VISUALIZING MEMORY UTILIZATION FOR THE PURPOSE OF VULNERABILITY ANALYSIS

by

William C. McConnell

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 and Application

May 28, 2008
Blacksburg, Virginia

Keywords: exploits, data capturing, function hooks, memory API, main memory, resource utilization, security, software vulnerabilities, visualization.

VISUALIZING MEMORY UTILIZATION FOR THE PURPOSE OF VULNERABILITY ANALYSIS

by William C. McConnell
Committee Chair: James Arthur
Computer Science

ABSTRACT

The expansion of the internet over recent years has resulted in an increase in digital attacks on computers. Most attacks, including the more dangerous ones, directly target program vulnerabilities. The increase in attacks has prompted a need to develop new ways to classify, detect, and avoid vulnerabilities. The effectiveness of these goals relies on the development of new methods and tools that facilitate the process of detecting vulnerabilities and exploits.

This thesis presents the development of a tool that provides a visual representation of main memory for the purpose of security analysis. The tool provides new insight into memory utilization by software; users are able to see memory utilization as execution time progression, visually distinguish between memory behaviors (allocations, writes, etc), and visually observe special relationships between memory locations. The insight enables users to search for visual evidence that software is vulnerable, violated, or utilizing memory incorrectly.

The development process for our visual tool has three stages: (1) identifying the memory utilization policies of the Windows 32-bit operating system; (2) identifying the data required for visual representations of memory and then implementing one possible method to capture the data; and (3) enumerating and implementing requirements for a memory tool that generates visual representations of memory for the purpose of vulnerability and exploit analysis.

ACKNOWLEDGEMENTS

First, I would like to thank my parents for providing encouragement and financial aide throughout my college years. I would also like to thank my close friends who have supported me with encouragement and company. Third, I want to thank my committee members for providing valuable input and attending my defense. Finally, I want to thank Dr. Arthur, my advisor, for giving me the opportunity for my graduate work and guiding me throughout the entire process.

TABLE OF CONTENTS

ABSTRACT .... **ii**

ACKNOWLEDGEMENTS .... **iii**

TABLE OF CONTENTS .... **iv**

LIST OF FIGURES .... **vii**

1 INTRODUCTION **1-1**

1.1 PROBLEM STATEMENT 1-4

1.2 ISSUES 1-5

1.3 SOLUTION APPROACH 1-6

1.4 ORGANIZATION 1-8

2 BACKGROUND **2-9**

2.1 SECURITY ASPECTS 2-9

2.1.1 Taxonomy of Vulnerabilities for Main Memory 2-10

2.1.2 Object Model for Main Memory 2-13

2.1.3 V&V Strategies for Main Memory 2-14

2.1.4 Primary Motivation 2-15

2.2 MEMORY VISUALIZATION 2-15

2.2.1 Memory Validator 2-15

2.2.2 Dump2Picture 2-17

2.2.3 MView 2-19

2.3 DATA CAPTURING 2-20

2.3.1 Strace 2-21

2.3.2 Detours 2-23

2.3.3 Other Tools 2-24

2.3.4 Comparisons 2-24

2.4 APPROACHES TO VISUALLY REPRESENTING MEMORY 2-25

2.4.1 Approach 1: Source Code and Compiler 2-25

2.4.2 Approach 2: Creating Visual Representations of Combined Memory Data 2-27

2.4.3 Approach 3: Controlled Environment 2-29

2.4.4 Summary and Chosen Approach 2-30

3 THE MEMORY MODEL 3-33

3.1 PHYSICAL MEMORY 3-33

3.2 VIRTUAL MEMORY 3-34

3.2.1 Virtual Addressing and Organization 3-36

3.2.2 Virtual Memory Management Layers 3-38

3.3 PROCESS MEMORY 3-42

3.3.1 Static Memory 3-43

3.3.2 Dynamic Memory 3-44

3.4 CONFIRMING THE MODEL 3-46

3.4.1 Maximum Stack Size Limitation 3-46

3.4.2 Stack Addressing 3-47

3.4.3 Hierarchy of Memory Management Functions 3-48

3.4.4 Process Memory Permission Rules 3-48

4 CAPTURING DATA RELATED TO MEMORY 4-51

4.1 TYPES OF MEMORY DATA 4-52

4.2 DATA SOURCES 4-53

4.3 POSSIBLE METHODS TO CAPTURE MEMORY DATA 4-54

4.4 CHOSEN DATA AND CAPTURING METHOD 4-56

4.4.1 Overview of the Application Programming Interface 4-57

4.4.2 Overview of API Functions Related to Memory 4-59

4.4.3 Overview of Inline Hooking 4-63

4.4.4 Implementation of API Function Hooking 4-66

4.4.5 Confirming the Accuracy of the Hook 4-69

4.5 CAPTURING READ & WRITE MEMORY OPERATIONS 4-70

4.6 DATA FORMAT REQUIREMENTS 4-70

5 MEMORY VISUALIZATION FOR SECURITY ANALYSIS 5-73

5.1 ISSUES AND CHALLENGES 5-73

5.2 DESIGN OPTIONS AND CHOICES 5-74

5.2.1 Design of the Graphical Environment: 5-74

5.2.2 Data Mapping: 5-75

5.2.3 Zooming, Interaction, Details, and Queries 5-76

5.3 REQUIREMENTS AND GUIDELINES 5-77

5.4 IMPLEMENTATION 5-80

5.4.1 The Event List 5-81

5.4.2 The Memory Image 5-82

5.4.3 User Interface 5-88

5.5 CONFIRMING THE VISUALIZATION 5-91

6 CONCLUSIONS AND FUTURE WORK 6-94

6.1 CONTRIBUTIONS 6-94

6.2 FUTURE WORK 6-97

6.3 SUMMARY 6-98

APPENDIX A A-100

APPENDIX B B-108

APPENDIX C C-109

BIBLIOGRAPHY Bib-112

LIST OF FIGURES

Figure 1: Constraints and assumptions for dynamic memory [Bazaz 2006]. 2-11

Figure 2: Constraints and assumptions for static memory [Bazaz 2006]. 2-12

Figure 3: Memory Validator 2-16

Figure 4: Hilbert Space Filling Curve 2-17

Figure 5: An overview and zoomed view of a Dump2Picture bitmap. 2-19

Figure 6: Virtual Address Visualization by MView [Bosch 2001] 2-20

Figure 7: Example output for Strace [BlindView Co. 2006]. 2-21

Figure 8: Function Flow for a Hooking Process 2-23

Figure 9: Microsoft's Apimon 2-24

Figure 10: Component overview for Approach 1. 2-26

Figure 11: Component overview for approach 2. 2-28

Figure 12: Component overview for approach 3. 2-29

Figure 13: Examples of Memory Configuration for the Windows OS 3-35

Figure 14: Virtual Addressing Example 3-37

Figure 15: Page frame states and available transitions between them. 3-39

Figure 16: Virtual Memory Management Layers 3-40

Figure 17: Stack Example 3-44

Figure 18: Stack Overload 3-46

Figure 19: Recursive Stack Overload 3-47

Figure 20: Recursive Call Addressing 3-47

Figure 21: API Hierarchy 3-48

Figure 22: Testing Process Memory Permissions 3-49

Figure 23: Writing to Non-Variable Data 3-50

Figure 24: The Inline Hooking Instruction Flow. 4-64

Figure 25: Setup Steps for an Inline Hook 4-65

Figure 26: Example of Function Definitions for the Hooking Process 4-67

Figure 27: Hook Function Example 4-68

Figure 28: Detours Component Interaction 4-69

Figure 29: Conformation Test for API Hooking 4-69

Figure 30: Data Format 4-72

Figure 31: Example of Mapping Addresses to a 2D Pixel Space 5-75

Figure 32: The Memory Event Class 5-82

Figure 33: Simplified Flow Chart for the Memory Bitmap 5-85

Figure 34: Un-Scaled vs. Scaled Bitmaps 5-86

Figure 35: Various Transparency Examples 5-87

Figure 36: Visual Scenarios 5-88

Figure 37: Example Layout for Our Tool. 5-90

Figure 38: Overview and Zoom in action. 5-91

Figure 39: Simple Testing Program. 5-92

Figure 40: Visual result of the code in Figure 39. 5-92

Figure 41: Visualizing Abnormal Behavior 5-93

1 Introduction

In recent years, the availability and use of the Internet have increased astronomically. Statistics show that the number of internet users increased to 1.093 billion in 2006 [IWS 2007]; furthermore, an increase to 1.78 billion Internet users by 2010 is expected [ETF 2007]. Not only is the number of Internet users increasing every year, but the way that they connect is changing and diversifying. Availability of both residential broadband access and public wireless networks has increased the average time per day that users spend connected to the Internet. Users can also connect to the Internet through increasing numbers of physical devices. Web-capable cell phones, palm devices, and video game systems become more common and Internet demanding every year. Furthermore, users perform more everyday tasks online like banking, communication, and shopping; these tasks require them to put vital personal information online. The increasing amount of personal information on the Internet has provided more incentive for hackers and digital thieves to exploit software for the purpose of identity theft, fraud, or other illegal activity.

The growth of the Internet has also resulted in a greater need for computer security to counteract viruses, worms, exploits, hacking, and other malicious computer activity. Computer security problems have existed since the early years of the modern computer. The majority of earlier security issues dealt with malicious programs that spread via email, download, or physical installation without an Internet connection. Once downloaded, the malicious program must be knowingly or accidentally executed by the user in order that it execute. However, starting in 2001, malicious programs began to focus primarily on exploiting software vulnerabilities [Kaspersky Labs 2007]. Now, software exploits have become associated with some of the most severe malicious programs because they can compromise security with little or no user interaction. Exploited vulnerabilities allow for directed attacks for the purpose of digital theft, hijacking, or sabotage. The Computer Emergency Response Team [CERT 2007] has documented 30,780 vulnerabilities since 1995 — 8,064 in 2006 alone. All of these attacks result in billions of dollars in losses from the need for patching software, the development of exploit counter-measures, and information recovery.

Many people believe that network applications are the primary target of software security attacks. However, statistics show that in recent years the number of exploits that target non-server applications has overtaken those that target server applications [Foster et al. 2005]. The most common exploits that target non-server applications are buffer overflows. Hackers use overflows to breach system security initially. More specifically, in 2004 buffer overflows accounted for 20% of the reported exploits. Detection and prevention of exploits requires an understanding of the exploit characteristics and vulnerabilities that allow an exploit to happen.

Two primary characteristics make the detection of vulnerabilities and exploits a complex and difficult task. The first characteristic occurs in the development stages of a given software release. In many software development life cycles, any method of vulnerability and exploit detection is done manually. Manual methods are typically slow and complex, and only detect a small fraction of the vulnerabilities and potential exploits that exist within a program. Furthermore, many developers and programmers are overly concerned with competition from other software, causing them to either forget vulnerability detection or be unconcerned about it; typically, they simply wait for the exploitation of vulnerabilities after releasing the software and then react by patching the software [Graff and Van Wyk 2003]. Sacrificing the development process and good programming standards for faster production time can lead to more vulnerabilities that, in turn, lead to more exploits. Ideally, detection of vulnerabilities and exploits should be a fast and effective process performed at any point in the lifetime of software. If existing detection processes were more efficient, they would be used more frequently and promote the development of more secure software.

Lack of understanding is the second primary characteristic that makes the detection of vulnerabilities and exploits a difficult and complex task. Most programmers and developers only deal with a vulnerability after it has been exploited. Therefore, programmers and developers typically have more understanding of exploits rather than the vulnerabilities that cause exploits. This lack of understanding causes two major problems: (1) developers and programmers fail to detect obvious and simple vulnerabilities, and (2), when they do detect a vulnerability or exploit, they repair it incorrectly, and subsequently cause new problems. If these people had a better understanding of how, where, and when vulnerability occurs, many software problems would be avoided before release, and problems that do occur could be found and repaired more effectively.

Problems during software development, and a lack of understanding of vulnerabilities, cause the introduction of vulnerabilities throughout software code. Finding new methods to detect vulnerabilities and exploits that work for all software resources can be difficult. However, concentrating on the different resources that software utilizes reduces the complexity of the detection process. For example, main memory is the source of many vulnerabilities and exploits. Buffer overflows are a common example of an exploited vulnerability. A general understanding of memory vulnerabilities is low because many developers assume that memory behaves properly, and therefore focus on other software functionality. Furthermore, operating systems keep utilization methods for main memory hidden from users. Therefore, to gain a better understanding of vulnerabilities requires new methods that directly focus on memory utilization.

1.1 Problem Statement

Visual analysis is one possible approach that can enhance our understanding of memory utilization, vulnerabilities, and exploits. Using visual representations of memory is a viable approach because memory utilization is constrained by spatial relationships that translate effectively to visual aspects – i.e., position, size, and color. Visual representations also utilize the natural visual abilities of the human brain to recognize patterns and detect anomalies. Hence, visual representations of memory that change over time elicit the best visual analysis. To generate a changing representation, the visual analysis should occur during the execution of a given program. Lastly, memory utilization defines a program's memory behavior. In other sciences, behavior is studied by making observations. Providing a visual representation of memory enables observations that are otherwise impossible due to the abstract nature of main memory; for example, if a user implements a program that allocates various block of memory, the user has limited awareness of how the block are located with respect to each other.

Visual analysis can occur at any point in the software life cycle. During the development phase, visual analysis can detect vulnerabilities and other potential problems before the release of software. After release, visual analysis can help the software maintenance phase by detecting vulnerabilities and increasing understanding of exploits. Furthermore, visual analysis is capable of focusing on various program sizes. Visual representations can focus on the memory utilization of a single function or an entire program.

Given the advantages and capabilities of visualization, we propose a memory visualization tool that can provide insightful visual representations of memory utilization. The visual representations must be accurate and able to work with enormous data sets. Furthermore, the insight enabled by the visual representations must facilitate the process of detecting vulnerabilities and exploits.

1.2 Issues

This section discusses the issues and difficulties with visually representing memory for the purpose of vulnerability and exploit analysis. Each major part in the process — data capture, visual representation, and the use of visuals to analyze vulnerability and exploits — present key concerns that must be overcome. The following major issues relate to generating visual representations of memory:

1) *Accuracy of Memory-Related Data*: Data related to the utilization of memory typically originates from the lower levels of an operating system. Low-level data can often be undocumented and strictly numerical. Initially, we must correctly identify the data required to represent memory visually. After obtaining the required data, we must correctly interpret its intended meaning. Otherwise, the data generates an incorrect visual representation of memory that can lead to incorrect vulnerability and exploit analysis
2) *Completeness of Memory-Related Data:* The lower level of the operating system – i.e., functionality that is inaccessible to the user – performs memory management and involves large amounts of instructions that execute extremely quickly. To collect a valid data set, we must obtain sufficient memory utilization instructions. Otherwise, the lack of data causes the generation of incomplete visual representations that can cause inaccurate vulnerability and exploit analysis.
3) *Obtaining Adequate Insight into Visual Representations of Memory*: Proper visual representation starts with organizing the required memory data. Next, we must map the

4) *Scalability of Visual Representations*: Given that memory data can become extremely large and complex, we must ensure that the visual representations of memory can effectively work with both small and large data sets. However, such visual representation cannot sacrifice detail to accommodate the size of a given data set.
5) *Understanding Memory Utilization*: Trying to detect vulnerabilities and exploits by analyzing visual representations of memory is pointless without understanding the memory utilization process. Therefore, we must understand how software utilizes memory before we design an analysis tool and use that tool.
6) *Enabling Correct Visual Analysis*: Vulnerability and exploit detection relies on accurate visual representations of memory throughout the execution time of a given program. This approach requires us to verify that a representation is changing correctly over time so that it depicts accurate behavior of software memory utilization. In addition, correct analysis requires users to have appropriate interaction methods with the visual representations.

1.3 Solution Approach

We propose to build a tool that generates visual representation of memory utilization to facilitate the process of vulnerability and exploit analysis detection. Developing a method for that uses visual representations of main memory requires three steps: (1) identifying memory utilization for an operating system, (2) identifying the required data needed for visual representation of memory and implementing a method that obtains the needed data, and (3) identifying requirements for a visual analysis tool and implementing a prototype tool. The details of each step are as follows:

1. *Identifying memory utilization:* For our work, we outline and confirm the memory management system for the Windows operating system. The outline begins at the base memory model; more specifically, this includes physical memory, virtual memory, management API, process memory, memory types, and memory structures. Using test programs that specifically elicits certain memory utilization verifies the memory model and adds information about lesser-known memory utilization aspects. Designing simple programs to specifically elicit certain memory behaviors will confirm the accuracy of our memory model definition. Understanding the memory model enables us to determine if a visual representation displays the correct information and utilization patterns. For example, if our model indicates that a given allocation must be within a certain memory range and then we implement a program that executes that given operation but, the visual representation displays the operation outside the expected range then we can infer that the representation is incorrect.
2. *Identifying and obtaining required data*. Definition of the required data starts by outlining the following: required data, possible locations at which the data can be obtained, and possible methods to capture the required data. We identify required data and its location by analyzing the memory model. Then, since there are several methods that can capture data, we choose to implement the capturing of API function calls using

3. *Identifying requirements for a visualization tool and developing a prototype.* The requirements identify the minimum needs to produce visual representations by combining our understanding of the memory model and the data captured by our API hook. We then use the requirements to implement a prototype tool that generates visual representations of memory for the purpose of vulnerability and exploit analysis. In particular, the requirements address the following aspects of the tool: scaling methods, design choices (e.g., colors), data navigation, and other features. Lastly, the functionality of the tool is tested by providing several sample programs as input for the tool.

1.4 Organization

The remainder of this thesis is organized as follows: Section 2 covers background sources that relate to our work. Section 3 presents a memory model for the 32-bit Windows operating system; this model guides the development of our tool. In section 4, we discuss the topics and research related to the data-capturing component of our effort. Section 5 discusses the design and implementation of our visual representation component. Finally, Section 6 contains conclusions and discussion about possible future work.

2 Background

In this section, we discuss background information and sources that relate to visually representing memory for the purpose of vulnerability and exploit analysis. There are four primary areas of background topics: (1) discussion of the underlying security motivations, (2) identification of preexisting memory tools that use visual representations (3) identification of related methods for capturing data related to memory utilization (4) discussion of other possible approaches to the visual representation problem.

2.1 Security Aspects

Security analysis is the primary motivation driving our interest in visual representations of memory. This research focuses on developing a visualization tool with the future goal of aiding vulnerability and exploit analysis. The goals and concepts that form the basis for our work originate from a Ph.D. dissertation by Anil Bazaz [2006] titled *A Framework for Deriving Verification and Validation Strategies to Assess Software Security*. The framework proposes new ways to characterize, identify, and detect vulnerabilities within software. More specifically, the framework helps our work by identifying the constraints, assumptions, and objects involved with memory vulnerabilities and memory exploits. The framework has three major components:

1. *A Taxonomy of Vulnerabilities*: The taxonomy defines constraints and assumptions that exist for the resource utilization of a program. The constraints and assumptions define the limitations of resource utilization or define expected functionality of resource utilization. If utilization of resources exceeds constrained limits or assumed behavior then a program becomes vulnerable to exploitation. Constraints and assumptions that relate to memory resources define the memory utilization that a visual representation needs to display.

2. *The object model*: The object model defines various program components — such as buffers — that exhibit constraints and assumption listed in the taxonomy. The objects defined for memory specify components that must be identifiable within a visual representation.
3. *Verification and validation (V&V) strategies*: This aspect of Bazaz's work presents methods that are used to test for violations of constraints and assumptions related to given process objects. Our visual representation of memory utilization facilitates and aids the V&V strategies presented in the dissertation by visually showing how the strategies test for violations.

Bazaz's dissertation organizes information by resource type – Input/Output, memory, etc. Building on this work, we focus on and use the information about main memory resources. In the following sub-sections, we discuss main memory aspects as they relate to constraints and assumptions, process objects, V&V strategies, and the development of a tool that visually represents memory.

2.1.1 Taxonomy of Vulnerabilities for Main Memory

The taxonomy identifies vulnerabilities for main memory by first outlining constraints and assumptions related to the memory utilization of a program during a secure run-state. Any deviation from these constraints and assumptions implies that the program has entered an insecure run-state and is vulnerable to an exploit. It is important to note that the existence of a vulnerable state only means that a program has the potential to transition to an exploited state. Bazaz divides the constraints and assumptions for main memory into dynamic and static categories.

Dynamic memory stores data that a program allocates depending on a changeable flow of execution. Thus, an operating system executes a program without previously knowing about the dynamic memory needs of the program. Furthermore, overall utilization of dynamic memory can vary for each execution of a program depending on variables such as timing, input data, or user interaction. At a lower level, stack and heap memory structures (defined in 3.3.2) guide utilization of dynamic memory. Lastly, numerous constraints and assumptions regulate utilization of dynamic memory and cause it to have a high risk of vulnerability. The list of constraints and assumptions for dynamic memory is given below in Figure 1 (as it appears in Bazaz's dissertation).

1) Data accepted as input by the process and assigned to a buffer will occupy and modify only specific locations allocated to the buffer.
2) The process will not interpret data present o the dynamic memory as executable code.
3) Environment variables being used by the process have expected format and values.
4) The process will be provided with the dynamic memory that it requests.
5) Data present on the dynamic memory cannot be observed while the process is in execution.
6) Data owned by the process and stored on the dynamic memory cannot be accessed after the process frees the memory.
7) A pointer variable being used by the process references a legal memory location.
8) A memory pointer returned by the underlying operating system does not point to zero bytes of memory.
9) A pointer variable being used by the process cannot reference itself.
10) Data accepted by the process will not be interpreted as a format string by the I/O routines.
11) The value of an integer variable/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.
12) A variable/expression used by the process as the index to a buffer will only hold values that allow it access to the memory locations assigned to the buffer.
13) An integer variable/expression used by the process to indicate length/quantity of any object will not hold negative values.

Figure 1: Constraints and assumptions for dynamic memory [Bazaz 2006].

Static memory is the second category of main memory. Unlike dynamic memory, an operating system knows how a program will utilize static memory before the program executes.

Hence, the size and location of static memory is fixed upon runtime and cannot vary during program execution. Furthermore, only two constraints and assumptions regulate the utilization of static memory, which means it has a low risk of vulnerability. The list of constraints and assumption for static memory is provided below in Figure 2 (as it appears in Bazaz's dissertation).

1) Data accepted as input by the process and assigned to a buffer occupies and modifies only specific locations allocated to buffer on the static memory.
2) Data held on the static memory cannot be observed while the process is in execution.

Figure 2: Constraints and assumptions for static memory [Bazaz 2006].

The constraints and assumptions listed in Figure 1 and Figure 2 dictate expected utilization of main memory. Most software, operating systems, and programmers follow these assumptions without anticipating or realizing that someone could manipulate their program beyond its intended functionality. It is also important to note that Bazaz's dissertation keeps the taxonomy very general; this approach allows application of the constraints and assumptions to a wide range of programs. Bazaz points out that his taxonomy does not include all vulnerabilities, because doing so would require multiple taxonomies and would be highly specific to certain programs.

