentRequirement).m_checkbox.setSelected(true);
                ((FunctionalRequirement)parentRequirement).selectAllParents();
        }

        /**************************************************************************
         * callTreeRepaintListeners()
         *
         * Repaint all trees linked to this FunctionalRequirement (node), using the
         * repaint listeners.
         */
        private void callTreeRepaintListeners()
        {
                // Call each listener, requesting a repaint for each one
                for(TreeRepaintListener repaintListener : m_repaintListeners)
                        repaintListener.treeRepaintRequested();
        }
}
```

Listing D.12: SecurityRequirement.java

```
package edu.vt.cirt;

import java.awt.Component;
import javax.swing.JLabel;


/**
 * This represents a single software security requirement.
 *
 * @author Lee M. Clagett (lclagett@vt.edu)
```

```
 */
public class SecurityRequirement extends Requirement
{

        // Required by Java for serialization (not used)
        private static final long serialVersionUID = 1L;

        private JLabel m_label;
        private String m_requirement;

        /**********************************************************************
         * Constructor(String, String)
         *
         * @param name String of the text that will be displayed in the tree UI
         * @param description String of the text that will be displayed in the bottom of the
             UI when the requirement is selected
         * @param requirement String of the text that makes the actual security requirement.
             I.e. "The process shall not suck"
         *
         * @exception IllegalArgumentException If the parameter requirement is null or blank
             .
         * @exception (From Requirement) IllegalArgumentException is thrown if name or
             description is null
         * @exception (From Requirement) IllegalArgumentException is thrown if name or
             description is a blank string
         */
        public SecurityRequirement(String name, String description, String requirement)
        {
                super(name, description);

                // *** ERROR CHECK PARAMETERS ***
                if(requirement == null || requirement.equals(""))
                        throw new IllegalArgumentException("Parameter requirement cannot be
                            null or a blank string");
                // END ERROR CHECK PARAMETERS

                m_requirement = requirement;
                m_label = new JLabel(name);
        }

        /**********************************************************************
         * getRequirementText()
         *
         * @return Returns the text for the actual security requirement, i.e. "The process
             shall not crash"
         */
        public String getRequirementText()
        {
                // *** ERROR CHECK PRE-CONDITIONS ***
                assert(m_requirement != null);
                assert(!m_requirement.equals(""));
                // END ERROR CHECK PRE-CONDITIONS

                return m_requirement;
        }

        /**********************************************************************
         * getUIComponent()
         *
         * @return The JComponent associated with the security requirement (a JLabel)
         */
        @Override
        public Component getUIComponent()
        {
                // *** ERROR CHECK PRE-CONDITIONS ***
                assert(m_label != null);
```