Memory constraints and assumptions define the expected utilization of memory by a program. Violation of these constraints and assumptions identifies anomalous and incorrect memory utilization. Hence, the constraints and assumptions also identify the memory utilization of a process that a visual representation must display. Our visual representation gives additional insight into the constraints and assumptions by showing how they occur during program execution.

2.1.2 Object Model for Main Memory

The object model section of the dissertation uses the taxonomy to identify processes objects related to main memory such as buffers and variables. The object model further separates and defines ways to isolate and identify vulnerabilities. Expanding on the taxonomy, the object model identifies memory objects that have utilization behavior defined by given constraints and assumptions listed in the taxonomy. Like the taxonomy, the object model characterizes process objects for memory as dynamic memory or static memory. Bazaz's process objects for dynamic memory are the following:

1. Buffer: A buffer refers to a contiguous block of memory, whose length varies from 1 to n, where n is a positive integer. For example, an array is a buffer.
2. Environment variable: An environment variable is a dynamic value that affects the way that a running process behaves. There are a number of these variables with different names and values. For example, PATH is an environment variable provided by the UNIX operating system.
3. Integer variable: An integer variable refers to a variable that takes only integral values (positive, negative and zero). For example, in C language, variables declared as int are integer variables.
4. Index variable: An index variable refers to a variable that is used to identify elements in a buffer.
5. Pointer variable: A pointer variable refers to a variable whose value refers to or points to another variable stored elsewhere in the computer memory. For example, in C language, variables declared as int * are pointer variables that point to an integer variable.

Static memory objects share the buffer, integer variable, and index variable process objects with dynamic memory objects. However, static objects have different utilization than dynamic objects. More specifically, the size of static objects initializes to a fixed size and does not change throughout the execution of a program.

Process memory objects are important to a visual representation of memory because they identify structures and variables that must be displayed if one is to facilitate vulnerability and exploit analysis. In addition, process objects for memory identify focal points for the definition of the memory model in Section 3. Including the process objects in the memory model is especially important because understanding vulnerabilities and exploits of memory objects requires understanding of these objects' utilization behavior.

2.1.3 V&V Strategies for Main Memory

The V&V strategies section concludes the framework by describing strategies for detecting vulnerabilities in the memory of a given program. The base strategies are broken down into three parts: (1) selecting memory objects that could be vulnerable, (2) choosing constraints and assumptions that could show if vulnerability is possible, and (3) defining a method to test for violations of constraints and assumptions.

The strategies identify scenarios that enable the testing of memory objects to determine if they allow violations of a given constraint or assumption. First, we look at one example of a derived method to test a targeted dynamic memory object:

> **Goal**: To test if the software process allows violation of the following assumption: data accepted as input by the process and assigned to a buffer will occupy and modify only specific locations allocated to the buffer.
> **Target**: Buffer
> **Method**: We attempt to store data larger than the size of the buffer in a fixed length buffer. The assumption is considered violated if the process does not restrict the size of data and copies it into the buffer.

Strategies such as the one above are designed to produce specific behavior that identifies either normal or abnormal memory utilization; it is this utilization behavior that a visual representation must display. For example, if we implement the above test, the respective visual representation would show the occurrence of violations if the buffer overflows. Visually displaying the results of the V&V strategies enables better understanding and analysis of vulnerabilities and exploits.

2.1.4 Primary Motivation

In his conclusions, Bazaz states that future work is needed to identify new analysis tools that facilitate the use of the framework; this is precisely what inspired the development of our analysis tool that uses visual representations of memory. Our visual representations of memory can expand the framework in the following ways: (1) facilitate using the framework, (2) identify new vulnerabilities, (3) define new constraints and assumptions, and (4) expand the V&V strategies by adding more testing methods.

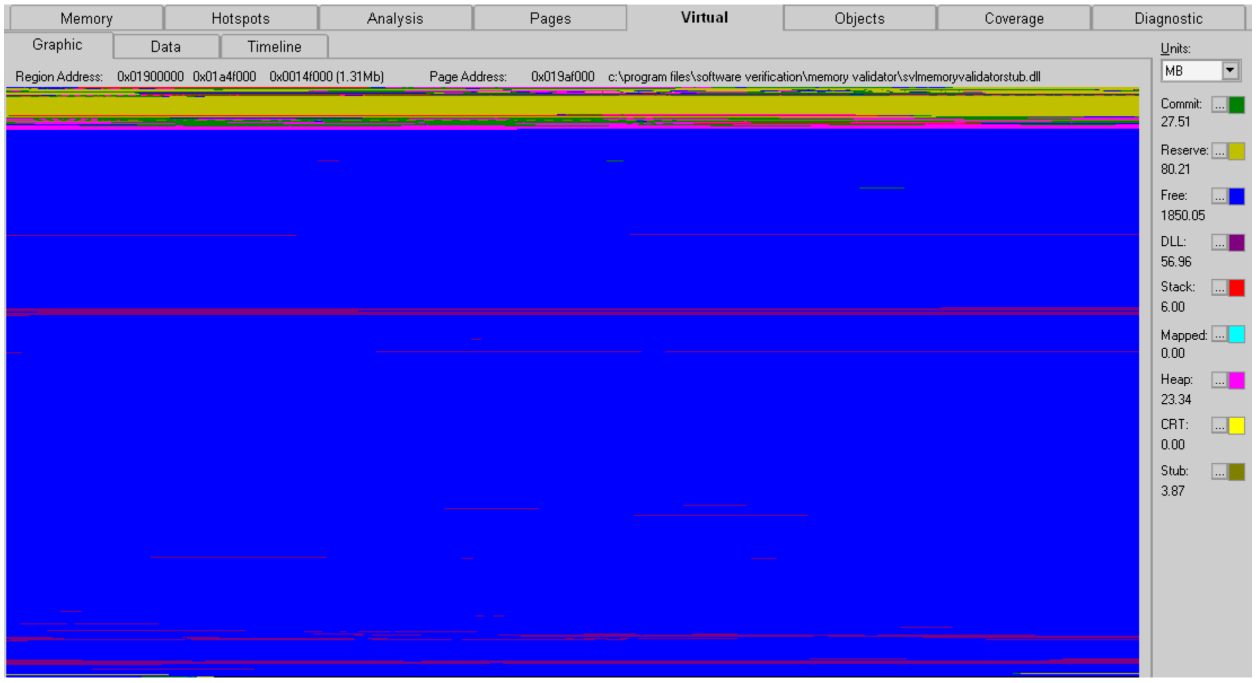
2.2 Memory Visualization

Visually representing memory is a key component in the development of our tool for vulnerability and exploits analysis. Unfortunately, very few existing tools create and utilize visual representations of memory in the manner that fit our needs of completeness and detail. This section discusses existing tools that use visual representations of memory. Information about each tool includes the following: (1) objectives, (2) functionality, and (3) relation to our work.

2.2.1 Memory Validator

Memory Validator is a diagnostic tool designed for software development [Software Verification 2006]. The design of the tool facilitates the detection of erroneous memory utilization that results during the development of new software. The tool is not open source, but has a large array of interesting features. Memory Validator gets its information by obtaining permissions to — or 'injecting' itself into — resources of a target process. Most of Validator's specifics are proprietary, but we are interested in the general aspects of its visualization.

Validator's representation of memory focuses on the virtual memory space of a process. The graphics are comprised of an NxN pixel section (Figure 3); virtual addresses are mapped to the screen first along the horizontal and then the vertical axis. The result is that the upper left corner represents the lowest address and the lower right corner represents the highest. In other terms, rows and columns map the addresses; this means that once a row is filled with address values, the next address is mapped to the first column of the next row.



Region Address: 0x01900000 0x01a4f000 0x0014f000 (1.31Mb) Page Address: 0x019af000 c:\program files\software verification\memory validator\svlmemoryvalidatorstub.dll

Commit: 27.51 🟩
Reserve: 80.21 🟨
Free: 1850.05 🟦
DLL: 56.96 🟪
Stack: 6.00 🟥
Mapped: 0.00 🟦
Heap: 23.34 🟪
CRT: 0.00 🟨
Stub: 3.87 🟫

Figure 3: Memory Validator

Types of memory such as free space or heap memory also map to certain colors — e.g., red for stack memory (as shown on the right side of Figure 3). Users can view specific data values of memory (address, for example) by using mouse interaction with the screen.

Unlike the methods used by our visualization tool, Validator is an example of representing the linear address space of memory within a two-dimensional graphical space. The advantage of using multiple dimensions is that the memory addresses map to more pixel space, which creates more detail. However, the overall special relationships between the addresses are lost. For example, the address represented by the last pixel in row one and the first pixel in row two are adjacent to each other in virtual memory, but are separated on the screen.

There are methods, however, that can map linear data to a graphical space without causing visual discontinuity. For example, Hilbert space filling curves can fill pixels mathematically [Rose 2001] so that any given data value is adjacent to the previous value and the next value (Figure 4).

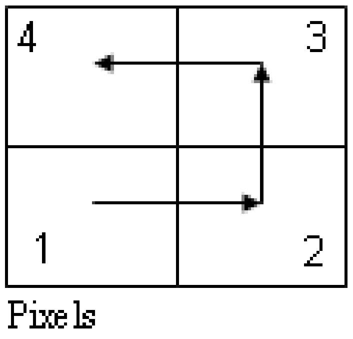


Figure 4: Hilbert Space Filling Curve

Although, Hilbert curves can preserve linear relationships on a 2D area, they have limitations that could affect our goal of supporting vulnerability analysis. For example, Figure 4 shows that the data value of 1 and 4 are physically adjacent to each other but are not numerically adjacent. In contrast, our approach represents memory with a single axis linear mapping that uses the physical distance between pixels to represent numerical distance between memory addresses. Utilizing strait linear mapping to one axis displays the memory as it is logically stored in memory.

2.2.2 Dump2Picture

Unlike Memory Validator, Dump2Picture is an example of a program that is less complex and which uses visual representations of memory [Vostokov 2007]. Dump2Picture provides a visual representation of memory for the purpose of program crash analysis. This program utilizes memory data recorded at a single point in time during the execution of a program. Then, as implied by its name, Dump2Picture converts memory data into a bitmap image.

The memory data used by Dump2Picture can reflect a variety of scenarios. The memory information can represent of all of the memory utilization of process, a specific address range, or the entire memory of a system. Unfortunately, this data represents memory utilization at only a single moment in time. In contrast, our visual representation tool uses data collected over the entire runtime of a process. Furthermore, in our program, only occurrences of memory utilization cause the recording of memory data – i.e., if there is no utilization activity then there is no data recording. We do not use Dump2Picture's data source for the following reasons:

1. Recording memory data at non-continuous points in execution time can miss vital information about vulnerabilities and exploits, which compromises the accuracy of a visual representation.
2. Capturing information about all memory utilization at a point in time produces redundant data. For example, if multiple captures are performed over a time span during which no memory utilization occurs, the data from each capture does not represent any change. Redundant data does not help analysis, but still requires processing that can reduce performance efficiency.

Dump2Picture's visual representation of memory is conveyed through a two-dimensional bitmap image (Figure 5-A). Each pixel in the bitmap represents an address range that is 8, 16, 24, or 32 bits long. The most notable characteristic of Dump2Picture is its use of transparency. Anytime memory does not take up an entire pixel, the color of that pixel has an adjusted transparency value (Figure 5-B). For example, a pixel representing a range of 1000 addresses that is 50% utilized results in a transparency value of 50%. Varying transparency is also a technique that we us for our tool because it ensures that all addresses are represented graphically regardless of display width.

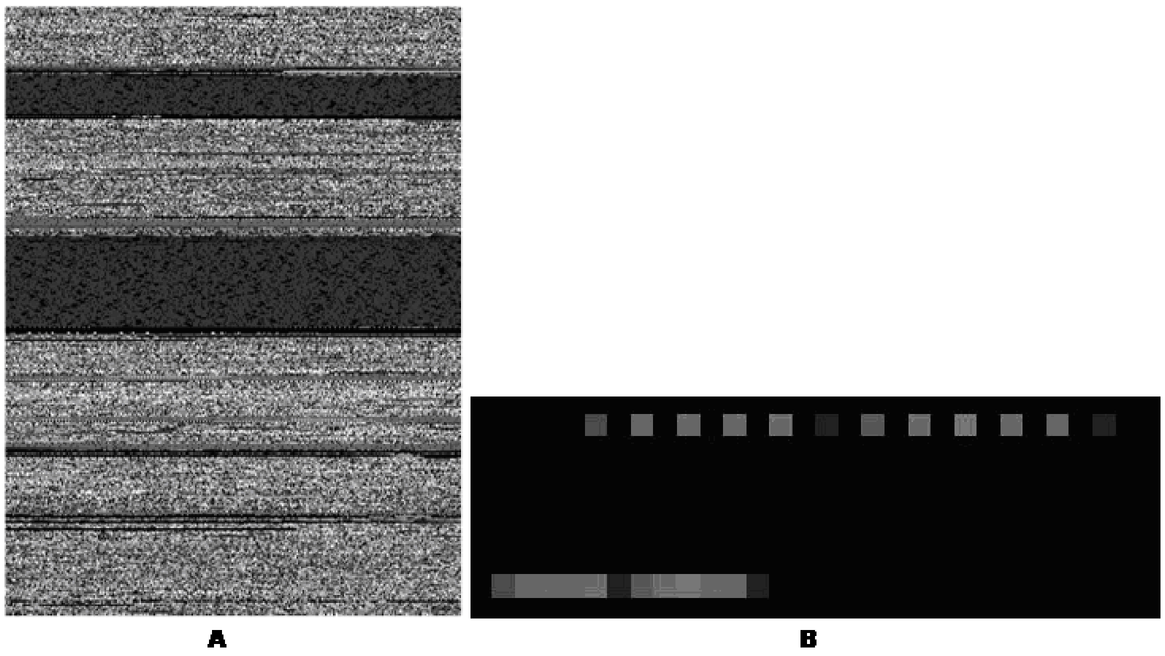


Figure 5: An overview and zoomed view of a Dump2Picture bitmap.

Considering that Dump2Picture has a specialized scope, it still provides a good example of a tool that visually represents memory. However, it is too limited to facilitate the goal of vulnerability and exploit analysis. The main limitation of Dump2Picture is its usage of data that is fixed to a point in time. Although it does use varying transparency, as does our program, it is limited to 32 bits per pixel and therefore has scalability issues with larger address ranges. Lastly, Dump2Picture differs from our approach by mapping memory addresses to a two-dimensional display area.

2.2.3 MView

The visualization tools presented above use an approach that differs from ours because they map address ranges to 2D graphical areas. In contrast to those tools, the MView visualization uses an address mapping that is similar to our approach. MView is a component of an overall visualization toolkit named Rivet [Bosch 2001] that focuses on the behavior of networked computer systems.

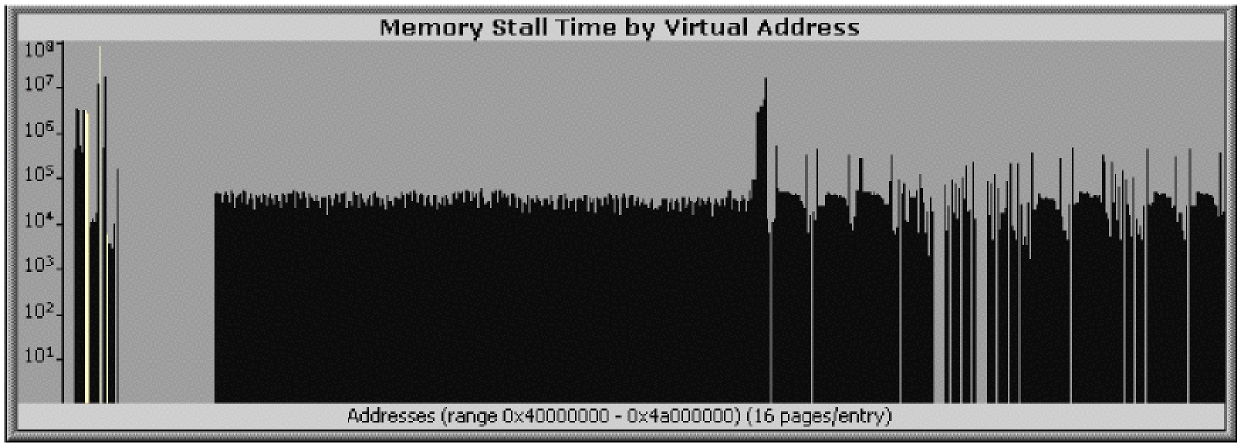


Figure 6: Virtual Address Visualization by MView [Bosch 2001]

Similar to our tool, MView maps virtual addresses to the horizontal axis in numerical sequence (Figure 6). Another similarity is that MView maps other primary data to the vertical axis; in this case, it maps the number of memory access delays (stalls) that occur on a networked multi-processor system. Furthermore, MView uses each horizontal pixel to represent multiple addresses. Unlike our approach that uses transparency, the pixels are treated as “bins” that contain the total count of memory stalls for the given address range, which then translates into the visual height of each column. Aside from the visual aspects, MView differs from our tool because it focuses on distributed memory systems.

2.3 Data Capturing

The goal of this research is to visually represent memory for the purpose of vulnerability and exploit analysis. However, visually representing memory would be difficult without first developing ways to capture accurate memory utilization data during program execution. In this section, we discuss several existing tools and methods that helped guide our data capturing methods.

2.3.1 Strace

Strace is an open source tool that captures data at the operating system level [BlindView Co. 2006]. More specifically, Strace captures calls made to functions defined by the Windows NT application programming interface (API). A method called "hooking" facilitates the data capturing. The level where the hook is located in the operating system – either the user or kernel level – determines the classification of the hook. Hooks that operate from the user level are generally stable and require few security permissions. By contrast, Strace utilizes a kernel hook that operates from the lowest level of the operating system. Unfortunately, kernel hooks require many security permissions to function and can be highly unstable. The design of Strace mimics a device driver, but requires no hardware interaction. Using a driver approach enables Strace to access kernel memory — addressed 0x80000000 and higher — that normal user programs cannot access.

```
3 133 139 NtAllocateVirtualMemory (-1, 1243984, 0, 1244028, 8192, 4, ... ) == 0x0
4 133 139 NtAllocateVirtualMemory (-1, 1243980, 0, 1244032, 4096, 4, ... ) == 0x0
5 133 139 NtAllocateVirtualMemory (-1, 1243584, 0, 1243644, 4096, 4, ... ) == 0x0
```

Figure 7: Example output for Strace [BlindView Co. 2006].

In other terms, Strace is an example of a rootkit. Rootkits are well known because they help hackers gain access to low level operating system information. However, Strace does not utilize a rootkit for malicious purposes. More specifically, Strace uses the following steps:

1. The operating system loads Strace into system memory as a device driver.
2. Strace gains access to the KeServiceDescriptorTable and the KeServiceDescriptorTableShadow. Both tables point to system service lists for

Ntoskrnl.exe and Win32k.sys [Hoglund and Butler 2006]. Strace hooks NT functions by changing both tables to point to Strace's own versions of each function.

3. When a user level program makes a call to a "hooked" function, the call first goes to the Strace version of the function.
4. Strace captures parameter values set in the function call. Figure 7 shows sample data that was captured by Strace.
5. Strace routes the function call to the original system function that processes the call.
6. The original system function sends return values to the Strace version, which then sends the return values to the program that originally made the call.
7. When Strace unloads from memory, it returns the KeServiceDescriptorTable and the KeServiceDescriptorTableShadow to the original versions. Otherwise, Strace causes the operating system to become unstable.

Strace is a command line program designed for data recording and does not feature automatic analysis capabilities. In addition, the developers of Strace, BlindView, stress that Strace is only beta quality and can causes system crashes. This instability also illustrates the volatile nature of kernel level hooks and makes them difficult to successfully program and modify. Strace provides one possible method for capturing data that is compatible with generating our visual representation of memory. However, Strace only captures NT API calls, whereas our goals also require data from other sources in order to produce an accurate visual representation of memory.

2.3.2 Detours

Detours is an open source tool developed by the Microsoft Research Department. There is also a professional version that is not free to download [Hunt and Brubacher 1999]. Detours gains access to function calls made to the Windows Application Programming Interface (API) for the purpose of adding temporary functionality. API functions provide an interface between the services offered by an operating system and the user programs that request services. Function hooks facilitate all capturing performed by Detours. Unlike Strace, Detours operates with the same security permissions as a typical program, which classifies it as a user level hook. More specifically, the tool intercepts calls to API functions using a technique called “inline hooking.”

Inline hooking works by positioning code between a running process and the API functions that it calls. The following steps comprise an over view of the inline hooking process for Detours ([Hunt and Brubacher 1999] & [Hoglund and Butler 2006]).

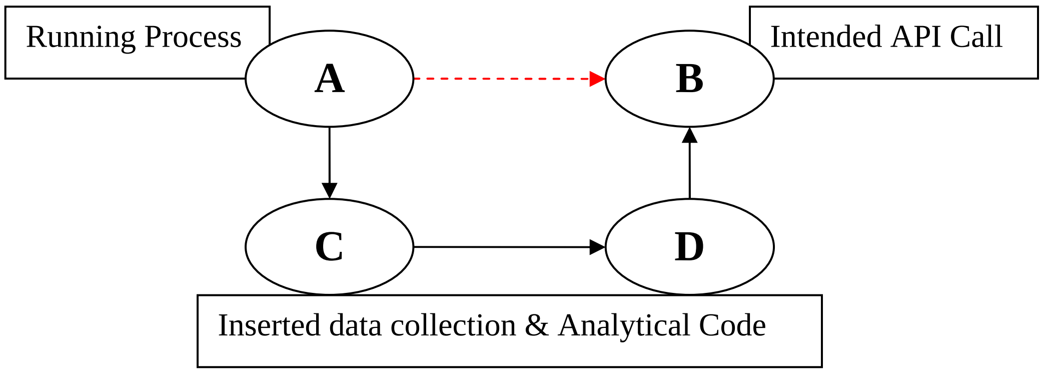


Figure 8: Function Flow for a Hooking Process

1. The process begins with a definition for a function to hook – Function A in Figure 8.
2. Detours changes Function A to call to Function C, instead of Function B as it would normally call.

3. Detours instructs Function C to call Function D.

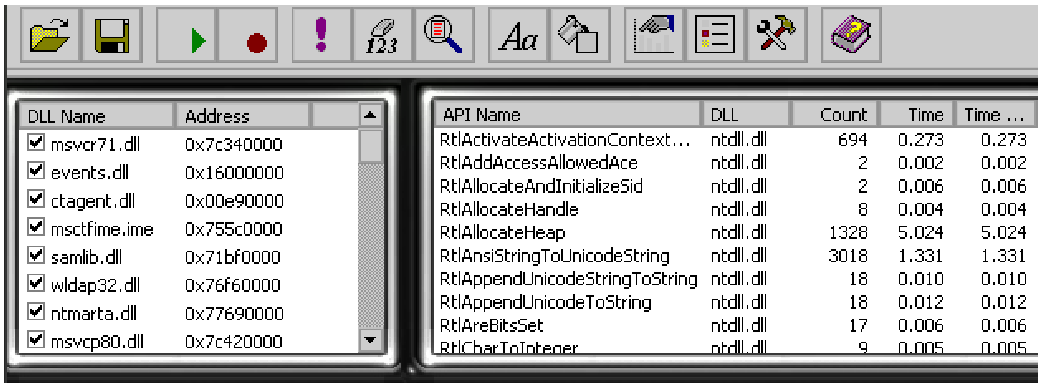
4. Function D is designed to direct the call flow back to Function B.

Before the above process occurs, Detours enables users to add code to hook functions that implements new functionality (e.g., data capture). Section 4.4.3 discusses in more detail about how to utilize Detours for data capturing.

2.3.3 Other Tools

API Monitor is another example of a data-capturing tool [APIMON 2006]. It is able to record API calls made by a running process that a user specifies. The data recorded by API Monitor is similar to the data recorded by Strace or Detours but with the added element of a user interface. Unfortunately, API Monitor only has a trial version and is not open source.

Microsoft provides its own API capture tool for Windows called Apimon (Figure 9). This tool utilizes the combination of API capturing and a user interface. Furthermore, Apimon allows for easy filtering. However, source code of Apimon is not available for modification.



DLL Name	Address
✓ msvcr71.dll	0x7c340000
✓ events.dll	0x16000000
✓ ctagent.dll	0x00e90000
✓ msctfime.ime	0x755c0000
✓ samlib.dll	0x71bf0000
✓ wldap32.dll	0x76f60000
✓ ntmarta.dll	0x77690000
✓ msvcp80.dll	0x7c420000

API Name	DLL	Count	Time	Time ...
RtlActivateActivationContext...	ntdll.dll	694	0.273	0.273
RtlAddAccessAllowedAce	ntdll.dll	2	0.002	0.002
RtlAllocateAndInitializeSid	ntdll.dll	2	0.006	0.006
RtlAllocateHandle	ntdll.dll	8	0.004	0.004
RtlAllocateHeap	ntdll.dll	1328	5.024	5.024
RtlAnsiStringToUnicodeString	ntdll.dll	3018	1.331	1.331
RtlAppendUnicodeStringToString	ntdll.dll	18	0.010	0.010
RtlAppendUnicodeToString	ntdll.dll	18	0.012	0.012
RtlAreBitsSet	ntdll.dll	17	0.006	0.006
RtlCharToInteger	ntdll.dll	9	0.005	0.005

Figure 9: Microsoft's Apimon

2.3.4 Comparisons

In section 2.3, we discuss examples of kernel level hooks, user level hooks, and capturing applications. As seen later in this thesis, we choose to utilize the Detours toolkit. Detours enables the most stable and easiest implementation of a capturing method for memory-related data.

These methods and tools are only examples of a few preexisting approaches for data capturing. Furthermore, these capturing examples only target API functions. Completeness of our visual representations requires data from various other sources.

2.4 Approaches to Visually Representing Memory

In this section, we identify possible approaches to the development of a tool for vulnerability and exploit analysis. The primary goal of each approach is to generate visual representations of memory for the purpose of vulnerability and exploit analysis. We have chosen to implement a visualization component that can be integrated into each approach. The first approach proposes generating a visual representation of memory by utilizing a compiler that collects memory data. The second approach generates visual representations of secure and insecure states for a given program. The third approach generates visual representations of an exploit signature. It is notable that any of the approaches could be developed into unique tools intended for vulnerability and exploit analysis.

2.4.1 Approach 1: Source Code and Compiler

The objective for the source code and compiler approach is to generate a visual representation for the memory utilization during a single execution of a program. This approach has three components (Figure 10): (1) a custom compiler, (2) an execution profiler, (3) and a visual generator. The purpose of the compiler is to generate a pre-execution template for memory utilization. The pre-execution template contains data related to memory structures and locations. The purpose of the execution profiler is to generate an execution profile by using the pre-execution template to help collect data related to a given execution. Finally, the purpose of the visual generator is to map the execution profile to a graphical space. The final output of the tool is a visual representation that users can manually inspect to perform vulnerability and exploit analysis.

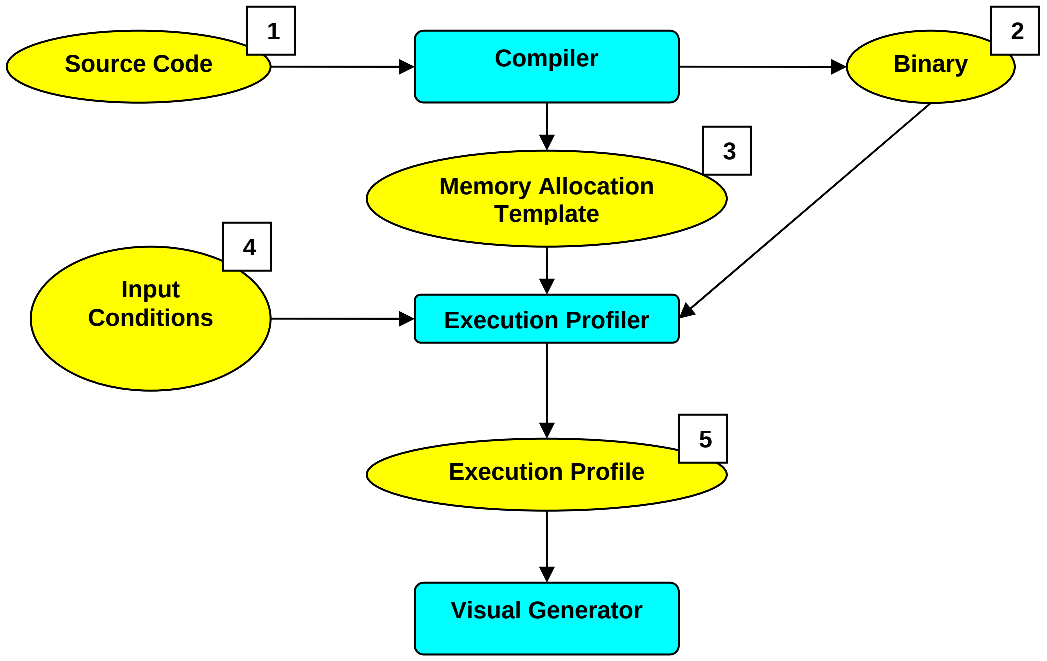


Figure 10: Component overview for Approach 1.

The approach begins with source code and an appropriate compiler. Modifying the compiler enables it to record data contained within the source code that relates to pre-execution memory. When the compiler runs, it produces a data file representing a pre-execution template for memory utilization (Figure 10-3) and an executable binary (Figure 10-2). The execution profiler adds information about memory utilization by executing the binary under given input conditions (Figure 10-4) while using the pre-execution template as a reference. The execution profiler generates a single file that represents an overall execution profile (Figure 10-5). For each unique set of input conditions, the execution profiler produces a unique data file for the total utilization of memory. Next, the visual generator component uses execution profile to create a visual representation that facilitates vulnerability and exploit analysis.

2.4.2 Approach 2: Creating Visual Representations of Combined Memory Data

The objective of the creating visual representations of combined memory data approach is to create signature profiles for the secure and insecure states of a given program by using visual representations. The purpose of secure and insecure profiles is to identify good versus bad behavioral trends. As input, the approach only requires an executable and does not need source code or pre-compiled information. A program to be tested runs under different input conditions in order to collect data related to memory utilization. The main goal of combining utilization data is the development of secure and insecure "signatures" that identify behavior related to memory utilization. A signature identifies the typical utilization characteristics of memory in a secure or insecure state. More specifically, a signature identifies behavioral bounds including: address ranges that a program normally accesses, minimum and maximum utilization of memory over time, and the general pattern of when and where memory access occurs. If a program exhibits memory utilization outside its normal bounds then it could be in a vulnerable or exploited state.

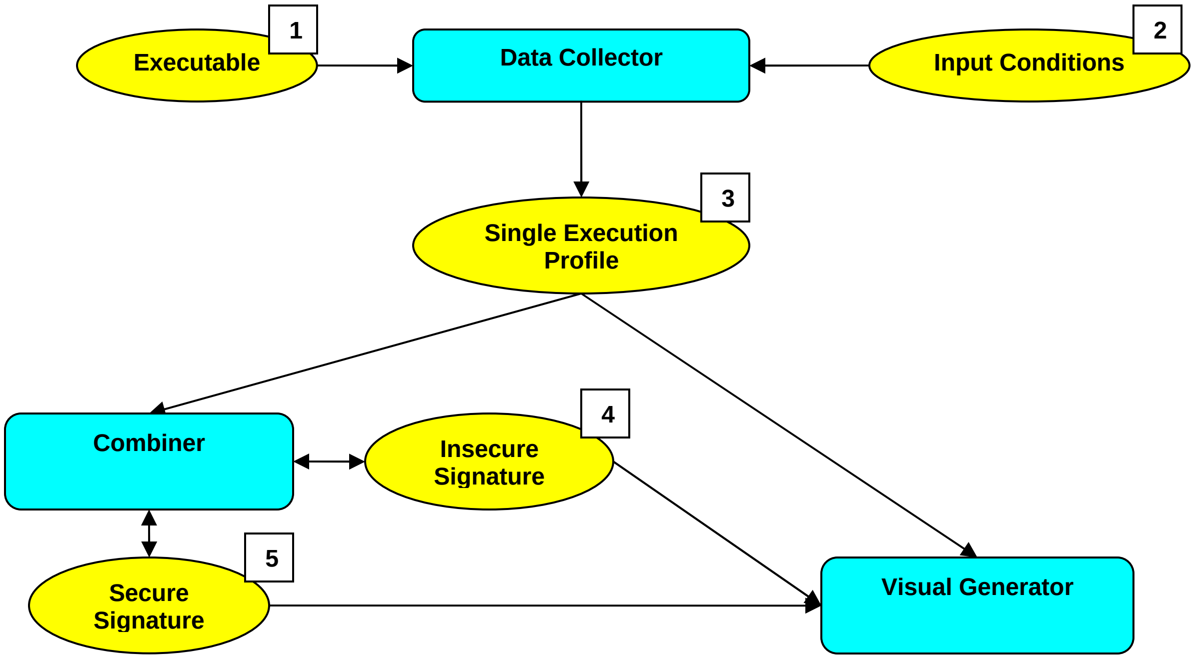


Figure 11: Component overview for approach 2.

A data collector, combiner, and a visual generator comprise the structure of the approach (Figure 11). The approach initially requires an executable program. To collect data related to memory, the executable runs on different sets of input conditions. For each set of input conditions, the data collector component records any data for memory utilization and stores the data in a file that represents a single execution profile (Figure 11-3). The visual generator uses the data file for the single execution profile to create a visual representation. The user then indicates if the representation they are viewing is secure or insecure. Next, the combiner takes the single execution profile and integrates it into either a file for a secure signature or an insecure signature. The visual generator can then render both secure and insecure signatures. The secure and insecure signatures then help facilitate vulnerability and exploit analysis in the future. A key issue for this approach is accuracy. The tool must assure the accuracy of signatures by generating signatures from a large number of execution profiles.

2.4.3 Approach 3: Controlled Environment

The objective of the controlled environment approach creates is to generate a visual representation of the memory utilization by a given exploit. The approach focuses on multiple programs that are vulnerable to the same type of exploit – buffer overflows for example. Execution of the vulnerable programs occurs within a controlled environment. A controlled environment is a layer between a program and the operating system and collects memory by intercepting requests for memory resources made to the operating system by a program. The approach combines all recorded utilization data into a single signature that represents a given exploit. The exploit signature can then be used to aide the understanding and detection of that exploit in the future. Structurally, the tool produced by this approach has three primary components: controlled environment, a combiner, and a visual generator (Figure 12).

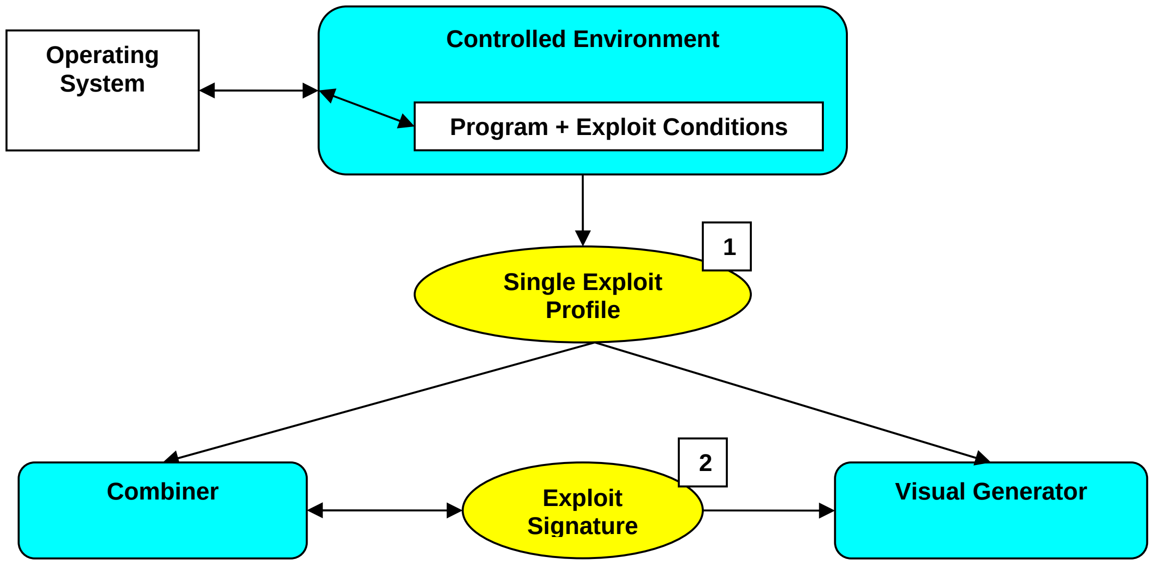


Figure 12: Component overview for approach 3.

Using the controlled environment approach initially requires one or many test programs that exhibit a vulnerability to the same exploit. The controlled environment component acts as a simulated operating system and processes any resource requests made by a program while quarantining the program from the rest of the system. The controlled environment takes a test program and automates its execution on a specific set of input conditions that exploit a vulnerability contained within the program. The controlled environment component collects data related to memory utilization and stores the data in a file representing a single exploit profile (Figure 12-1). The combiner automatically integrates new data files into the signature file (Figure 12-2). Finally, the visual generator creates representations of memory utilization by for both single exploit profiles and exploit signatures. Overall, the visual representation provides the user with a generalized behavioral view of the memory used by a given exploit. Users can then facilitate vulnerability and exploit analysis by viewing the visual representations.

2.4.4 Summary and Chosen Approach

As shown in Figure 10, Figure 11, and Figure 12, all three approaches share common design components. It is possible that implementation of each approach can reuse various components (e.g., the data collector). Furthermore, the approaches can be combined together to further facilitate the process of vulnerability and exploit analysis.

The visual generator is the only component to remain unchanged in all three approaches. Its main purpose is mapping a data file with a specified format into a visual representation of memory utilization. Implementation of the visual generator component can remain separate from the implementation of other components. Therefore, we can independently build and independently test the visual generator.

Although there is a data collector component present in all three approaches, the specifics of its functionality vary in each approach. The only uniform part of the data collector is the format it uses to record data files. Other than formatting similarities, the profiler must interact with different types of files and components for each approach. For the source code and compiler (approach 1), the data collector deals with a data file related to static memory, a binary, and various input conditions

The compiler component is unique to approach 1. Therefore, design of the compiler remains separate; once implemented, the compiler plugs into the overall system. Approach 1 is also a possible extension of approach 2 since generation of visual representation requires no information about source code.

Design of the controlled environment component remains independent from the design of other components. Although unique to approach 3, the controlled environment component can be added to approaches 1 and 2 as the way to execute a program. Furthermore, using the controlled environment would make data collection consistent across all three approaches.

Lastly, automation can be added to encapsulate all of the components. Since we know that approach 3 collects generalized profiles of exploits, and approaches 1 or 2 monitor a single program, automation can be added that compares the memory utilization of a program to "signatures" of know exploits. Theoretically, automation would enable users to visually observe the results of vulnerabilities and exploit analysis and enable them to gain additional understanding about memory utilization.

For the purposes of our research, we focus on designing a data collector component and the visual generator component. More specifically, we choose the scope of approach 2 that focuses on a program running in normal conditions. Furthermore, the scope of approach 2 is more convenient because it tests software in its natural environment rather than developing and verifying a controlled environment, as in approach 3.

In the remainder of this thesis, we discuss how to develop both a data collector (data capturing tool) and a tool that generates visual representation of memory utilization. The data-capturing tool focuses only on function calls made to the Application Programming Interface. The visual tool focuses only on the generation of visual representations of memory utilization and not on automatic analysis for vulnerabilities and exploits.

3 The Memory Model

The goal of this research is to generate a visual representation of memory utilization for the purpose of security analysis. However, to accomplish our goal we must understand the memory utilization that characterizes vulnerabilities and understand the memory objects and structures related to abnormal utilization of memory. Identifying abnormal utilization of memory requires the identification of normal memory utilization. Identification of utilization characteristics aids in our general understanding of how memory works and provides a base for formulating guidelines that direct the development of a visual tool.

Many different definitions can apply to the word "memory" within the context of computing. Hence, we need to differentiate between different definitions of memory and outline if they are important to our goals. We must understand the following memory-related terms: memory types, addressing methods, memory functions, function levels, and normal utilization of memory.

Section 3 presents a memory model for the standard Windows operating system. This section discusses the properties and characteristics related to physical memory, virtual memory, and process memory. Lastly, the section discusses examples of code that confirms or adds to the memory model.

3.1 Physical Memory

At the hardware level of a computer, three primary physical devices represent memory: Central Processing Unit (CPU) memory (registers and cache), Random Access Memory (RAM), and secondary storage. Each physical device has unique characteristics and purposes.

CPU memory is the fastest type of memory and has the smallest capacity. Registers and the cache are two separate types of memory contained within the CPU. Registers are extremely fast yet small memory segments — only a few bytes in some cases — that store values that a process frequently references during execution. Compared to registers, the cache is slightly slower and larger — up to several megabytes. Similar to registers, the cache holds frequently referenced data. The CPU uses both registers and the cache to avoid delays that occur if the CPU fetches data from external locations of memory.

The most versatile computer memory — with regard to size and speed — is RAM. Primarily, RAM stores temporary information required to run the operating system and other programs. Operating systems only use RAM for data storage and transfers data to the CPU for computations. RAM temporarily stores data, which is lost when the computer turns off.

Secondary storage is the third physical memory type. Secondary storage can come in many forms: hard drives, flash cards, floppy disks, CDs, and others. Unlike RAM, secondary storage permanently stores data. In certain cases in which the operating system fills its RAM, it may use some secondary storage as 'virtual RAM,' which is also known as scratch space. Secondary storage is primarily home to the file system of the operating system, which stores documents, system files, media files, etc. The file system exhibits a unique set of security problems, which puts it outside the scope of our work.

3.2 Virtual Memory

By definition, virtual memory is "an addressing scheme implemented in hardware and software that allows non-contiguous memory to be addressed as if it were contiguous" [Wikipedia 2007]. Usually, a typical computer contains between one and four gigabytes of physical RAM memory in addition to a given amount of CPU memory. However, the amount of physical memory has little correlation to the amount of virtual memory. An operating system can manage a virtual memory space that is larger than the available physical memory space. The operating system (OS) accomplishes this end by mapping addresses of virtual memory to physical resources. Furthermore, virtual memory enables various memory sources (RAM, registers, and cache) to have a unified address space. For example, a program running within the OS can utilize virtual memory without any information related to physical locations.

A standard 32-bit Windows operating system (2000, XP, Server) utilizes a default 4GB of address space for virtual memory [Microsoft Corp. 2007b]. Each process running within the OS utilizes a separate 4GB virtual address space, which means that processes can reference an identical address that the OS maps to two different physical locations. In the standard case, the 4GBs of address space contain two sections of 2GBs each (Figure 13-A). Other configurations are possible including a 3GB/1GB ratio (Figure 13-B). For the 2GB/2GB configuration, the lower addressed 2GBs (0x00000000 to 0x7FFFFFFF) contain data for programs provided to the user. The 2GB section in the higher address range (0x7FFFFFFF to 0xFFFFFFFF) — also called kernel space – exclusively contains data needed to run the OS.

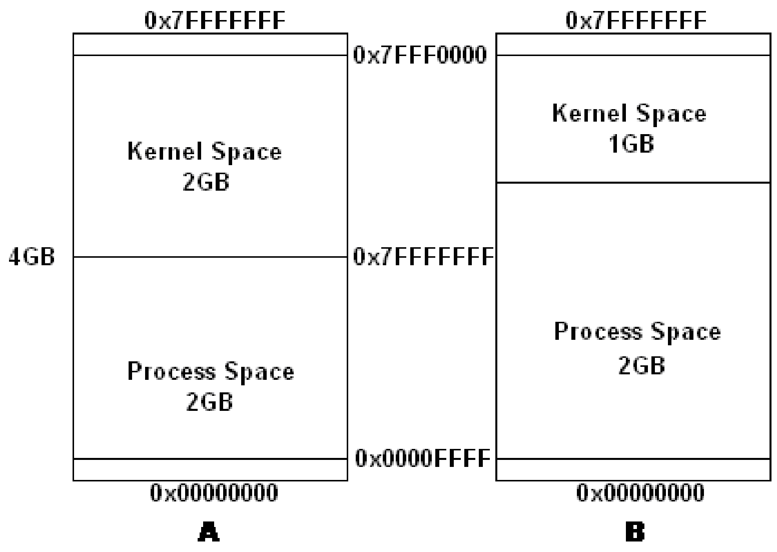


Figure 13: Examples of Memory Configuration for the Windows OS

Additionally, the OS never uses the addresses between 0x00000000 and 0x0000FFFF and between 0x7FFF0000 and 0x7FFFFFFF (Figure 13); this approach provides a buffer zone that helps prevent programs from referencing address locations outside the total address range.

Virtual memory is the focus of our research. By focusing on virtual memory, we can observe memory utilization while staying abstracted from a given process; this means that we observe utilization requests after a process makes them instead of monitoring a process directly. Processes do not track memory objects, such as buffers, as they relate to other memory locations, but observing virtual memory enables us to see how a process utilizes its entire memory space.

3.2.1 Virtual Addressing and Organization

The management of virtual memory by an operating system begins with addressing. Currently, most available computer systems either use 32-bit or 64-bit addressing. Our memory model focuses on only 32-bit memory systems, but many aspects of 64-bit systems correlate with 32-bit systems. To begin the virtual addressing process, Windows employs a structural system using tables and pages. The following components are involved in this process:

- Page: A page is a memory block of contiguous addresses; it is also called a page frame. In the case of the standard 32-bit computer system, a page contains 4096 bytes of memory [Microsoft Corp. 2007a].
- Page Table: The page table is a data structure maintained by the operating system that contains 1024 page table entries (PTEs) [Microsoft Corp. 2007a]. A PTE contains the start location of a specific page frame in physical memory. A separate page table group is assigned for each process running within the system.
- Page Directory: The page directory is a data structure kept by the operating system that contains 1024 page directory entries (PDEs) [Microsoft Corp. 2007a]. A PDE contains

the memory location of a given page table. As with page tables, each process running within the system is assigned a page directory.