```
                // END ERROR CHECK PRE-CONDITIONS

                return m_label;
        }
}
```

Bibliography

[1] Acunetix. Acunetix: Web security scanner. Retrieved July 2009 from http://www.acunetix.com/vulnerability-scanner/download.htm.

[2] Edward G. Amoroso. Fundamentals of Computer Security Technology. Prentice Hall, Englewood Cliffs, NJ, 1994.

[3] Axelle Apvrille and Makan Pourzandi. Secure software development by example. IEEE Security and Privacy, 3(4):10–17, 2005.

[4] Brad Arkin, Scott Stender, and Gary McGraw. Software penetration testing. IEEE Security and Privacy, 3(1):84–87, 2005.

[5] James D. Arthur, Anil Bazaz, Richard E. Nance, and Osman Balci. Mitigating security risks in systems that support pervasive services and computing: Access-driven verification, validation and testing. International Conference on Pervasive Services, 0:109–117, 2007.

[6] American Bankers Association. Digital signatures using reversible public key cryptography for the financial services industry. Technical Report ANSI X9.31-1998, 1998. Appendix A.2.4.

[7] American Bankers Association. Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ecdsa). Technical Report ANSI X9.62-1998, 1998. Annex A.4.

[8] Elain Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management. Technical Report Special Publication 800-57, National Institute for Standards and Technology, 2007.

[9] Elain Barker, Lily Chen, Andrew Regenscheid, and Miles Smid. Recommendation for pair-wise key establishment scheme using integer factorization cryptography. Technical Report Special Publication 800-56B, National Institute of Standards and Technology, 2009.

[10] Elain Barker, Don Johnson, and Miles Smid. Recommendation for pair-wise key establishment scheme using discrete logarithm cryptography. Technical Report Special Publication 800-56A, National Institute of Standards and Technology, 2007.

[11] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators (revised). Technical Report Special Publication 800-90, National Institute for Standards and Technology, 2007.

[12] Steven Bauer and Nissanka B. Priyantha. Secure data deletion for linux file systems. In SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium, pages 12–12, Berkeley, CA, USA, 2001. USENIX Association.

[13] Anil Bazaz. Framework for Deriving Verification and Validation Strategies to Assess Software Security. PhD thesis, Virginia Polytechnic Institute and State University, 2006.

[14] Kent Beck. Extreme Programming Explained. Addison-Wesley, Boston, MA, 2005.

[15] Matt Bishop and David Bailey. A critical analysis of vulnerability taxnomies. Technical Report CSE-96-11, Department of Computer Science, University of California, Davis, CA, 1996.

[16] Kurt Bittner. Use Case Modeling. Addison-Wesley, Boston, MA, 2003.

[17] Common Criteria Board. Common criteria for information technology security evaluation, version 3.1, 2009.

[18] Barry W. Boehm. Verifying and validating software requirements and design specifications. Software, IEEE, 1(1):75–88, 1984.

[19] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting mobile communications: The insecurity of 802.11. In MobiCom '01: Proceedings of the 7th annual international conference on Mobile computer and networking, pages 180–189, New York, NY, USA, 2001. ACM.

[20] Gustav Boström, Jaana Wäyrynen, Marine Bodén, Konstantin Beznosov, and Philippe Kruchten. Extending xp practices to support security requirements engineering. In SESS '06: Proceedings of the 2006 international workshop on Software engineering for secure systems, pages 11–18, New York, NY, USA, 2006. ACM.

[21] Jean Campbell and Randall Easter. Annex a: Approved security functions for fips pub 140-2, security requirements for cryptographic modules. Technical Report Federal Information Processing Standards Publication 140-2, National Institute for Standards and Technology, 2009.

[22] Jean Campbell and Randall Easter. Annex c: Approved random number generators for fips pub 140-2, security requirements for cryptographic modules. Technical Report Federal Information Processing Standards Publications 140-2, National Institute of Standards and Technology, 2009.

[23] Jean Campbell and Randall Easter. Annex d: Approved key establishment for fips pub 140-2, security requirements for cryptographic modules. Technical Report Federal Information Processing Standards Publication 140-2, National Institute of Standards and Technology, 2009.