The page and table organization of virtual memory means that a specific address value must represent more than a singular meaning. A 32-bit virtual memory address has three components: two 10-bit segments and a 12-bit segment (Figure 14). Each component within the address corresponds to a part of the page/table structure.

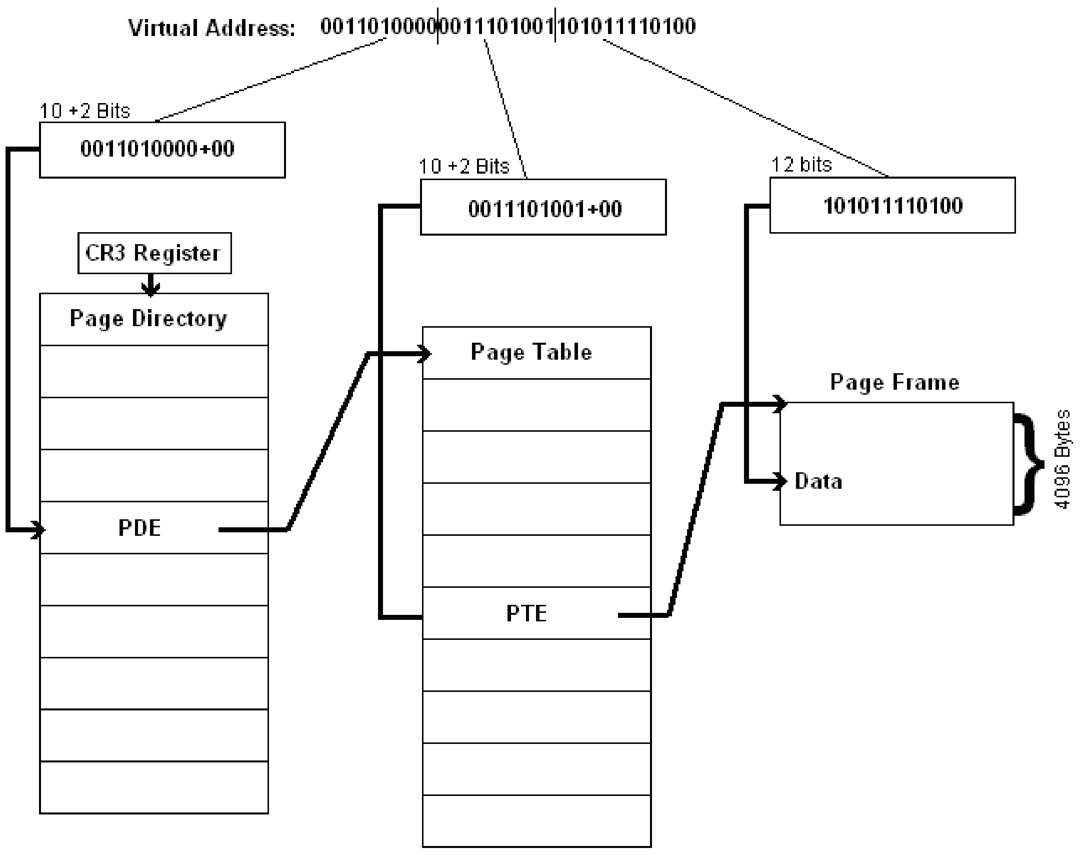


Figure 14: Virtual Addressing Example

Translating of these segments begins at the CPU's CR3 register. The CR3 contains a pointer to memory location of the page directory; any new address translation must begin at the CR3 register. After the location of the page directory, the following translation process occurs:

- Step 1: The first 10-bit segment actually represents a 12-bit segment. The OS expands the 10-bits to 12-bits by adding two low-order zero bits; 1011011000 would become 1011011000**00** for example. The resulting 12-bits represent an offset into the page directory [Microsoft Corp. 2007a]. The page directory entry located at the offset contains a pointer to a given page table.
- Step 2: The second 10-bit segment of a virtual address determines the offset into the page table found during Step 1. The OS adjusts the second 10 bit to be 12-bits by using the same method as in Step 1. The page table entry at the second offset identifies the start location of a particular page frame in physical memory.
- Step 3: The third segment represents the offset from the start of the page frame found in Step 2 to the location of the target data.

The 32-bit address has similar functionality to the mail system, where an address contains a state, street, and house number.

3.2.2 Virtual Memory Management Layers

Windows uses a hierarchy of functions to manage memory. Management functions affect the status of page frames or utilize the data contained in the frames. According to Windows documentation, a given page frame has one of three states: reserved, committed, or free [Microsoft Corp. 2007a]. Each state defines what operations the OS can apply to a given frame.

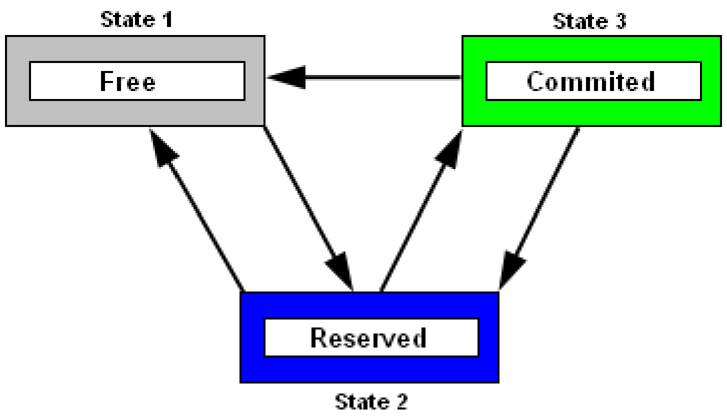


Figure 15: Page frame states and available transitions between them.

A Reserved status indicates that a process received a page, but no official allocation has occurred. Reserving memory makes sure that a process has enough memory available before it needs to use the memory. In other terms, a reserve status prevents other processes from utilizing a given memory page. However, reserved memory does not require a commit operation before the OS frees it. The status of a page changes to committed when a process allocates its reserved memory. Lastly, "free status" means that no process currently has rights to that page; the page is therefore available for reservation. To control the status of pages, the OS uses various memory management functions.

Overall, there are six main memory management function groups [Microsoft Corp. 2007a]. Each function group is located in either the NT Kernel level or the Win32 subsystem (Figure 16). Functions operate with different permission settings relative to the layer where they are located. Functions operating from the Win 32 Subsystem only have permission to modify a given virtual address space of a specific process. In contrast, functions operating from the NT kernel level have permission to modify all virtual and physical memory.

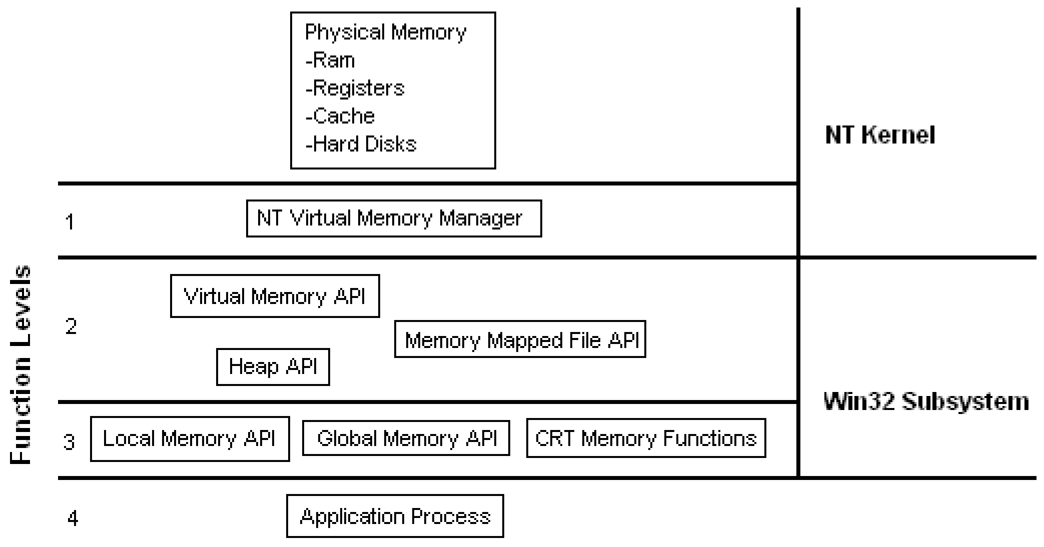


Figure 16: Virtual Memory Management Layers

Below are the descriptions of the six function groups responsible for Windows memory management:

- NT Virtual Memory Manager (NT VMM): As shown in Figure 16-1 — the NT VMM is the only group to reside in the NT Kernel level and has full permission to memory resources. The design of the NT VMM creates a bridge between the Win32 subsystem and the hardware level. In addition, the design is generic enough to work with various hardware models. The NT VMM prevents programs from illegally referencing other program memory or kernel level memory. Furthermore, it maintains the table-page structure discussed in section 3.2.1 for the entire system.
- Memory-Mapped File API: This function group manages hard disk locations that map into a virtual memory space. Memory-mapped files have their own unique properties, which puts them outside the scope of our research.
- Virtual Memory API: The function group for virtual memory is located in the Win32 subsystem and interfaces directly with the NT Virtual Memory Manager (Figure 16-2).

Most functionality provided by this API is similar to NT VMM functionally, except the two approaches enable different permission levels. Furthermore, the VM API is only responsible for the virtual address space of an individual process whereas the NT VMM is responsible for the entire address space of the system.

- Heap Memory API: The heap API provides the root functionality for the heap memory of a process. The heap is a type of memory structure that we discuss later in Section 3.3.2.2.
- Local & Global API: These groups of functions are similar to the Heap API but their design diversifies compatibility with various programs. Local and Global functions have identical functionality.
- CRT Memory functions: The CRT — or the C Runtime — function group is located closest to the user application level (Figure 16-3). Malloc and free are examples of well-known functions that reside in the CRT group.

As mentioned initially, these function groups are arranged in a hierarchical structure, as shown in Figure 16. The structure of the function groups is a result of development adding new compatibility and functionality over time. When programmers write code, they can choose to use older functions such as malloc or directly access new functions such as LocalAlloc. However, functions located in different groups do not act individually. For example, if a process calls the CRT function named malloc, the following would be a possible scenario:

1. Parameters pass to malloc from the user level program.
2. The resulting memory request passes to the Heap API, which manages the overall heap structure of a process.

3. The Heap API passes the request to the Virtual Memory API, which manages the overall virtual address space of a process.
4. Finally, the request passes to the NT Virtual Memory Manager, which performs the memory transaction at the hardware level and integrates the result into the virtual memory space of the OS.
5. Results and return parameters return through all of the functions previously mentioned to the user level process.

It is important to restate that the various function groups operate at different permission levels, i.e., the NT kernel versus the Win32 subsystem. Furthermore, levels have no information about anything happening at lower levels; hence, the Win32 subsystem is unaware of any functionality, data, or activity at the NT Virtual Memory Manager.

The function groups listed in this sub-section are a vital source of data for our research. By intercepting calls to memory management functions, we can reconstruct the memory activity related to allocate and free operations. Section 4 discusses more information on data capturing. So far, we have discussed how the view of memory differs between the NT VMM and Win32. Furthermore, we have discussed function hierarchy within both the NT VMM and Win32 and how memory operations interact with each other. Now we must discuss the memory view from the level above the Win32 — the process level.

3.3 Process Memory

To this point, we have examined physical memory, the virtual system memory, and virtual memory management provided by the Windows OS. However, the interpretation of memory from those levels is different from the interpretation made by a user process. When a user process requests memory, it has no information regarding the processing of the request or the final location in physical memory. Instead, a process only has information regarding address values and the structures within its own memory space. In this sub-section, we discuss virtual memory from the point of view of a user process.

3.3.1 Static Memory

The OS knows how much static memory a program requires before the program executes. The size and location of static memory remains fixed throughout the execution of a program. The constraints and assumptions in 2.1.1 also state that any data contained within static memory may not be monitored during program execution. The size and location of static memory may be fixed before execution, but modification of the data is allowed during program executes.

There are three primary components that form static memory: the data segment, block storage segment (BSS) [Bazaz 2006], and the text segment. The data segment contains all global variables that have constant declarations and an initialized value, "char C = 'x'" for example. On the other hand, the BSS contains all global variables that have non-constant declarations and no initializing, "char C;" for example. The text segment contains instructions from the binary that enable a program execution. Text space is read-only and illegal writes to it cause program failure [Harris et al. 2005].

Overall, the operating system enables greater protection of static memory than dynamic memory; for this reason, there are only two constraints and assumptions for static memory listed in Section 2.1.1. Furthermore, our research does not focus on static memory because of the limited vulnerabilities that it exhibits.

3.3.2 Dynamic Memory

As opposed to static memory, the OS has no information about the allocation requirement of dynamic memory before the execution of a program. According to Bazaz's taxonomy, dynamic memory is the most vulnerable memory type. Because dynamic memory has a high risk of vulnerability it becomes a primary focus of this research. One of two memory structures guides all utilization of dynamic memory by a process: the stack or the heap. Although the stack and heap are fundamentally memory, each has different usage and behavior.

3.3.2.1 Stack Memory

Often people classify stack memory as static memory because a program uses the stack to store static data within its code. However, because the structure of the stack, varies it is therefore dynamic memory. After loading a program for execution, the operating system assigns the process a memory section for its stack space. Windows can only allow a process a stack space of limited size; if a process tries to allocate more than the maximum amount it abnormally terminates. Refer to section 3.4.1 to find program examples that overload the stack.

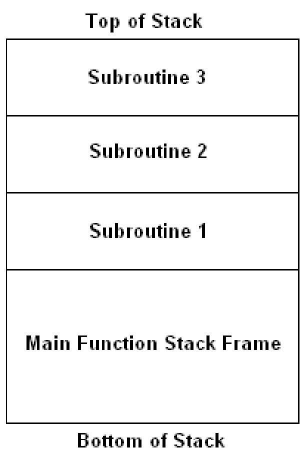


Figure 17: Stack Example

The structure of the stack defines its operational characteristics. The stack operates by adding new items to the top of the stack; only items currently at the top may be removed (Figure

17).This method is more commonly known as 'last-in-first-out'. Generally, the first item to go into the stack is the information about the program's main function, which is called a stack frame. The stack frame contains the local variables, parameters, and the return address of a given function; it is also defined as an activation record [Wikipedia 2007]. A new stack frame is added to the stack each time a program calls a subroutine; then when a function has completed and returns to the caller, the stack frame is removed.

Within each stack, there are two information types: variable and non-variable. Variable data is any data needed by the process to execute the functionality within its code. Examples of variable data include integers, arrays, strings, and characters. In contrast, the OS requires non-variable data to run a program and not for the code functionality of a program. A return address is an example of non-variable data.

- Return Address: When a function calls a subroutine function, the return address indicates the point in the code of the calling function found directly after the call. The operating system uses the return address to continue the execution of a function once a subroutine completes.

During runtime, the assumption is that the functionality of a program cannot modify non-variable data. Our research makes a distinction between variable and non-variable because many exploits illegally attempt to alter non-variable data.

3.3.2.2 Heap Memory

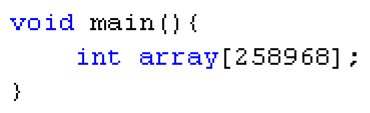
The Heap is a memory block that contains all dynamically allocated data requested by a program. Functions such as malloc and new are typical functions that utilize heap memory; for example, data declared by "int* array = malloc int[10]" goes to the heap whereas "int array[10]" goes to the stack. At the first request for heap memory, the system reserves a group of page frames, each of which is typically 4096 bytes in size. Further heap requests allocate memory from the memory initially reserved by the system. If there is not enough memory currently in the heap for a request, the system sets aside more pages [Microsoft Corp. 2007a]. When a program deletes blocks from the heap, the address range that they occupy simply becomes available again without affecting the overall size of the heap. Due to the nature of dynamic program data, allocations and deletions within the heap can cause memory fragmentation. If fragmentation occurs, the system attempts to fit new allocation requests into the first available location. If a new allocation cannot fit into any gap, the system must reserve new blocks for the heap even if the sum total of available memory is sufficient.

3.4 *Confirming the Model*

Up to this point, we have discussed memory from the physical level up to the process level. Now, we present examples of code that highlights and confirms elements of the memory model. Furthermore, some of our examples provide new information about the memory model that is not clearly visible in present literature.

3.4.1 Maximum Stack Size Limitation

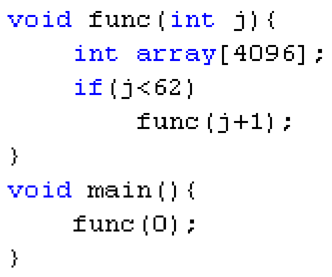
The Windows 32-bit OS limits the total stack space that it can assign to a single process. To show that a limit exists and to get an estimate of what the limit is, we must intentionally overload the stack. The code in Figure 18 allocates just enough integers to overload the maximum allowed stack space – roughly 2GB.



```
void main(){
    int array[258968];
}
```

Figure 18: Stack Overload

Figure 18 uses static data to overload the stack, but adding too many activation records can also force an overload. We know that any subroutine gets its own activation record on the stack. Typically, the system removes the activation record when the subroutine completes. However, when there are recursive calls to a subroutine function, the system continuously adds activation records until the reversal of the recursion. The example in Figure 19 shows that the stack overloads if too many recursive function calls occur in succession.



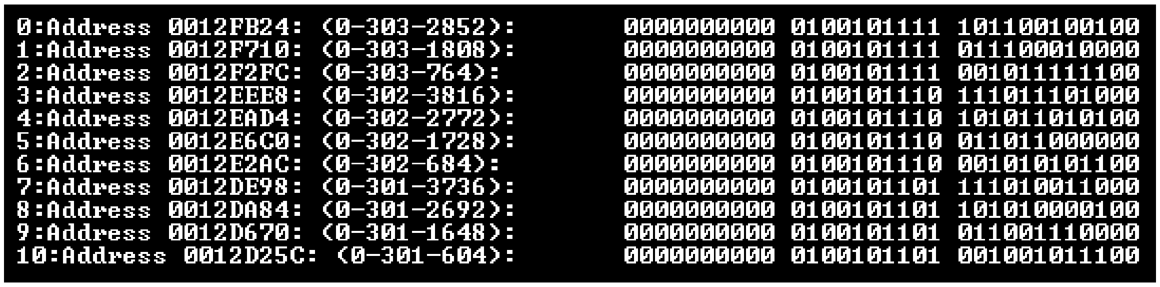
```
void func(int j){
     int array[4096];
     if(j<62)
          func(j+1);
}
void main(){
     func(0);
}
```

Figure 19: Recursive Stack Overload

Each new function call adds another activation record onto the stack; because each call does not return, the stack eventually overloads when roughly 2GB of integers are allocated.

3.4.2 Stack Addressing

A program similar to the one shown in Figure 19 generates the output below that shows the data locations contained in recursively called functions. The output shows start addresses of statically allocated arrays within each recursive function. The program displays addresses as pointers and then separates the addresses to show the three elements of a 32-bit address. The offset values are all separated by equal memory distances — 3052 bytes in this case.



```
0:Address 0012FB24: (0-303-2852):        0000000000 0100101111 101100100100
1:Address 0012F710: (0-303-1808):        0000000000 0100101111 011100010000
2:Address 0012F2FC: (0-303-764):         0000000000 0100101111 001011111100
3:Address 0012EEE8: (0-302-3816):        0000000000 0100101110 111011101000
4:Address 0012EAD4: (0-302-2772):        0000000000 0100101110 101011010100
5:Address 0012E6C0: (0-302-1728):        0000000000 0100101110 011011000000
6:Address 0012E2AC: (0-302-684):         0000000000 0100101110 001010101100
7:Address 0012DE98: (0-301-3736):        0000000000 0100101101 111010011000
8:Address 0012DA84: (0-301-2692):        0000000000 0100101101 101010000100
9:Address 0012D670: (0-301-1648):        0000000000 0100101101 011001110000
10:Address 0012D25C: (0-301-604):        0000000000 0100101101 001001011100
```

Figure 20: Recursive Call Addressing

The above output shows the utilization of three page tables to address all of the arrays. Furthermore, the addressing values show that the stack grows from high to low addressing, which means that the top of the stack is a lower address that the bottom.

3.4.3 Hierarchy of Memory Management Functions

An API hook — a method discussed in Section 4 – captured the following data sample. The data was captured from a typical program; it shows that when a program calls malloc then it also calls HeapAlloc directly afterwards with the same parameters. The same thing happens with the call to free. This example illustrates the nested interaction between the various function groups.

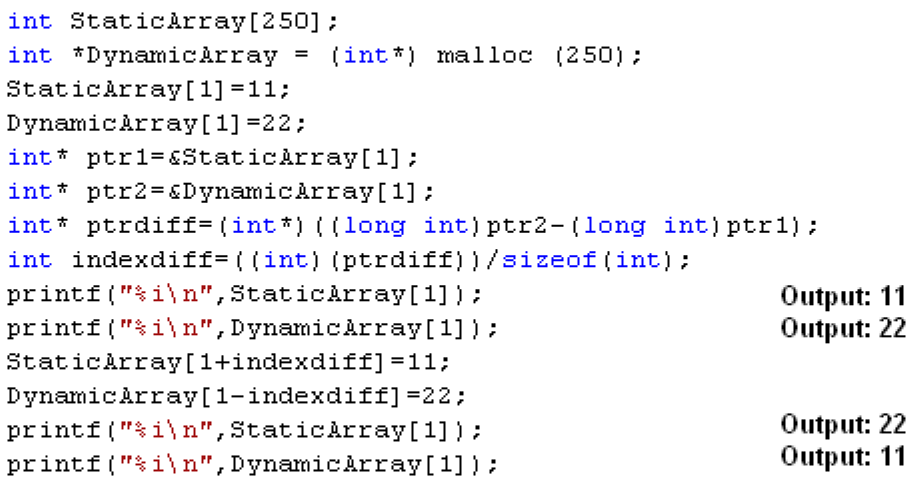
```
malloc(ntdll)  4599144    4599272    128   Success
HeapAlloc      4599144    4599272    128   Success
free(msvcrt)   4599144    4599272    128   Success
HeapFree       4599144    4599272    128   Success
```

Figure 21: API Hierarchy

Running similar tests shows the interaction among other API levels; other examples allocation functions include HeapAlloc calling VirtualAlloc, LocalAlloc calling HeapAlloc, and VirtualAlloc calling NtAllocateVirtualMemory.