[24] CERT. Cert statistics, 2009. Retrieved July 2009 from http://www.cert.org/stats/cert_stats.html.

[25] CESG. Cesg claims tested mark scheme issue 3.0.1, May 2009.

[26] Cigital. Its4: Software security tool. Retrieved September 2009 from http://www.cigital.com/its4/.

[27] Noopur Davis. Secure software development life cycle processes. Technical report, Carnegie Mellon University's (CMU) Software Engineering Insitute (SEI), 2005.

[28] Ulrich Drepper. Defensive programming for red hat enterprise linux (and what to do if something goes wrong). Technical report, Red Hat, 2009.

[29] Morris Dworkin. Recommendation for block cipher modes of operation. Technical Report Special Publication 800-38A, National Institute of Standards and Technology, 2001.

[30] Morris Dworkin. Recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality. Technical Report Special Publication 800-38C, National Institute of Standards and Technology, 2004.

[31] Morris Dworkin. Recommendation for block cipher modes of operation: The cmac mode for authentication. Technical Report Special Publication 800-38B, National Institute of Standards and Technology, 2005.

[32] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Technical Report Special Publication 800-38D, National Institute of Standards and Technology, 2007.

[33] Donald G. Firesmith. Engineering security requirements. Journal of Object Technology, 2(1):53–68, Janurary-February 2003.

[34] Office for Official Publications of the European Communities. Information technology security evaluation criteria (itsec), 1991.

[35] Charles B. Haley, Jonathan D. Moffett, Robin Laney, and Bashar Nuseibeh. A framework for security requirements engineering. In SESS '06: Proceedings of the 2006 international workshop on Software engineering for secure systems, pages 35–42, New York, NY, USA, 2006. ACM.

[36] Michael Howard and David LeBlanc. Writing Secure Code. Microsoft Press, Redmond, Washington, 2003.

[37] Michael Howard and Steve Lipner. The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software. Microsoft Press, Redmond, Washington, 2006.

[38] Security Innovation. Holodeck: Software testing using fault injection. Retrieved July 2009 from http://www.securityinnovation.com/holodeck/index.shtml.

[39] ISO/IEC. Programming languages - c. Technical Report 9899:1999, International Organization for Standardization, 1999.

[40] Nikolai Joukov and Erez Zadok. Adding secure deletion to your favorite file system. In SISW '05: Proceedings of the Third IEEE International Security in Storage Workshop, pages 63–70, Washington, DC, USA, 2005. IEEE Computer Society.

[41] Jan Jürjens. Secure Systems Development with UML. Springer Berlin Heidelberg, Berlin, New York, 2005.

[42] Burt Kaliski and Matt Robshaw. Message authentication with md5. RSA Labs' CryptoBytes, 1:5–8, 1995.

[43] RSA Laboratories. Pkcs #5 v2.1: Password-based cryptography standard. Technical report, RSA Security, October 2006.

[44] Steven Lavenhar. Code analysis. Technical report, Cigital, Inc, 2005.

[45] Wenbo Mao. Modern Cryptography: Theory and Practice. Prentice Hall, Upper Saddle River, NJ, 2003.

[46] Steve McConnell. From the editor - an ounce of prevention. IEEE Software, 18(3), 2001.

[47] Gary McGraw. Testing for security during development: Why we should scrap penetrate-and-patch. IEEE Aerospace and Electronic Systems Magazine, 13:13–15, 1998.

[48] Nancy Mead, Eric Hough, and Theodore Stehney. Security quality requirements engineering (square) methodology. Technical report, Carnegie Mellon Unviersity's (CMU) Software Engineering Institute (SEI), 2005.

[49] Nancy R. Mead. Security requirements engineering. Technical report, Carnegie Mellon Unversity's (CMU) Software Engineering Institute (SEI), 2008.

[50] Jeanne Meserve. 'smart grid' may be vulnerable to hackers. Electronic, 2009. Retrieved July 2009 from http://www.cnn.com/2009/TECH/03/20/smartgrid.vulnerability/index.html.

[51] C. C. Michael and Will Radosevich. Risk-based and functional security testing. Technical report, Cigital, Inc, 2005.

[52] Sun Microsystems. Download free java software - sun microsystems. Retrieved October 2009 from http://www.java/getjava.

[53] Russel Miles and Kim Hamilton. UML 2.0. O'Reilly Media, Inc., Sebastopol, Ca, 2006.

[54] Department of Defense. Trusted computer system evaluation criteria. Technical report, United States of America, 1985.