3.4.4 Process Memory Permission Rules

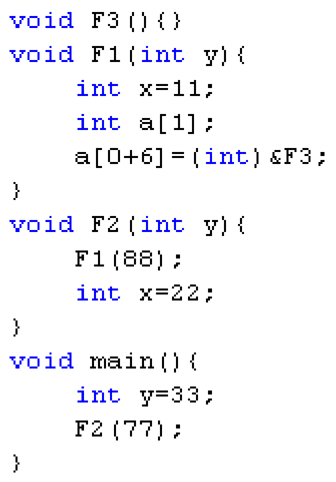
Permission rules are an aspect of process memory that we have not discussed up to this point. A process can read and write to any address within its allocated address space at any time during runtime. Under normal circumstances, the code that governs the functionality of a program restricts reads and writes to appropriate addresses. For instance, the assumption is that a process modifies an array within the bounds of that array. Logically, however, we can write code that references abnormal addresses of the array. For example, the code in Figure 22 utilizes pointer arithmetic to reference index locations outside the defined bounds of the array. More specifically, the code indexes a static array to modify the contents of a dynamic array, and vice versa.



```
int StaticArray[250];
int *DynamicArray = (int*) malloc (250);
StaticArray[1]=11;
DynamicArray[1]=22;
int* ptr1=&StaticArray[1];
int* ptr2=&DynamicArray[1];
int* ptrdiff=(int*)((long int)ptr2-(long int)ptr1);
int indexdiff=((int)(ptrdiff))/sizeof(int);
printf("%i\n",StaticArray[1]);                    Output: 11
printf("%i\n",DynamicArray[1]);                   Output: 22
StaticArray[1+indexdiff]=11;
DynamicArray[1-indexdiff]=22;
printf("%i\n",StaticArray[1]);                    Output: 22
printf("%i\n",DynamicArray[1]);                   Output: 11
```

Figure 22: Testing Process Memory Permissions

In section 3.3.2.1 we make a distinction between variable and non-variable data. During normal operation, the OS needs non-variable data for the internal operations of a process; this data is not intended to be accessible by user code. However, the OS allows modification of the non-variable data because it is within the overall address space of a process. The code in Figure 23 manually overwrites the return address of one sub-routine in the stack space with another return address.



```
void F3(){}
void F1(int y){
     int x=11;
     int a[1];
     a[0+6]=(int)&F3;
}
void F2(int y){
     F1(88);
     int x=22;
}
void main(){
     int y=33;
     F2(77);
}
```

Figure 23: Writing to Non-Variable Data

Overwriting the non-variable data in the above code causes the program to crash. The crash occurs because the data overwrite irreversibly disrupts the stability of the program. However, if a skilled hacker alters the return value to point back to executable shell code then the program runs the illegal code. Overwriting a return address, which is also known as a buffer overflow, is one example of a memory model violation that we would like to detect with a visual representation.

4 Capturing Data Related to Memory

The previous section presents a memory model that identifies how programs use memory. Our end goal is to generate a visual representation of the memory model. However, visualization is a data-driven task, so we must define a process that produces complete and accurate data; if the data is insufficient then any visualization will be insufficient. Our overall process of collecting memory data for visualization has four steps:

1. *Step 1: Identifying the required data.* To begin, we must identify exactly what data is required to generate complete and accurate visual representations of the memory model. More specifically, this step identifies the type of data, such as addresses, sizes, or other parameters. This step only identifies the minimum data types required for visualization. The minimum requirement of data identifies the data that is avail to visualization; missing this data guarantees the failure of the visualization.
2. *Step 2: Locating the data.* After identifying the data we need, we must identify the location of the data. In this case, the locations relate to the level of memory model: physical, virtual, or process.
3. *Step 3: Obtaining the data.* Once we locate the data, we have to capture it. Fortunately, there are many methods of data capturing available. Unfortunately, no single capturing method has both a simple implementation and is capable of facilitating the capturing of all the data we require. We only implement one capturing method for this research because data capturing is not the primary goal of our research; future capturing methods can be added to supplement the method we choose. This section will also discuss the method we choose in detail.

4. *Step 4: Formatting the data.* During the final step, we format the data. Formatting enables the visualization tool to process the data directly rather than waste computational resources to initialize the data. A defined format also enables future capturing methods to integrate with our visualization tool.

The remainder of this section discusses the following: (1) Step 1, which identifies the data, (2) Step 2, which locates the data, (3) possible methods to capture the data, (4) details about the data we focus on capturing, (5) the implementation of our capturing method, and (6) Step 4, which formats the data.

4.1 Types of Memory Data

Identifying the data required for generating a visual representation is the first step of our data collecting process. Section 3 discusses many functions and structures with varying behavior and functionality. However, any memory operation does one of two core tasks: (1) modify the memory space via allocations, deletions, resizing, and relocations or (2) access the data contained within the memory structure via read and write operations.

Two fundamental operations alter the memory structure: allocations and frees. An allocation has the same fundamental functionality regardless of whether it requests memory from the heap or the stack. In addition, the core functionality of an operation is the same for memory commits, reserves, de-commits, releases, or any other variation of allocate and free operations. Each allocation and free has the following basic components: start address, requested size, and success status; without these components, allocation and free operations would not function regardless of additional parameters.

Operations that access the data stored in memory are either reads or writes. Programs may perform complex memory operations such as comparisons or additions, but they simplify to series of read and write operations. For example, an addition operation separates into two reads and a write. A read and write operation has the following components: a start address, size of the data to access, a data buffer, and success status.

Separating memory operations into core components has several advantages. First, it identifies the “bare minimum” data requirement that represents the memory model accurately. Second, it separates the vital data from secondary data; future enhancements can supplement the visualization using the secondary data. Lastly, it highlights the information we must capture accurately. Inaccurate capturing of the vital data components introduces flaws into the visual representation of the memory model.

4.2 Data Sources

Now that the vital data is identified, we must identify the location of the data. Referring to the memory model in section 3, there are three primary memory levels containing memory data: physical, virtual, and process.

Data located at the physical level is extremely complex; operations separate into many simple operations by the time they reach the physical level. This approach greatly magnifies the potential size of the data set, and subsequently, increases the complexity of converting the data into a visual representation. It is important to note that raw data from the physical level is impractical to analyze because exploits do not focus on this level.

Virtual memory is a better data source than physical memory for the purpose of detecting vulnerabilities and exploits; this is because exploits typically execute from the virtual or process levels. Furthermore, the virtual layer is more accessible for capturing methods than the physical level. Data related to allocation and free operations from this level comes from the virtual memory management functions discussed in section 3.2.2. However, the virtual level has limited data related to read and write operations.

The third possible location of memory data is the process level. Logically, the process level is the best choice for capturing read and write information because any memory operation originates from this level; however, security permissions limit the flexibility of implementing capturing methods at this level.

4.3 Possible Methods to Capture Memory Data

After locating the data, we now identify possible methods that capture the data. More specifically, this section identifies various methods as they relate to the physical, virtual, or process level.

Capturing memory information from the physical layer is the most difficult option to implement because the operating system protects physical memory from user level processes. However, several possible methods are as follows:

- *Method 1: Designing highly specific memory hardware with automatic capturing integrated into the hardware.* Implementing new hardware would obviously be an extremely specialized and high-end task. Furthermore, this option is well beyond the scope of this research.
- *Method 2: Implementing specialized device drivers for memory hardware.* A device driver has the ability to capture data as it travels from the operating system to the hardware. However, the problem is that device drivers are very high-end software and could alter the natural behavior of the computer or cause system instability if improperly implemented.

- *Method 3: Capturing the assembly instruction stream going to the CPU during runtime.* This approach requires hardware alterations to CPU-related components. The major problem with this method is that it requires complex and specialized hardware. Furthermore, this method would cause an extreme decrease in the computational speed of the CPU.

In contrast to the physical level, methods that capture data from the virtual level are more numerous and easier to implement. Within virtual memory, data capture can focus on the memory space directly or focus on the functions that manage the memory space. Our research focuses on the management function to obtain memory data. The problem is that user processes do not have default permission to access functions that manage virtual memory. Possible methods for capturing data from management function are as follows:

- *Method 1: Rootkit implementation.* Rootkits (section 2.3.1) operate with kernel level permission (just as the operating system does), which allows them access to all memory management functions. Unfortunately, rootkits are difficult to implement and existing ones are difficult to customize. Furthermore, rootkits have a high risk of causing system instability.
- *Method 2: Implementation of function hooks.* Function hooks (discussed in section 2.3.2) capture data from individual API functions that we select. More importantly, function hooks have more stability than rootkits. We choose to implement this method to capture the data required for the allocation and free part of the visualization (details for function hooking are in section 4.4.3).

The third option is to capture data from the process level. In this case, the data capture occurs at the same level as user processes. However, the operating system protects processes from each other with security permissions. Possible methods for capturing data from the process level are as follows:

- *Method 1: Specialized Compiler*. This custom component captures static allocation and free data information (3.3.1).
- *Method 2: Controlled Environment.* A virtual runtime environment is another method to capture memory data from the process level. A controllable environment acts as a communication layer between the process and the operating system; furthermore, the process does not know the difference between the environment and the operating system.

Other than compilers and controlled environments, capturing data from the process level is limited because capturing from this level has a high risk of affecting the natural memory utilization of a given process; for example, altering a program to capture its own information would add unnatural functionality and memory usage.

4.4 Chosen Data and Capturing Method

As discussed in the previous sections, there are several options available for the implementation of data capturing. However, we only implement one capturing method because our overall focus is a visualization tool and not techniques for data capture. Capturing calls to API functions is the method we implemented to generate data for our visual representation. Unfortunately, this method only captures data related to allocation of free operations. To discuss the method we choose to implement, the following sub-sections overview three topics: (1) the application programming interface, (2) key API functions, and (3) the implementation of the hooking method. These three topics encompass both the location of memory data within the virtual level and the specific method that we used to capture the data.

4.4.1 Overview of the Application Programming Interface

An API — or application programming interface — provides an interface for programs to request resources from the operating system (OS) while keeping user programs quarantined from the OS [Microsoft Corp. 2007a]. For example, a heap API function is not responsible for implementing physical memory allocation; rather the API function passes the request to lower-level operating system functions that handle the physical allocation.

Each API represents a group of functions that relate to a similar category. Windows has seven API categories: base services, graphics device interface, user interface, common dialog box library, common control library, windows shell, and network services [Microsoft Corp. 2007a]. Out of the seven, the base services and user interface categories contain the majority of API functions for memory resources.

In section 3, we only discuss the functional layout of the Windows virtual memory management API; however, section 3 does not cover the location of the API. In Windows operating systems, information related to API functions is located in dynamic link libraries (DLL). The following describes the properties of a DLL [Microsoft Corp. 2007a]:

- Dynamic Link Library (DLL): A DLL is a shared function library that is accessible to all processes running in the OS.
    - A DLL contains two function types: exported and internal. Both processes and other DLL functions can call exported functions. Other DLL functions can only call internal functions. A process may not directly call certain functions because they are only accessible from within a DLL.
    - Only one instance of a DLL exists in memory at a given time. After one DLL loads for one process, other processes access it without loading a new copy of the

DLL. Each DLL maps from system memory into the virtual address space of multiple processes. Mapping to multiple processes allows each process to function as if it has a separate copy of a given DLL. However, functions within the DLL assume they are located within the memory space of a process; this means that the function does not alter the system memory where it is located, but rather alters the address space of the calling process. Finally, the DLL unloads from memory only when all processes finish utilizing it.

- Security permissions protect the contents of a DLL to prevent modification of API functions.
- Some API functions — such as malloc — have code definitions within several different DLL files. This quality is especially important to the data capturing for our research because by capturing only one version of a function, we may miss the version actually invoked by a process.
- Certain functions — particularly from the CRT group — are located outside a DLL; this occurs when a function statically links when building a program. Statically linked functions are contained within .lib files whereas dynamic linked functions come from DLLs. Static links enable an executable to have its own private copy of a function within its binary. As a result, the capturing of calls to statically linked functions cannot occur at the memory management API layer.

At this point, we know what API groups to target and where to locate them. Now we must identify exactly what specific functions we need to target in order to capture the data required for visual representation of memory utilization.

4.4.2 Overview of API Functions Related to Memory

Before data capturing begins, we must identify API functions that are relevant to memory allocation and free behavior. This sub-section identifies key memory functions categorized by their API group (the full list of identified functions is in Appendix A). More specifically, we identify the name, parameters, return value, and functionality of each function.

4.4.2.1 NT Virtual Memory API Manager

As discussed in section 3.2.2, the NT virtual memory manager operates between other API groups and the physical level. The NT API is the most protected layer in the operating system. No process should interact directly with the NT layer. Unfortunately, the lack of direct interaction also results in a lack of published documentation. However, thanks to NTinternals.net there is partial documentation of NT API functions [NTinternals.net 2006].

- NtAllocateVirtualMemory
    - Parameters: IN HANDLE ProcessHandle, IN OUT PVOID *BaseAddress, IN ULONG ZeroBits, IN OUT PULONG RegionSize, IN ULONG AllocationType, IN ULONG Protect.
    - Return Value: NTSYSAPI NTSTATUS NTAPI
    - The function allocates a region starting at the address indicated by *BaseAddress* of the size indicated by *RegionSize*. If *BaseAddress* is zero then the function defaults to finding the first best fit inside the virtual address space. The value of *AllocationType* defines if the allocation is committing or reserving memory.
- NtFreeVirtualMemory
    - Parameters: IN HANDLE ProcessHandle, IN PVOID *BaseAddress, IN OUT PULONG RegionSize, IN ULONG FreeType.

- Return Value: NTSYSAPI NTSTATUS NTAPI
- This function frees a memory region starting at *BaseAddress* of size *RegionSize*. If *RegionSize* is not provided, the entire region at *BaseAddress* is freed. The value of *FreeType* indicates if the memory operation is a release or de-commit.

4.4.2.2 Virtual Memory API

The virtual memory API documentation is provided by Microsoft [Microsoft Corp. 2007a].

- VirtualAlloc
  - Parameters: in LPVOID lpAddress, in SIZE_T dwSize, in DWORD flAllocationType, in DWORD flProtect
  - Return Value LPVOID
  - This function allocates a virtual memory block of size dwSize starting at address *lpAddress*. The allocation type can be equal to MEM_COMMIT, MEM_RESERVE, or MEM_RESET. MEM_COMMIT — valued 0x1000 — specifies the function to commit a block of free or reserved memory. MEM_RESERVE — valued 0x2000 — indicates the function is reserving a block of memory that is currently free. MEM_RESET — valued 0x8000 — instructs the function that the process does not currently need the committed memory block, but wants to keep the block reserved for future use. When the function completes, it returns the start address of the requested memory block.
- VirtualFree
  - Parameters: in LPVOID lpAddress, in SIZE_T dwSize, in DWORD dwFreeType
  - Return Value: BOOL

    - This function reverses the functionality of VirtualAlloc by freeing a block of size *dwSize* that starts at address *lpAddress*. If the value of *dwFreeType* is MEM_DECOMMIT — 0x4000 — then the function de-commits the memory block but keeps it reserved for future utilization by a process. If the value is MEM_RELEASE — 0x8000 — then the function frees the entire block, starting at *lpAddress*, back to the system. The function returns a "false" value if any parameters are invalid or if it is attempting to free unallocated memory.

4.4.2.3 Heap API

The heap memory API documentation is provided by Microsoft [Microsoft Corp. 2007a].

- HeapAlloc
    - Parameters: in HANDLE hHeap, in DWORD dwFlags, in SIZE_T dwBytes
    - Return Value: LPVOID
    - This function is responsible for allocating a memory block of size *dwBytes* from the heap structure, which is identified by *hHeap*. If a heap memory block is successfully allocated it cannot be moved until it is freed. If the function succeeds, it returns the starting address of a block; otherwise, it returns a NULL pointer.
- HeapFree
    - Parameters: in HANDLE hHeap, in DWORD dwFlags, in LPVOID lpMem
    - Return Value: BOOL
    - This function frees a block memory starting at address *lpMem* from the heap structure identified by the *hHeap* handle. The targeted block is not released back to the system, but returns to the total available memory within the heap.

4.4.2.4 Local & Global Memory API

Local and global memory functions are considered obsolete functions because they do not provide as much functionality as the before-mentioned heap functions. However, older programs still utilize them in certain cases. Equivalent local and global functions have identical functionality. Microsoft [Microsoft Corp. 2007a] provides the documentation for local and global API.

- LocalAlloc & GlobalAlloc
    - Parameters: in UINT uFlags, in SIZE_T uBytes
    - Return Value: HLOCAL
    - This function allocates a memory block of size *uBytes*. If the allocation succeeds, the function returns a handle to a new memory object.
- LocalFree & GlobalFree
    - Parameters: in HLOCAL hMem
    - Return Value: HLOCAL
    - This function frees the memory contained within the memory object, which is represented by the *hMem* handle. If the function succeeds, it returns a NULL handle.

4.4.2.5 CRT Memory API

As discussed in section 3.2.2, the CRT API layer operates between the other API levels and the process level. CRT memory API documentation is provided by Microsoft [Microsoft Corp. 2007a].

- malloc
    - Parameters: size_t size

    - Return Value: void pointer
    - A process calls malloc to request a memory block from its heap. If the allocation is successful, the function returns the pointer to the requested block. The function returns a NULL pointer if there is no memory available for allocation.
- free
    - Parameters: void *memblock
    - Return Value: None
    - This function frees a memory block — specified by the pointer *memblock* — that malloc has previously allocated.

4.4.3 Overview of Inline Hooking

After identifying the required data, we can now identify a method to capture the data. For this research, we choose to capture memory management API by using the *inline hooking* process. This sub-section discusses the details of inline hooking and the implementation of the function hooks.

By definition, hooking is a chain of procedure calls designed for event handling [Wikipedia 2007]. Without hooking, a typical chain of calls consists of only two parts: a source function and target function. For example, a subroutine (the source) calls the malloc function (the target) to allocate memory. The term 'inline' indicates that the function hook occurs in the call chain between the source and target function. The entire inline hooking process involves four objects: source function, the hook, trampoline function, and target function.

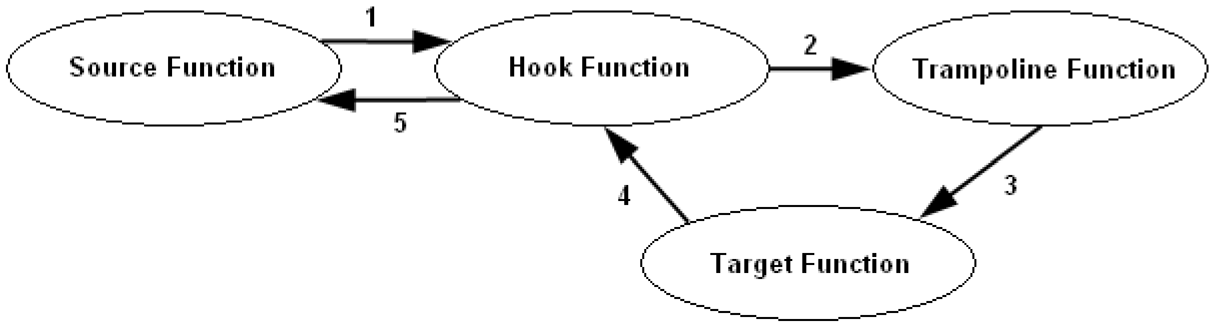


Figure 24: The Inline Hooking Instruction Flow.

These objects form a call chain consisting of five steps (Figure 24). The descriptions of the hooking objects are as follows:

- Source Function: The source function is the function that initially calls another function, i.e., any instance of one function invoking another. The source function can be a subroutine in a running process or another API function.
- Hook Function: The hook function contains any extra functionality we wish to add to the functionality of the target function. In this case, we add code that records information about the parameters and return values relevant to reconstructing the memory model. The source function directly calls the hook function. Then, the hook function calls the trampoline function to invoke the original target function. The final functionality occurs when the target function returns parameters to the hook function. Hook functions are not unique to an inline hooking procedure and are used by other hooking methods. Normally, developers use hook functions as diagnostic tools. However, rootkits and other high-end hacking tools use hooks to access the lower levels of the operating system.
- Trampoline Function: By definition, a trampoline function 'bounces' a call to another function [Wikipedia 2007]. In the case of an inline hook, a trampoline function for a given target function contains two parts. First, it contains the first few instructions of the

target function — specifically, the first five bytes (Figure 25). Second, the trampoline contains a jump instruction to the code of the target function starting after the initial five bytes [Hunt and Brubacher 1999]. The hook function invokes the trampoline function and uses it to pass parameter information from the hook function to the target function.

- Target Function: The target function is a given API function that is hooked. During an inline hooking procedure, the target is unaltered except for its first five bytes; these bytes are replaced with code that causes the invocation of the hook function.

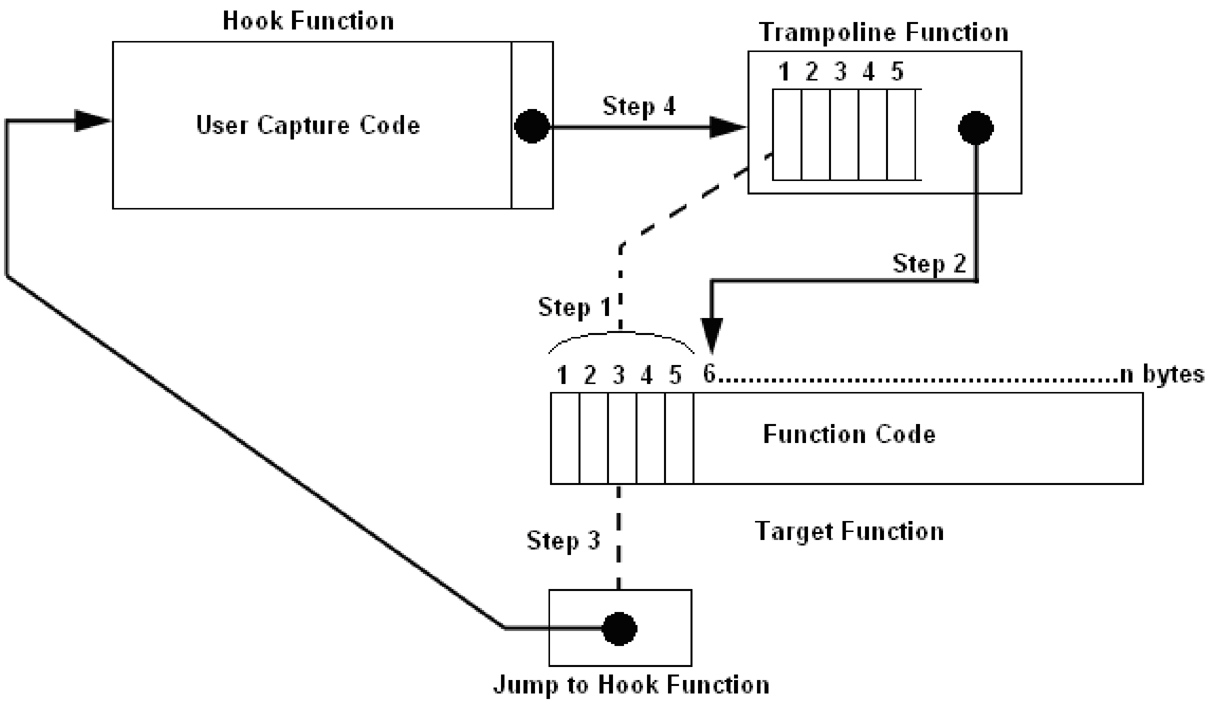


Figure 25: Setup Steps for an Inline Hook.

The implementation of an inline hook separates into five steps; four of these steps are shown in Figure 25.

1. Step1: This step copies the first five bytes of the target function into the code of the trampoline function. The copy insures that the original code of the target function is not

lost or improperly altered. The operating system could become unstable without saving the original code of the target function.

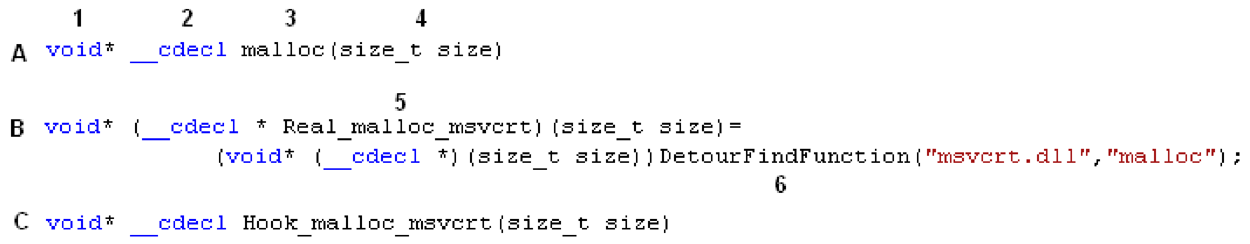
2. Step 2: This step adds a jump command to the end of the trampoline function that references the sixth bit of the target function; this means that there is no interruption of the normal functionality of the target when the hook function invokes the trampoline function.
3. Step 3: This step overwrites the first five bytes of the target function with a jump instruction that points to the start of the hook function.
4. Step 4: This step adds a jump instruction to the hook that points to the trampoline function.
5. Step 5: The final step (which is not shown in Figure 25 undoes all modification of the target function and the trampoline function when the hooked program terminates. The step can cause instability of the operating system if done improperly.

Once the hook is in place, the instruction flow between a source and target function has been altered to a call chain of five steps (Figure 24).

4.4.4 Implementation of API Function Hooking

The Detours 2.1 toolkit provides the source code that implements the process in Figure 25 (provided by the Microsoft research department [Hunt and Brubacher 1999]). Detours is a function library that allows users to design their own hooking programs without having to implement the low-level steps of the inline hooking process. The Detours package also includes several sample programs that implement hooks on various functions. For our research, we utilize one of samples to implement the hooking process; all we must provide are the target function definitions and hook functions for the memory API we wish to capture.

The first thing that Detours requires is the exact target function definitions (code for all function definitions can be found in Appendix B); this includes providing the exact name (Figure 26-3), calling convention (Figure 26-2), parameters (Figure 26-4), and return value (Figure 26-1). The hook fails if any value is incorrect.



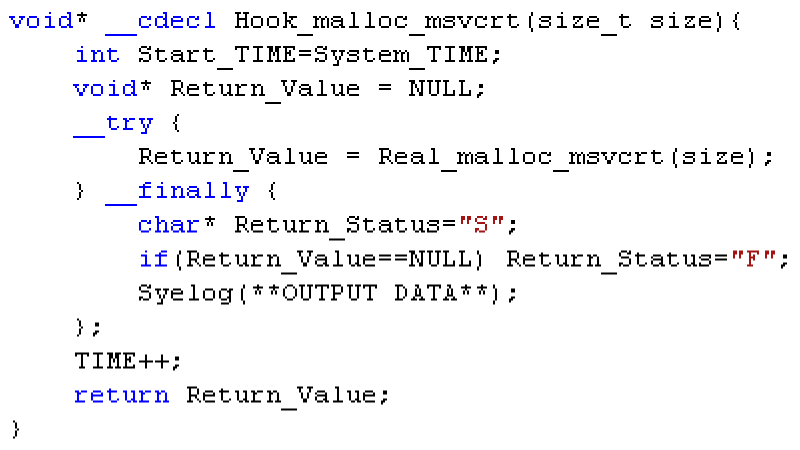
```
    1        2        3          4
A void* __cdecl malloc(size_t size)

                               5
B void* (__cdecl * Real_malloc_msvcrt)(size_t size)=
              (void* (__cdecl *)(size_t size))DetourFindFunction("msvcrt.dll","malloc");
                                                  6
C void* __cdecl Hook_malloc_msvcrt(size_t size)
```

Figure 26: Example of Function Definitions for the Hooking Process

As shown in Figure 26-A, the process starts with a function to hook. Next, indicating the address location of the target function to Detours happens by two methods.

- Method 1: If the location of the target function is unknown, then the Detours function named “DetourFindFunction” locates the needed address when given a proper function name and the parent DLL (Figure 26-5).
- Method 2: If the location of the target function is known though an include file or build-time link, then the line in Figure 26-B is set equal to the function name — HeapAlloc for example. The hook fails or crashes the program if Detours cannot find the function or if the provided definition is incorrect.

Finally, we must provide functionality for hook functions corresponding to each target function. An example hook is shown below in Figure 27 (code for all functions hooks is found in Appendix C). Aside from the function name, the hook has the same call convention, return value, and parameters as the target function (Figure 26-5) If these values differ from the source the hooking process fails. As shown in the example code, the Syelog function does the actual data recording; another Detours sample program named syelog.exe provides this function. The syelog program runs as a secondary process to the target executable. When the Syelog function runs, it passes the output parameters to the syelog program for processing to a file (Figure 28); this procedure prevents the I/O overhead from running within the hooked executable. Section 4.6 discusses the format of the data sent to syelog.



```
void* __cdecl Hook_malloc_msvcrt(size_t size){
    int Start_TIME=System_TIME;
    void* Return_Value = NULL;
    __try {
        Return_Value = Real_malloc_msvcrt(size);
    } __finally {
        char* Return_Status="S";
        if(Return_Value==NULL) Return_Status="F";
        Syelog(**OUTPUT DATA**);
    };
    TIME++;
    return Return_Value;
}
```

Figure 27: Hook Function Example

Compiling the completed source code generates a new DLL file. Detours offers two secondary programs that apply the hook DLL to a given executable.

- Setdll: This program alters a target executable to reference the hook DLL. It permanently alters the binary of the target executable to reference the hook DLL anytime it is run.

- Withdll: This program loads a target executable with the hook DLL at runtime. Unlike setdll, there is no permanent alteration of the executable binary with this program.

Both *setdll* and *withdll* have similar results, but we use withdll for our work because it is more stable. *Withdll* runs from the command line by specifying a given DLL and a target executable: “withdll -d:MemAPI.dll EXCEL.EXE”. If *withdll* is successful, a process runs normally with the exception that all hooked functions log allocation and free data to syelog.exe (Figure 28).

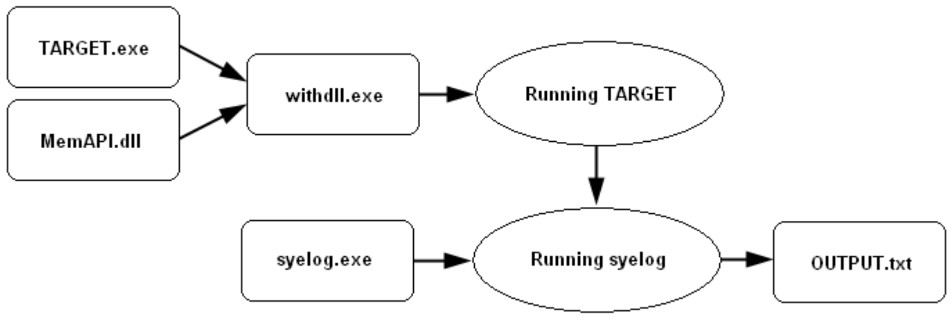
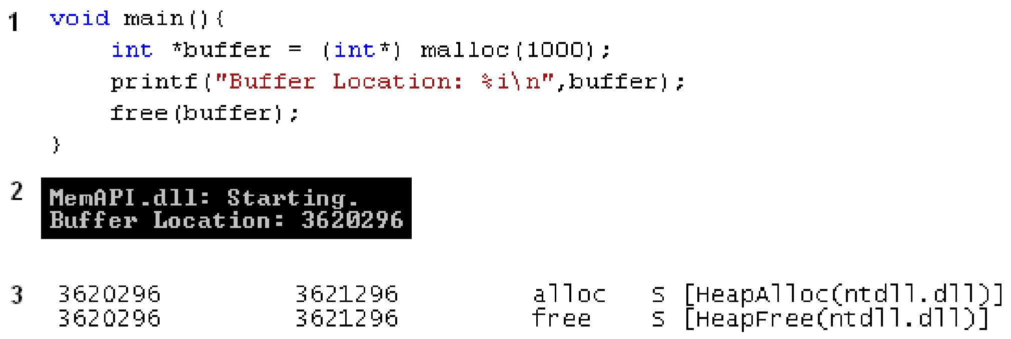


Figure 28: Detours Component Interaction

4.4.5 Confirming the Accuracy of the Hook

Before using the captured data to generate visual representations, we must confirm its accuracy. To facilitate this goal, a small test program allocates a buffer and outputs the starting address of the buffer (Figure 29-1). Then we apply the hook to our test program using withdll.exe. If the allocation is successfully hooked, the start address that is captured should match the address printed by the code.



1
```
void main(){
    int *buffer = (int*) malloc(1000);
    printf("Buffer Location: %i\n",buffer);
    free(buffer);
}
```

2
```
MemAPI.dll: Starting.
Buffer Location: 3620296
```

3
```
3620296        3621296        alloc    S [HeapAlloc(ntdll.dll)]
3620296        3621296        free     S [HeapFree(ntdll.dll)]
```

Figure 29: Conformation Test for API Hooking

As shown in Figure 29-2, the start address is 3620296. After adding the size of 1000, the resulting end address is 3621296. The output hook — seen in Figure 29-3 — shows the addressing for the allocation and free requests match the address output by the code.

4.5 Capturing Read & Write Memory Operations

The previous subsections discussed data capturing related to allocation and free information located in the API. Although we do not implement a capturing method for read and write operations, we still need to supplement the data to make it complete for testing the visual representations. The supplemental data comes from test programs that we use which have built-in print statements that output the memory operations executed by the code. The built-in capturing technique has several downsides: (1) it does not work with existing programs, (2) it affects the natural behavior of a program, and (3) it only captures specifically indicated data. However, we still implement the technique because testing of any visual representation cannot occur without an example of a complete data set. This technique may not be practical in the long term, but we choose to do it given our focus on visualization and not data capture.

4.6 Data Format Requirements

This sub-section discusses the standard data format required by a visual representation of memory utilization and discusses refinement needs for the captured API data. A standard data format helps do the following things: keep any data parsing or refinement outside the functionality of the visual tool, combine variable named and complex data into one uniform scheme, and give any future data capturing work a template for easy integration.

All captured memory data has the following six elements:

- Start Address: The start address is the core piece of information needed for reconstructing any memory operation. Allocations need a specific address for a memory request or they return the start address if the system determines a different allocation location. Memory "free" operations require a start address; otherwise, they fail. Read and write commands must have a reference to the location of the data that they need. The value of the start

- End Address: The end address is the complement component to the start address. Combined together, the start address and the end address define two points in the memory space that can translate to two points graphically. The size of a given memory area can also be derived from the start and end address. Like the start address, the end address must be a valid value within the host system and represented by an integer.
- Operation Type: As discussed earlier in this section, memory operations divide into four core types: allocations, frees, reads, and writes. Each core type represents a character string equal to "alloc," "free," "read", or "write." Any other string value is an error.
- Memory Type: As discussed previously, process memory is one of two types: variable or non-variable. This data component is a string value equal to "variable," "non-variable," or "unknown." Potentially, visualization can expand to include other valid values such as reserved, committed, static, and dynamic, heap, or stack.
- Return Status: This component indicates if the memory operation succeeded. The status is a single capital character, either 'S' (success) or 'F (failure).'
- Additional Information Descriptor: This component provides additional details about a given operation. The entire description is a string contained within a single set of brackets. Users can determine details within the description. The brackets contain any information that is secondary to the graphical functionality of a visual representation; examples include API function name, DLL name, notes, miscellaneous parameters, and address translations.

When the data is recorded to an output file, Figure 30 shows the format of the components; a single tab space separates all components

.

StartAddress   EndAddress   OperationType   MemoryType   ReturnStatus   [Descriptor]

Figure 30: Data Format

At this point, we have identified the data components and format. Now we must confirm that the data captured by the API hook corresponds to the format. Unfortunately, due to the inherent functionality of some API, certain data is not ready for the visual representation. For example, HeapFree does not have an end address or a return value parameter. Hence, we must scan the preceding data to find the corresponding HeapAlloc that contains the memory block size to complete the format requirements. Ordering is another issue with API. For example, if malloc invokes HeapAlloc then the entry for HeapAlloc appears before the entry for malloc. Time stamping the API data enables a formatting program to sort the data into chronological order. Repetition of data is a third major problem. Some information could be redundant because of nested API functions. To compensate for redundant data, we implement more supplement code that compresses and eliminates duplicate data. After resolving these data issues, our allocate and free data is ready for visual representations. The read and write data requires no refinement because it has implicitly defined output. At this point, we have the allocation, free, read, and write data to adequately test visual representation.

At this point, the data is ready for conversion into a visual representation of the memory model. Our visualization tool assumes that any data captured for analysis has been formatted. We now need to discuss the process of converting the formatted data into a visual representation.

5 Memory Visualization for Security Analysis

To this point, we have discussed the following considerations: the need for support tools to aid security analysis, the memory model that defines normal behavior, and where and how to obtain raw memory data. We are now ready to create a visual representation of memory utilization. However, the process of creating visual representations has many issues and challenges. This section discusses the development process for our visual representation tool. We discuss the following items: general issues and challenges associated with creating visual representations, possible and chosen design options, requirements and guidelines for the development of a visual tool, a prototype implementation of the tool, and conformation of the tool by testing it with sample data.

5.1 Issues and Challenges

The goal of any visualization of data is to create insight into that data; this means that the visual representation of the data provides more knowledge and information to the user than the data would in its raw form. Before creating a visual representation, the data is the first source of issues. Only certain data is realistic to visualize. Any given data set has various attributes: start and end address for example. These attributes translate to visual components, such as color or location, to generate a visual representation. If a given data set is too simplistic — i.e., has few attributes — a visual representation for that data could be useless and provide no additional insight. Likewise, if visualization tries to include too many attributes at once, any insight could be lost due to clutter.

Effective visual representations depend on mapping data attributes to appropriate visual components. Position, color, length, angles, and volumes all need to work together in order to generate a given visual representation. We must choose how visual components map to the memory data in order to yield the most insight. For example, key data attributes should not map to lesser visual elements such as color, but rather critical elements such as position.

5.2 Design Options and Choices

Creating visual representations has many different design options; some greatly enhance data whereas others are not appropriate for a given case. In general, we follow Shneiderman's visualization mantra of "overview first, zoom and filter, then details on demand" [Shneiderman 1996]. The mantra gives the implementation of our tool a guideline that helps get the most insight into the memory data set. Therefore, the visual tool separates into three parts: the overview, zooming and filtering options, and detail queries. The overview is simply the primary graphical elements. Zooming and filtering techniques refer to visual modifications to the overview; these modifications typically result in a new overview or child visualizations. Lastly, detail queries give users an un-abstracted view of the data behind the visual representation. This sub-section discusses key design options for creating visual representations. The design options separate as they relate to a given component of the visualization mantra.

5.2.1 Design of the Graphical Environment:

A visual representation displays data in one, two, or three dimensions. It is important to make the distinction that we are referring to graphical dimensions and not the dimensions of the data set. The graphical environment mostly affects the overview and zoomed/filtered visual components

- One-dimensional visual representations typically represent a single value; these visualizations are extremely limited and are rarely used. We choose not to use this option since our data set is large and has several data dimensions.

- Two-dimensional visual representations are the most common; examples include graphs, maps, flow charts, and pie charts. Typically, 2D visual representations enable the best blend of computing efficiency, graphical detail, and compatibility with different hardware. For our implementation, we choose this option for the graphical environment. Two dimensions allow us to visualize the linear memory address structure and complement it with other information from our data set.
- Three-dimensional visual representations typically show complex special data or compare multiple data elements at the same time; examples include three-axis scatter plots and virtual environments. Using three dimensions to represent certain data sets (including ours) can make the visualization more complex than necessary. Furthermore, effective 3D visualization requires high graphical computing performance or specialized hardware.

5.2.2 Data Mapping:

Choosing a data mapping method is equally important when choosing the correct graphical environment. As discussed in section 2.2.1, some visual memory tools map address spaces to a 2D plane. These tools map address locations to pixel rows. When the row fills, the addresses then map to the next row, as shown in Figure 31.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

Figure 31: Example of Mapping Addresses to a 2D Pixel Space

The above figure shows the problem with a 2D mapping of memory addresses. Logically, address location 8 is adjacent to 9, but in the 2D mapping the two locations are not visually adjacent. In contrast, we choose to map all memory addresses only to the horizontal axis. This approach maintains the natural structure relationships between memory address. A natural structure also provides users with better insight into memory behavior. However, our tool is not one-dimensional; the veridical axis maps to secondary data elements, such as like type of memory. Section 5.4.2 discusses further details about our mapping techniques.

5.2.3 Zooming, Interaction, Details, and Queries

Zooming and filtering are necessary for a visual representation of memory data due to the overall size of the data set and the large values contained within the data set. Our tools allow users to zoom to any memory range contained within a given data set. When a user zooms into a subset of memory addresses, the tool spawns a separate visual representation without removing the original overview; this approach allows the user to have a reference to the overview at all times. The tool also enables users to view a dynamic amount of visual representations for zoomed address ranges.

The tool processes memory data sequentially (i.e., one memory event at a time). Users have the following options: let the tool automate time progression, manually step through memory events backward or forward, and jump to a specific event. Letting the tool automate the progression of time enables the user to watch memory utilization over execution time. Stepping forward and backward enables users to pause time progression and carefully investigate certain memory events. Jumping to a given event gives the option to bypass uninteresting areas and quickly navigate the data set.

The tool provides two different levels of detailed data. First, the entire data set is available in text format, which enables users to find interesting data manually or jump to a position in the data. Second, the tool provides the details of the current memory event at the current time index. Users can also make queries into a specified address or address range.

5.3 Requirements and Guidelines

This sub-section identifies the requirements that move the r development of the tool from the design phase to implementation. The requirements separate into backend, graphical, data, and interaction categories.

Graphical Requirements:

- A pixel space encapsulates a given address range.
  - The width of the pixel space equals the width of the application minus a margin area. The height of the pixel space has a dynamic value with a minimum height equivalent to the number of secondary data elements that the vertical axis depicts.
- The address space with a data set maps to the horizontal pixels of the pixel space.
  - Mathematical scaling of the address space decreases the number of addresses represented by an individual pixel.
    - The tool subtracts the value of the minimum address location from all addresses in the data set.
    - The tool calculates the largest common factor of all address values, then divides all addresses by that value.
    - All address values must be a non-decimal value after scaling.
    - All address values readjust to their original value before a user views them.
  - Each transparency value of a pixel derives from the ratio of allocated addresses to unallocated addresses within the range of that pixel.

        - If a pixel represents any amount of allocated addresses then the transparency value starts at a minimum value that guarantees the visibility of the pixel.
    - The address per pixel ration cannot be less than one after scaling. If the ratio is less than one, the tool increases the address range until it is greater or equal to one.
    - The top and bottom edges of each pixel space never have a transparency so to add a visual highlight to areas with memory activity.
- Data elements that do not relate to addressing map to the vertical pixels and color of the pixel space
- The original overview pixel space always is at the top of the visualization space and cannot be deleted.
- Zoom interaction with the overview pixel space produces child pixel spaces that appear below the overview.
    - Zoom interaction with the child pixel spaces produces new child pixel spaces that appear below the parent space.
    - User interaction causes the generation of child pixel spaces.
    - Users can delete child pixel spaces at anytime.
    - A child pixel space never encompasses address values outside the address range of the overview.
- The tool must provide a key that indicates how colors translate to data.

Backend Requirements:

- The tool keeps and maintains a replica structure of virtual memory that changes as the tool processes the data set. The replica structure of virtual memory governs the graphical elements.
    - The tool keeps an individual replica structure of virtual memory for each horizontal pixel that represents the address space within the pixel.
- The tool disregards any unformatted data.

Data Representation Requirements

- The minimum group of secondary data includes variable allocation, non-variable allocation, read, write, and violation/error status.
- A different color represents each data element not related to memory addressing.
- Brighter and highly contrasting colors represent the memory operations that result in errors, violations, or otherwise unnatural memory utilization.
- The start and end address of the current memory operation must be marked on each pixel space.
- If a read or write occurs within a given pixel, the tool keeps the pixel marked to indicate that a read or write occurred until the tool deletes the respective address range.
- The tool must never occlude colored areas that indicate variable allocation, non-variable allocation, read, write, or violations/errors.
- Each pixel space must have a descriptor area below it that provides general details.

Other Requirements

- Users must be able to reference detailed information about any pixel space or address range.

- Users must be able to control the simulations speed by changing the progression of time and pausing.
- Users must have a view of the data in text format and be able to use the text view to navigate the data.
- The tool must show details related to the current memory operation.
- Users must have easy control over the current time index of the simulation.

For the purpose of our research, we have identified the minimum requirements to enable the visual representation of the memory model in section 3. These requirements focus on the basic types of memory utilization: allocate, delete, read, and write. However, the requirements also enable future expansion that includes types of utilization functions such as commit and release.

5.4 Implementation

This sub-section discusses the implementation of the requirements to create a working prototype of a tool that generates visual representation of memory utilization. To begin, we implement the tool by using the C# programming language and Microsoft Visual Studio 8 (VS8) development environment. This combination enables the use of built-in toolkits that allow us to focus on logic and functionality rather than low-level code. Furthermore, C# and VS8 also provide a convenient template for graphical applications.

The remainder of this sub-section identifies key components of the prototype tool. These components relate to generation of visual representations and not the backend operations of the tool.

5.4.1 The Event List

The event list component manages the flow of the visual simulation. The tool creates the event list when it reads a data file. Each data line represents a new memory event (Figure 32-1). The memory event class within the list has nine key parts:

- ID: The tool assigns a unique ID value when it reads the event from the input file. The ID also represents the chronological position of the event in the simulation timeline.
- Start Address: The start of the address range that the utilization event affects.
- End Address: The end of the address range that the utilization event affects.
- Type: This value refers to the operation type. The type can be one of the following values: ALLOC, FREE, READ, or WRITE (Figure 32-4).
- Memory Type: This value represents the memory classification that the utilization event affects. The memory type can be one of the following values: RESERVED, COMMITED, STACK, HEAP, FREE, DECOMMIT, RELEASE, or UNKNOWN (Figure 32-2).
- Memory Subtype: This value indicates if the utilization event affects memory that is variable or non-variable. The UNKNOWN value indicates if the memory subtype is irrelevant for the utilization event.
- Return: This value indicates the success status of the utilization event.
- Description: This value contains any textual description of the event.