[55] United States Attorney's Office District of New Jersey. International telephone hacking conspiracy busted; indictment in the united states, arrests and searches in italy, and continued operations in the philippines, 2009.

[56] National Institute of Standards and Technology. Computer data authentication. Technical Report Federal Information Processing Standards Publication 113, U.S. Department of Commerce, 1985.

[57] National Institute of Standards and Technology. Automated password generator (apg). Technical Report Federal Information Processing Standards Publication 181, U.S. Department of Commerce, 1993.

[58] National Institute of Standards and Technology. Escrowed encryption standard (ees). Technical Report Federal Information Processing Standards Publication 185, U.S. Department of Commerce, 1994.

[59] National Institute of Standards and Technology. Advanced encryption standard (aes). Technical Report Federal Information Processing Standards Publication 197, U.S. Department of Commerce, 2001.

[60] National Institute of Standards and Technology. The keyed-hash message authentication code (hmac). Technical Report Federal Information Processing Standards Publication 198-1, U.S. Department of Commerce, 2002.

[61] National Institute of Standards and Technology. Recommendation for the triple data encryption algorithm (tdea) block cipher. Technical Report Special Publication 800-67, U.S. Department of Commerce, 2004.

[62] National Institute of Standards and Technology. Secure hashing standard (shs). Technical Report Federal Information Processing Standards Publicataion 180-3, U.S. Department of Commerce, 2008.

[63] National Institute of Standards and Technology. Digital signature standard (dss). Technical Report Federal Information Processing Standards Publication 186-3, U.S. Department of Commerce, 2009.

[64] OWASP. Webscarab. Retreived July 2009 from http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project.

[65] Roger Pressman. Software Engineering: A Practioner's Approach. McGraw-Hill, New York, NY, fifth edition, 2001.

[66] RTI. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, May 2002.

[67] Fred B. Schneider. Trust in Cyberspace. National Academy Press, Washington, DC, 1999.

[68] Bruce Schneier. Applied Cryptography. John Wiley & Sons, New York, NY, 1996.

[69] Bruce Schneier. Attack trees - modeling security threats. Dr. Dobb's Journal, 1999.

[70] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hyberston, Frank Buschmann, and Peter Sommerlad. Security Patterns. John Wiley & Sons, West Sussex, England, 2006.

[71] Robert C. Seacord. Secure Coding in C and C++. Addison-Wesley, Upper Saddle River, NJ, 2006.

[72] G. Sindre and A.L. Opdahl. Eliciting security requirements by misuse cases. In Technology of Object-Oriented Languages, International Conference on, pages 120–131, 2000.

[73] William Stallings. Cryptography and Network Security. Prentice Hall, Upper Saddle River, NJ, fourth edition, November 2005.

[74] Frank Swiderski and Window Snyder. Threat Modeling. Microsoft Press, Redmond, Washington, 2004.

[75] I.A. Tondel, M.G. Jaatun, and P.H. Meland. Security requirements for the rest of us: A survey. Software, IEEE, 25(1):20–27, Jan.-Feb. 2008.

[76] Carnegie Mellon University. Systems security engineering capability maturity model (sse-cmm) version 3.0. Technical report, Carnegie Mellon University, June 2003.

[77] Kenneth R. van Wyk. Adapting penetration testing for software development purposes. Technical report, Carnegie Mellon University's (CMU) Software Engineering Insitute (SEI), 2008.

[78] Kenneth R. van Wyk and Gary McGraw. Bridging the gap between software development and information security. IEEE Security and Privacy, 3(5):75–79, 2005.

[79] John Viega and Matt Messier. Secure Programming Cookbook for C and C++. O'Reilly, Sebastopol, CA, 2003.

[80] David Wagner. Boon. Retrieved September 2009 from http://www.eecs.berkeley.edu/ daw/boon/.

[81] David A. Wheeler. Flawfinder. Retrieved September 2009 from http://www.dwheeler.com/flawfinder/.

[82] Craig Wright, Dave Kleiman, and Shyaam Sundhar R.S. Overwriting hard drive data: The great wiping controversy. In ICISS '08: Proceedings of the 4th International Conference on Information Systems Security, pages 243–257, Berlin, Heidelberg, 2008. Springer-Verlag.

[83] Nobukazu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyama. A survey on security patterns. Progress in Informatics, 5:34–47, 2008.