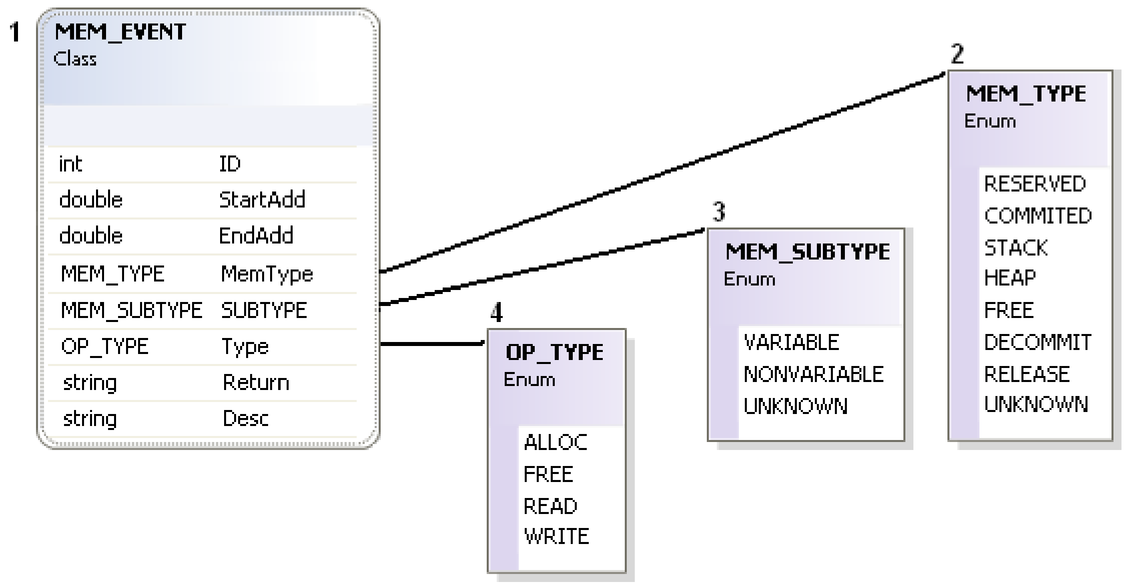


Figure 32: The Memory Event Class

Once completed, the event list guides the progression of the simulation. The tool uses the event list to update the pixel space and provide details to users.

5.4.2 The Memory Image

The core of the visual representation is the memory image bitmap. We chose to use bitmaps instead of general graphical rendering for two primary reasons. One, the task of modifying a bitmap and then drawing it to the screen is better for performance than drawing the data pixel by pixel. Two, bitmaps provide encapsulation for each memory range instance, both overview and zoomed. Furthermore, the memory image enables us to have a general method to generate visual representations; in other words, the generations of overviews or zoomed address spaces will remain the same and only require us to specify a range of addresses. The implementation of the memory pixel space (MEM_IMG) starts with the creation of a separate class. The MEM_IMG class is responsible for maintaining the memory structure of the pixel space and all transparency values. The description of the implementation starts from the ground up, beginning with memory structure management.

5.4.2.1 The Memory Block Class

The replica of the memory structure behind the pixel space starts with the MEM_BLOCK class (Figure 33-4). The memory block represents an address range created by an allocation event. The class contains a start address and end address that also function as a two-part identification key that the tool uses to compare two blocks. A type and subtype identify the type of memory contained within the block. Finally, three flags indicate the occurrence of a read, write, or illegal access. For example, if a write event references a non-variable block, then the tool sets the violation flag.

5.4.2.2 The Memory Space Class

The MEM_SPACE class (Figure 33-3) maintains all blocks within a given address range. This class requires a memory range with a size of at least one. The memory space contains a MEM_BLOCK list that represents its internal structure. The class also maintains statistics regarding how much memory various types occupy. In case of an allocate or free memory event, the MEM_SPACE class determines how much new memory to add depending on whether the new memory request has already been allocated or is contained within another allocated block. In the case of read or write events, the memory space is responsible for flagging the correct blocks and flagging references to non-variable types as violations.

The MEM_SPACE-MEM_BLOCK combination forms the logical recreation of the memory model in section 3 for a given range of addresses. This recreation is the most vital part of the visualization tool. The graphical methods simply apply color, position, and other properties to the information in the memory space. If the memory space is flawed, the visual representation is flawed. Small glitches in the graphics might not affect user insight into the memory space, but flaws in the space greatly hinder user insight.

5.4.2.3 The Memory Pixel Class

The MEM_SPACE class and MEM_BLOCK class provide the logical reconstruction of a memory space, which the MEM_PIXEL then uses (Figure 33-2). This class represents the graphical elements of a single pixel. It has two key components: (1) each memory pixel contains its own MEM_SPACE member and (2) the memory pixel uses its memory space to maintain an alpha value. The alpha value is a transparency value that represents the ratio of allocated space to total memory space. A transparency value can range from 0 to 255, but the tool calculates it starting from a base of 25; this approach enables users to easily see pixels with a low allocated/total ratio.

5.4.2.4 The Memory Image Class

Encapsulating the MEM_PIXEL is the MEM_IMG class (Figure 33-1). The MEM_IMG class manages the graphics of a given pixel space. The class keeps a MEM_PIXEL list that represents the transparency values for a bitmap. Each bitmap is one pixel high and represents a unique color. The tool shows colors for variable, non-variable, read, write, and violations. Limiting each bitmap to one pixel high maximizes computational performance; a full-scale bitmap requires the modification of Width*Height pixels rather than Width pixels. The tool generates full-scale bitmaps by simply rendering each bitmap to the screen several times.

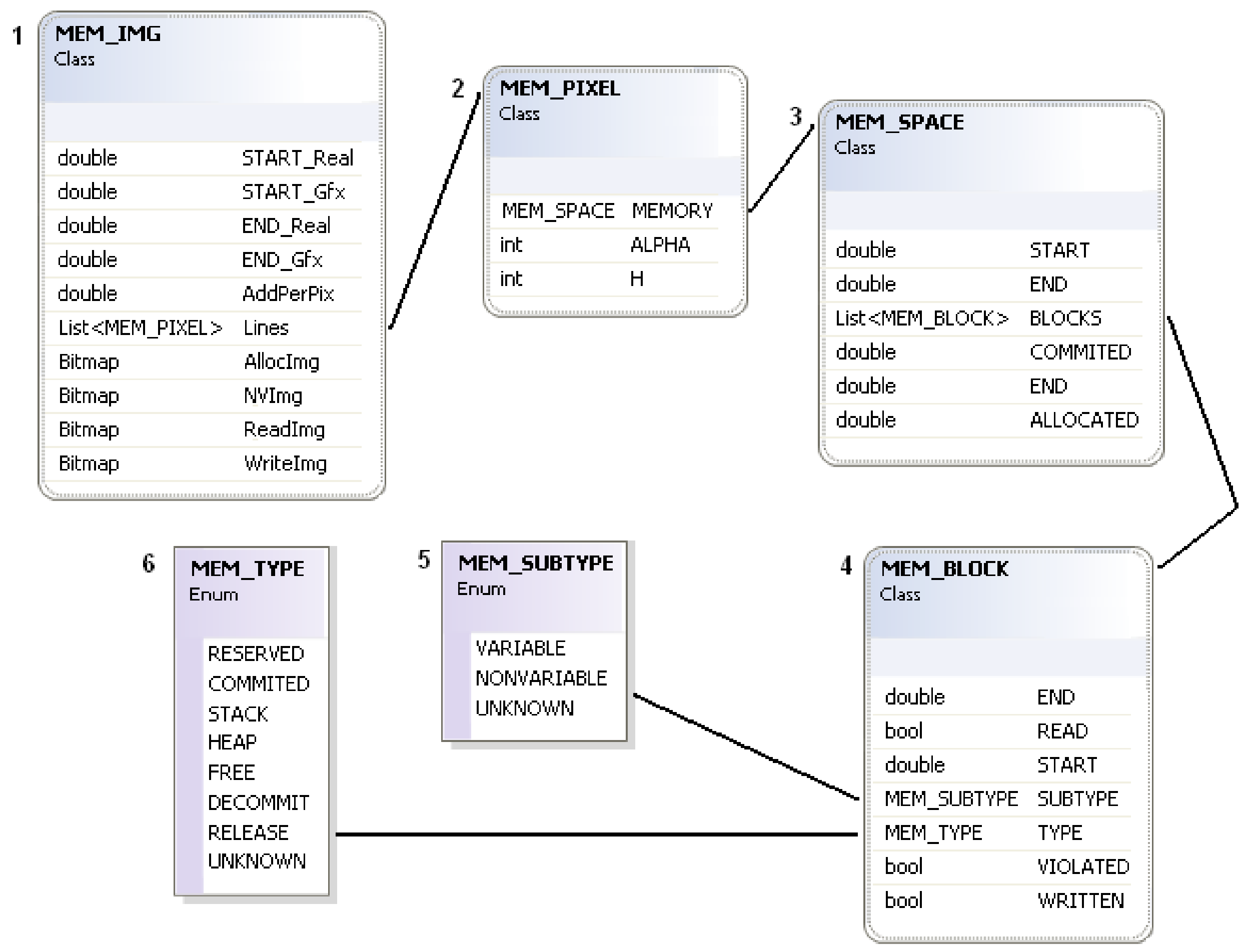


Figure 33: Simplified Flow Chart for the Memory Bitmap

5.4.2.5 Memory Image Scaling

As defined by the requirements, the memory image is also responsible for scaling the address space. Scaling the address space minimizes the ratio of address to available pixels; this approach enhances the graphical detail of each bitmap without compromising accuracy. Creating a new memory bitmap begins with a start address and end address. Using the start and end address, the tool traverses the event list and finds the largest common factor of all addresses within the given range. The tool then creates a new MEM_IMG and passes it the start address, end address, and common factor. Using the start address and end address, the MEM_IMG translates any address by subtracting the original start address and dividing by the common factor. For example, the following scenario shows the difference between scaled and non-scaled memory images.

- To illustrate the differences between scaled and un-scaled address spaces within each bitmap this example uses the following values:
    - Start address = 5,000
    - End address= 20,000
    - Largest Common Factor = 10
    - Bitmap Width = 950
- 20,000 – 5,000 / 950 = 15.8 → 16 (Requirement: must be whole number)
    - 16 x 950 = 15200 Total addresses
- 20,000 – 5,000 / 10 / 950 = 1.6 → 2
    - 2 x 1200 = 2400 Total addresses

The above example yields the two bitmaps shown in Figure 34 (un-scaled on the top); this figure shows that the tool can accurately pack more addresses into fewer pixels. The example scenario deals with a 15,000 byte address space; however, scaling works best for larger spaces.

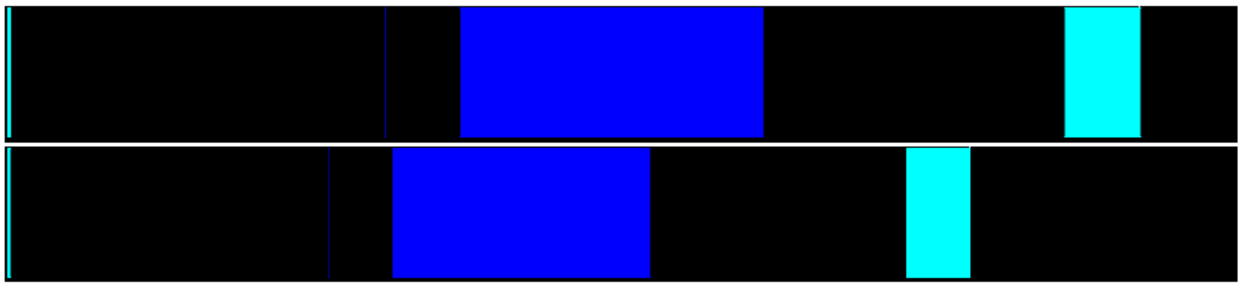


Figure 34: Un-Scaled vs. Scaled Bitmaps

5.4.2.6 Visual Properties

When the tool renders the result of memory events to the screen, each bitmap's visual properties can be one of several configurations. The major part of any configuration is transparency. As mentioned before (5.4.2.3), transparency represents the ratio of allocated memory to total memory contained within a given pixel. Figure 36 shows four transparency examples; we magnified the image in the figure, with the result that each colored bar represents one pixel in the actual image. Each part of Figure 36 represents the following: part A shows the maximum transparency value, B shows roughly 70% transparency, C shows a memory block example that crosses over 2 pixels, and D shows roughly 10% transparency.

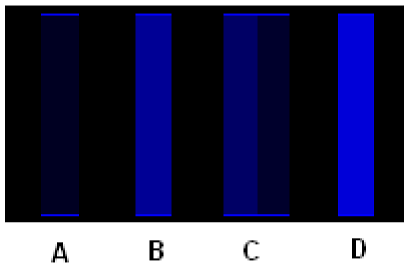


Figure 35: Various Transparency Examples

Color is the second part of any image configuration. Each main color indicates the memory type (variable or non-variable) and secondary colors represent read, write, or invalid memory operations. Figure 36 show various examples of memory event results with no transparency: part A is only non-variable memory, B is only variable memory, C is variable memory that has been read at least once, D is variable memory that has been written at least once, E has been read and written at least once, F is non-variable memory that has been read or written illegally, G is an example of a crossover from variable to non-variable.

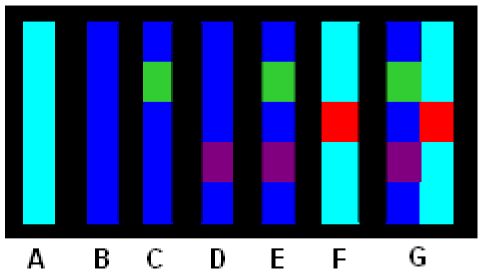


Figure 36: Visual Scenarios

Other than the color red — a typical color for errors — all colors were arbitrarily chosen; we chose colors that could be easily distinguished from one another visually. Black color in the background gives more contrast. .

5.4.3 User Interface

Secondary to the graphical bitmap space, the tool relies on secondary display areas and interaction to supplement the visual data. The design of each detail or interaction component focused only on functionality and insight; no additional effort was used in determining location, size, or appearance of each component.

5.4.3.1 Details

Providing secondary non-graphical information to the user is a key to effective visualization. All details in our tool come from the event list (5.4.1) or pixel information. The tool provides the following details to the user:

- Current Event Indicator: On the main image and all child bitmaps, the address range affected by the current memory event is marked (Figure 37-5).
- Image Detail Bar: An information bar – located below each bitmap in the display space – provides details about address space within the respective bitmap (Figure 37-8).
- Current Event Details: The tool displays text details about the current memory event (Figure 37-9).

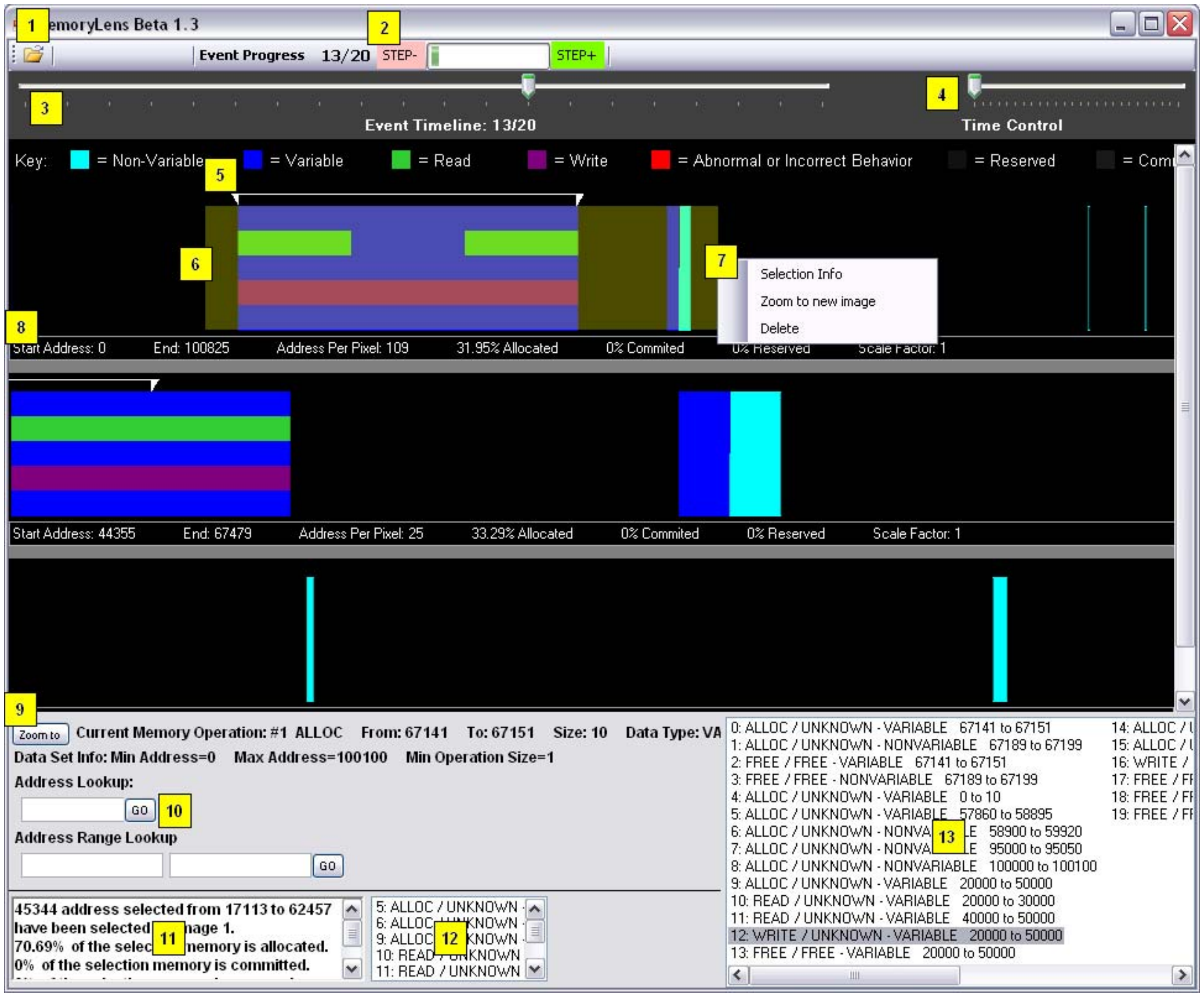
- Address Lookup: if a user provides a valid address, they can zoom directly to an image centered at that address (Figure 37-10).
- Address Range Lookup: If the user provides two valid addresses, the tool creates a child bitmap to fit the range (Figure 37-10).
- Event List Text: The tool displays a complete list of all memory events in text format (Figure 37-13).
- Address Range Details: If a user requests detailed information about a given address range, the tool displays the information in a field as shown in Figure 37-11. Any memory event that affects the given range is listed (Figure 37-12).

5.4.3.2 Interaction

Interaction is critical to our tool due to the potentially massive size of memory data sets. Users must be able to quickly navigate the data set and bypass uninteresting information. The tool provides the following user interaction components:

- File Finder: A file browser dialog enables users to find memory data files (Figure 37-1). A user can load a new file at any time, both at startup and during any point in analysis.
- Stepwise Time Control: Two buttons enable users to increment the progression of time both forward and backward (Figure 37-2).
- Timeline Control: Users can also control the time index by using a track-bar (Figure 37-3). This interaction component allows a user to see and jump to any point in the timeline.
- Time Automation Control: A second track-bar controls the number of events that the tool processes per second (Figure 37-4). The control is defaulted to zero. The value of the track bar determines the number of events that the tool processes before each graphical refresh.

- Image Space Menu: Through mouse interaction, the user can select an address/pixel range (Figure 37-6). The tool highlights the selected area with a transparent overlay. After a user has selected a range, they have the option to look up details about it or make a child image of it (Figure 37-7).
- Current Event Zoom: This control enables the user to zoom directly to a child bitmap centered at the location of the current event (Figure 37-9).



```
Start Address: 0    End: 100825    Address Per Pixel: 109    31.95% Allocated    0% Commited    0% Reserved    Scale Factor: 1
Start Address: 44355    End: 67479    Address Per Pixel: 25    33.29% Allocated    0% Commited    0% Reserved    Scale Factor: 1

Current Memory Operation: #1 ALLOC    From: 67141    To: 67151    Size: 10    Data Type: VA
Data Set Info: Min Address=0    Max Address=100100    Min Operation Size=1
Address Lookup:
Address Range Lookup

45344 address selected from 17113 to 62457
have been selected    nage 1.
70.69% of the selec    memory is allocated.
0% of the selection memory is committed.

5: ALLOC / UNKNOWN -
6: ALLOC    KNOWN -
9: ALLOC    KNOWN -
10: READ    NKNOWN
11: READ / UNKNOWN

0: ALLOC / UNKNOWN - VARIABLE   67141 to 67151
1: ALLOC / UNKNOWN - NONVARIABLE   67189 to 67199
2: FREE / FREE - VARIABLE   67141 to 67151
3: FREE / FREE - NONVARIABLE   67189 to 67199
4: ALLOC / UNKNOWN - VARIABLE   0 to 10
5: ALLOC / UNKNOWN - VARIABLE   57860 to 58895
6: ALLOC / UNKNOWN - NONVA    E   58900 to 59920
7: ALLOC / UNKNOWN - NONVA    E   95000 to 95050
8: ALLOC / UNKNOWN - NONVARIABLE   100000 to 100100
9: ALLOC / UNKNOWN - VARIABLE   20000 to 50000
10: READ / UNKNOWN - VARIABLE   20000 to 30000
11: READ / UNKNOWN - VARIABLE   40000 to 50000
12: WRITE / UNKNOWN - VARIABLE   20000 to 50000
13: FREE / FREE - VARIABLE   20000 to 50000
14: ALLOC / U
15: ALLOC / U
16: WRITE /
17: FREE / FF
18: FREE / FF
19: FREE / FF
```

Figure 37: Example Layout for Our Tool.

5.5 *Confirming the Visualization*

The previous sub-sections showed the design goal, requirements, and implementation choices for our visualization tool; now we need to see if the designed result is produced when sample data is given to the tool. We do not test our visualization with security analysis scenarios; instead, we use simple data for a proof of concept.

To begin, Figure 38 shows how the tool processes data with large differences in size. In the non-zoomed view (Figure 38-A), it appears that there is a single gap between two memory blocks. The medium zoomed view (Figure 38-B) shows that there is actually a number of small memory blocks between two larger ones. Finally, Figure 38-C shows the memory space zoomed to the maximum amount. All three views also show how transparency values change as the magnification increases.

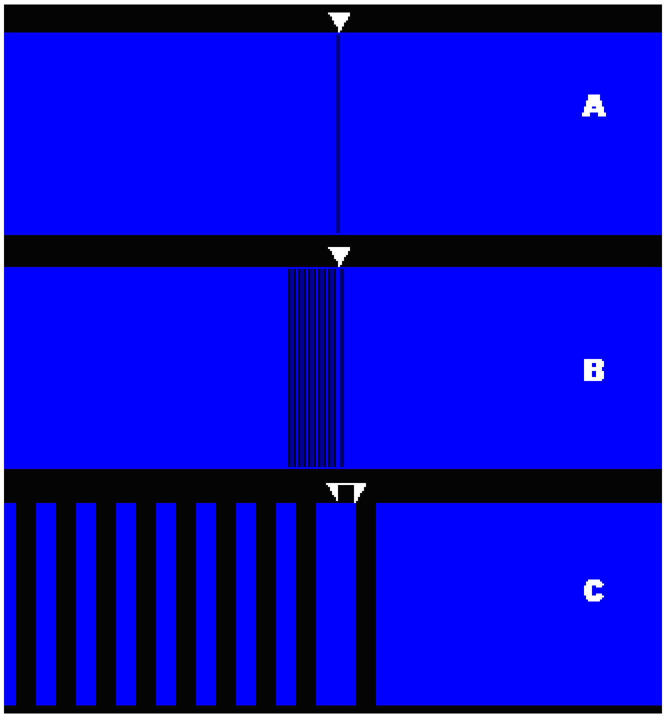
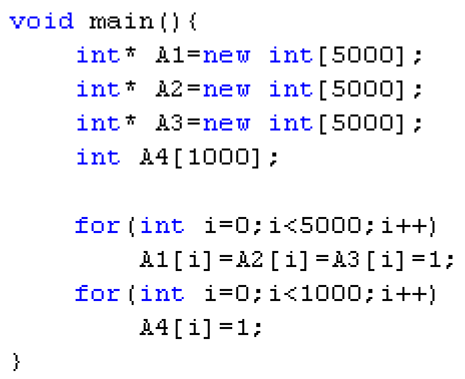


Figure 38: Overview and Zoom in action.

In another test, the program shown in Figure 39 generates a sample data file. The program allocates three dynamic arrays and one static array. Then, the program sets all array indices to one.



```
void main(){
    int* A1=new int[5000];
    int* A2=new int[5000];
    int* A3=new int[5000];
    int A4[1000];

    for(int i=0;i<5000;i++)
        A1[i]=A2[i]=A3[i]=1;
    for(int i=0;i<1000;i++)
        A4[i]=1;
}
```

Figure 39: Simple Testing Program.

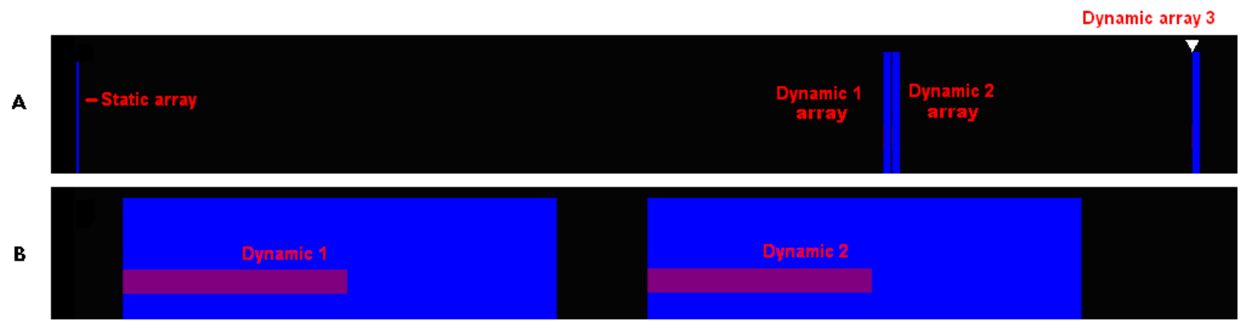
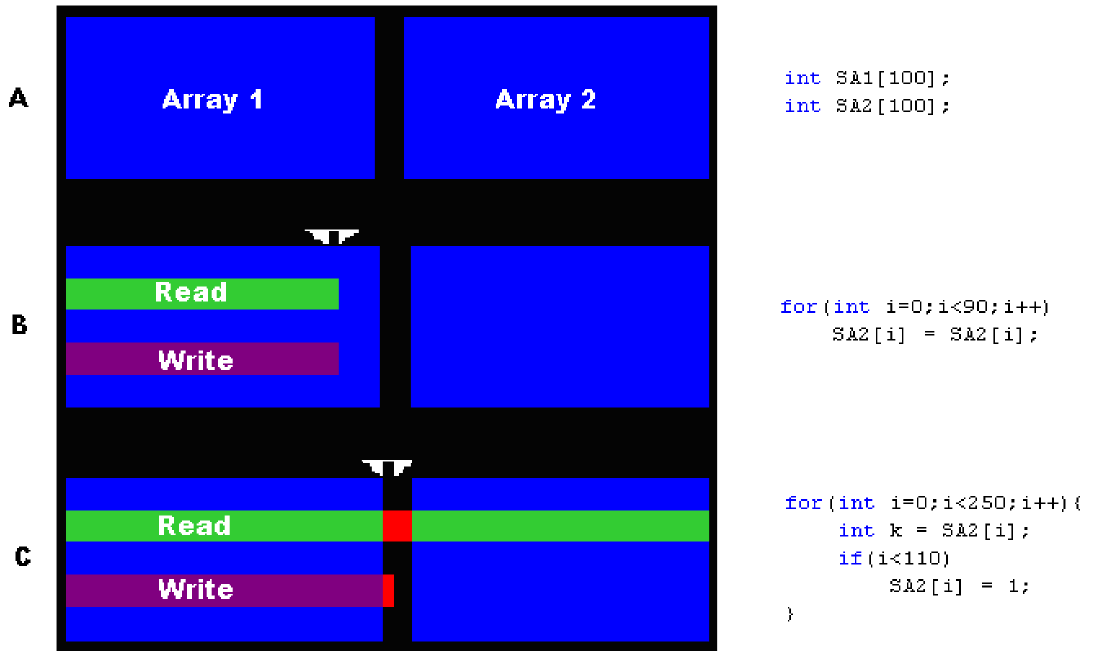


Figure 40: Visual result of the code in Figure 39.

Figure 40 shows the result of the program in Figure 39. The figure shows several characteristics that confirm parts of the memory model. First, Figure 40-A shows the separation of the address ranges that contain the heap and stack. Secondly, the figure shows that the OS places dynamic arrays wherever they fit first, although the program requested them adjacently. Next, Figure 40-B (result of a zoom into A) shows the progression of write operations through two dynamic arrays.

Although the scope of our current work does not include formal testing for vulnerability analysis, we can illustrate a simple example of a potential vulnerability scenario. Figure 41 shows how a successful buffer overflow would appear in the visualization. First, Figure 41-A shows code and visual aspects for the initial allocation of two arrays. Next, Figure 41-B shows the progression of a for loop that performs read and write operations through the first array. Finally, Figure 41-C shows an overflow occurring as the write operations access addresses that are not within the first array. The advantage of this visualization is that any abnormal memory utilization visually stands out from other visual components.



```
int SA1[100];
int SA2[100];
```

```
for(int i=0;i<90;i++)
    SA2[i] = SA2[i];
```

```
for(int i=0;i<250;i++){
    int k = SA2[i];
    if(i<110)
        SA2[i] = 1;
}
```

Figure 41: Visualizing Abnormal Behavior

These examples confirm the correct behavior of the tool with respect to a few example memory scenarios. The two examples in this sub-section do not test numerous memory behaviors; we test the tool with other samples, but do not include them in this thesis.

In summary, this section has discussed our process of implementing a tool that converts the data obtained in chapter 4 into a visual representation of the memory model in chapter 3. We now have a baseline framework for the general process and a working prototype that enables a proof-of-concept.

6 Conclusions and Future Work

The increase in exploitation of vulnerabilities in recent years is the primary motivation that drives our research. Our specific goal is to answer Anil Bazaz's call for more analytical tools, which he makes in his dissertation titled *A Framework for Deriving Verification and Validation Strategies to Assess Software Security* [2006]. To answer Bazaz's call, we designed and implemented a visualization tool that facilitates the process of detecting vulnerabilities by generating visual representations of memory utilization. To develop the tool, we separated the work into three parts:

- The memory model outlines the standard functionality of the Windows 32-bit memory system. We also use the model to identify the correctness of the visual representation.
- The process of data collection identifies the data required to generate visual representations of memory. This process includes the identification of required data, implementation of one method that captures data, and definition of a standard data format.
- We identify requirements and implement a prototype tool. The prototype provides a proof-of-concept for the combination of the memory model, the data collection process, and the requirement list.

Our work only discusses the tool and its foundations; we do not cover any specific security analysis.

6.1 Contributions

The information and work presented for our research contributes to the fields of software analysis and information visualization. Our notable contributions include the following:

- *The Memory Model*: The model characterizes standard behavior of memory utilization. We built our memory model by combining information related to memory utilization such as memory objects (e.g., buffers, pointers, etc), structures (stack and heap), and functionality (API). Before combining, information related to various memory topics is normally found throughout many sources such as books, websites, documented code, and undocumented code, which makes it difficult to obtain a full understanding of the overall process for memory utilization. Furthermore, many existing sources focus on memory at only one level whereas our model combines knowledge related to the physical level, virtual level, and process level. We also add to existing source material by implementing simple programs that extract lesser-known information. These simple programs elicit specific information not commonly found in literature, such as the method that enables one sub-routine to write to the memory range of another sub-routine; these simple programs also provide a mechanism to test the existing information in the memory model.
- *Process for Data Collection*: The process for data collection identifies and captures the data necessary for the visualization of memory. Our process of data collection identifies the following variables: the minimum data required to generate visual representations (allocation, frees, reads, and writes); the location of data related to allocation and free operations that modify the memory structure; and possible methods to capture the data from the physical, virtual, and process levels. Our process only focuses on the minimum data requirement for the visualization but the process can facilitate future work that includes other data related to memory operations such as commits or releases.

- *API Functions Hooks*: The using hooks of API functions related to memory provides the data necessary for visualization. Although our capturing method uses an existing toolkit, we implemented new hooks for specific API functions that relate to memory utilization. Furthermore, we identified the definitions of many memory functions that are otherwise difficult to locate (e.g., NtAllocateVirtualMemory). The function hooks provide both functionality for our purposes as well as templates for hooking other functions in the future.
- *Standardized Data Format*: The standardized format prepares memory data for visualization. The main purpose of the data format is to enable easy integration of future data sources and separate the processing overhead of formatting from our prototype so that it only focuses on visualization. Our formatting process also identified common issues related to memory data. First, data captured from the API must be sorted so that all entries match the chronological order in which they were called. Second, some functions such as HeapFree use a heap identification number to free a memory range with no reference to specific addresses; hence, we have to supplement the data by scanning it to find the respective HeapAlloc call.
- *Requirements for a Visualization Tool*: The requirement list facilitates the implementation of a baseline tool that enables insightful visual analysis. In the requirement list, we specify a new method to map memory data. Key components of our visualization method are as follows: address map to a 1-dimensional horizontal area, secondary information (operation type for example) map to the vertical axis, and transparency and scaling equations that are used to represent an address range that is larger than the graphical space. Without these minimum requirements, insight into

- *The Prototype Tool*: Our final contribution is the implementation of a working prototype tool based on the list of requirements. We use the prototype as a proof-of-concept that the combination of the memory model, data collection, and requirement list is capable of producing a tool that generates visual representation of memory utilization. Furthermore, we designed components within the tool, such as the MEM_IMG, to enable quick and simple generations of visual representations both for our tool and for possible use in other memory analysis work.

In sum, we have defined a process that generates visual representations of memory utilization. By following a step-by-step process, starting at the memory model, we have enabled our visual representations to accurately display memory data for the user. Given that our tool is currently a proof-of-concept version, we hope that future work can directly apply the tool to the process of detecting vulnerabilities and exploits within software.

6.2 Future Work

Our work focused on the development of a new method to facilitate the process of detecting vulnerabilities and exploits; more specifically, we focus on visualization of memory. In this sub-section, we discuss some of the possible areas for improving and expanding our work:

- *Expansion of the memory model*: We only implemented a few simple programs that tested existing information or generated new information for the model. Adding more literature sources and simple programs that add more in-depth detail can further improve the model. Additionally, new information related to higher-level functions such as

- *Addition of new methods for data capturing*: We only focused on implementing a capturing method that targets memory operations that involve allocations or deletions. Our work would greatly benefit from the addition of capturing methods that focus on or include data related to read and write operations.
- *Addition of higher-end visualization requirements*: This thesis presents baseline requirements for memory visualization that encompasses allocations, deletions, reads, and writes. Future work might add requirements for higher-level memory data such as commit and reserve operations or information related to heap and stack structures.
- *Expansion of the memory visualization tool*: Our requirements list produces a tool that generates one form of a visual representation for memory. Future work could add secondary visualization components, details, and user interactions to enhance insight into memory data.
- *Utilization testing*: Our research focused on the background and design of an analysis tool. Future work might expand our work by testing the effectiveness of our tool when directly applied to the process of detecting vulnerabilities and exploits. The tool can also be tested by applying it to actual security issues, e.g., security problems in the real world. Furthermore, the tool could be tested for insight into other memory behavior aside from security, such as like memory efficiency.

6.3 Summary

The growth explosion of the internet over recent years has resulted in a respective growth explosion in digital attacks on computers. A large subset of computer attacks directly target program vulnerabilities. The increase in attacks has prompted a need to develop new ways to classify, detect, and avoid vulnerabilities. The dissertation by Anil Bazaz discussed a new process of classifying and detecting vulnerabilities by providing a framework comprised of a taxonomy, an object model, and V&V strategies. Our work is motivated by Bazaz's call for new tools that aid the detection process; it is this call that our work tried to answer, with specific reference to main memory resources. In this thesis, we presented the development of a tool that visualizes memory utilization for the purpose of security analysis. The development started with the creation of a memory model that is based on the Windows 32-bit operating system. We then defined what memory data we needed and implemented one possible method to capture it: API hooking. We then identified requirements for a visualization tool using the memory model as a guide. Our work finished with the implementation of a prototype that drew upon the requirement list. In conclusion, we have presented the foundation for a tool that can help detect vulnerabilities and flaws in software with regard to main memory utilization.

APPENDIX A

NT Virtual Memory API Manager

Documentation on NT API functions is provided by NTinternals.net [NTinternals.net 2006].

- NtAllocateVirtualMemory
    - Parameters: IN HANDLE ProcessHandle, IN OUT PVOID *BaseAddress, IN ULONG ZeroBits, IN OUT PULONG RegionSize, IN ULONG AllocationType, IN ULONG Protect.
    - Return Value: NTSTATUS
    - The function is designed to allocate a region starting at the address indicated by BaseAddress of the size indicated by RegionSize. If BaseAddress is zero then the function defaults to finding the first best fit inside the virtual address space. The value of AllocationType defines if the allocation is committing or reserving memory.
- NtFreeVirtualMemory
    - Parameters: IN HANDLE ProcessHandle, IN PVOID *BaseAddress, IN OUT PULONG RegionSize, IN ULONG FreeType.
    - Return Value: NTSTATUS
    - This function frees a memory region starting at BaseAddress of size RegionSize. If RegionSize is not provided the entire region at BaseAddress is freed. The value of FreeType indicated if the memory is released or de-committed.
- NtReadVirtualMemory

- Parameters: IN HANDLE ProcessHandle, IN PVOID BaseAddress, OUT PVOID Buffer, IN ULONG NumberOfBytesToRead, OUT PULONG NumberOfBytesReaded OPTIONAL
  - Return Value: NTSTATUS
  - This function reads the contents of a memory spaces starting at the base address location.
- NtWriteVirtualMemory
  - Parameters: IN HANDLE ProcessHandle, IN PVOID BaseAddress, IN PVOID Buffer, IN ULONG NumberOfBytesToWrite, OUT PULONG NumberOfBytesWritten OPTIONAL.
  - Return Value: NTSTATUS
  - This function writes the contents of Buffer to a memory spaces starting at the base address location.
- RtlAllocateHeap
  - Parameters: IN PVOID HeapHandle, IN ULONG Flags, IN ULONG Size
  - Return Value: PVOID
  - This function allocates the appropriate memory space for a heap object.
- RtlCreateHeap
  - Parameters: IN ULONG Flags, IN PVOID Base OPTIONAL, IN ULONG Reserve OPTIONAL, IN ULONG Commit, IN BOOLEAN Lock OPTIONAL, IN PRTL_HEAP_DEFINITION RtlHeapParams OPTIONAL
  - Return Value: PVOID

    - This function creates a new heap object that belongs to a given process. The various secondary parameters control the initial state of the created heap.
- RtlDestroyHeap
    - Parameters: IN PVOID HeapHandle
    - Return Value: NTSTATUS
    - This function deletes the heap object that is identified by the handle.
- RtlFreeHeap
    - Parameters: IN PVOID HeapHandle, IN ULONG Flags OPTIONAL, IN PVOID MemoryPointer
    - Return Value: BOOLEAN
    - This function frees a section of memory from a given heap object.

Virtual Memory API

- VirtualAlloc (VirtualAllocEx & VirtualAllocExNuma variants)
    - Parameters: in LPVOID lpAddress, in SIZE_T dwSize, in DWORD flAllocationType, in DWORD flProtect
    - Return Value LPVOID
    - This function allocates a virtual memory block of size dwSize starting at address lpAddress. The allocation type can be equal to MEM_COMMIT, MEM_RESERVE, or MEM_RESET. MEM_COMMIT—valued 0x1000—specifies the function to commit a block of free or reserved memory. MEM_RESERVE—valued 0x2000—indicates the function reserves a block of memory that is currently free. MEM_RESET—valued 0x8000—instructs the

- VirtualFree (VirtualAllocEx variant)
    - Parameters: in LPVOID lpAddress, in SIZE_T dwSize, in DWORD dwFreeType
    - Return Value: BOOL
    - This function reverses the functionality of VirtualAlloc by freeing a block of size dwSize that starts at address lpAddress. If dwFreeType's value is MEM_DECOMMIT—0x4000—then the function de-commits the memory block but keeps it reserved for future use by process; if the value is MEM_RELEASE—0x8000—then the function frees the entire block—starting at lpAddress—back to the system. The function returns a false value if any parameters are invalid or if it is attempting to free unallocated memory.

Heap API

The heap memory API documentation is provided by Microsoft [Microsoft Corp. 2007a].

- HeapAlloc
    - Parameters: __in HANDLE hHeap, __in DWORD dwFlags, __in SIZE_T dwBytes
    - Return Value: LPVOID
    - This function is responsible for allocating a memory block of size dwBytes from the process's heap structure—identified by hHeap. Furthermore, if a heap memory block is successfully allocated it cannot be moved until it is freed. If the

function succeeds it returns the allocated block’s starting address, otherwise it returns a NULL pointer.

- HeapCompact
    - Parameters: __in HANDLE hHeap, __in DWORD dwFlags
    - Return Value: SIZE_T
    - This function reorganizes a heap structure by reducing free space between heap blocks.
- HeapCreate
    - Parameters: __in DWORD flOptions, __in SIZE_T dwInitialSize, __in SIZE_T dwMaximumSize
    - Return Value: HANDLE
    - This function creates a new heap object for a given process.
- HeapDestroy
    - Parameters: __in HANDLE hHeap
    - Return Value: BOOL
    - This function deletes a given heap object.
- HeapFree
    - Parameters: __in HANDLE hHeap, __in DWORD dwFlags, __in LPVOID lpMem
    - Return Value: BOOL
    - This function frees a block memory starting at address lpMem from the heap structure identified by the hHeap handle. The targeted block is not released back to the system, but rather returns to the heap’s total available memory.

- HeapReAlloc
    - Parameters: __in HANDLE hHeap, __in DWORD dwFlags, __in LPVOID lpMem, __in SIZE_T dwBytes
    - Return Value: LPVOID
    - This function changes the allocation properties of a given heap block.

Local & Global Memory API

The local and global memory API documentation is provided by Microsoft [Microsoft Corp. 2007a].

- LocalAlloc & GlobalAlloc
    - Parameters: in UINT uFlags, in SIZE_T uBytes
    - Return Value: HLOCAL
    - This function is designed to allocate a memory block of size uBytes. If the allocation succeeds, the function returns a handle to a new memory object.
- LocalFree & GlobalFree
    - Parameters: in HLOCAL hMem
    - Return Value: HLOCAL
    - This function frees the memory contained within the memory object—represented by the hMem handle. If the function succeeds it returns a NULL handle.
- LocalReAlloc & GlobalReAlloc
    - Parameters: __in HLOCAL hMem, __in SIZE_T uBytes, __in UINT uFlags
    - Return Value: HLOCAL
    - This function changes the allocation properties of a given memory block.

CRT Memory API

The CRT memory API documentation is provided by Microsoft [Microsoft Corp. 2007a].

- _alloca
    - Parameters: size_t size
    - Return Value: void pointer
    - This function allocates stack memory. It automatically releases its allocated memory and has no respective freeing function
- calloc
    - Parameters: size_t num, size_t size
    - Return Value: void pointer
    - Calloc is called by a process to request a memory block from its heap with all contents set to zero. If the allocation was successful the function returns the pointer to the requested block. The function returns a NULL pointer if there is no memory available for allocation.
- malloc
    - Parameters: size_t size
    - Return Value: void pointer
    - Malloc is called by a process to request a memory block from its heap. If the allocation was successful the function returns the pointer to the requested block. The function returns a NULL pointer if there is no memory available for allocation.
- free

    - Parameters: void *memblock
    - Return Value: None
    - This function is designed to free a memory block—specified by the pointer memblock—that malloc has previously allocated.
- realloc
    - Parameters: void *memblock, size_t size
    - Return Value: void pointer
    - This function changes the size of a given memory block.

APPENDIX B

Code Definitions for Original Memory Management Functions

```
#if _MSC_VER < 1300
LPVOID (WINAPI * Real_HeapAlloc)(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes)
    = HeapAlloc;
#else
LPVOID (WINAPI * Real_HeapAlloc)(HANDLE hHeap, DWORD dwFlags, DWORD_PTR
dwBytes)
    = HeapAlloc;
#endif

BOOL (WINAPI * Real_HeapFree)(HANDLE hHeap, DWORD dwFlags, LPVOID
lpMem)=HeapFree;

NTSTATUS (NTAPI * Real_NtAllocateVirtualMemory)(HANDLE,PVOID *, ULONG,
PULONG, ULONG, ULONG)=
      (NTSTATUS (NTAPI * )(HANDLE,PVOID *, ULONG, PULONG, ULONG, ULONG))
      DetourFindFunction("ntdll.dll","NtAllocateVirtualMemory");

NTSTATUS (NTAPI * Real_NtFreeVirtualMemory)(HANDLE,PVOID *,PULONG,ULONG)=
      (NTSTATUS (NTAPI * )(HANDLE,PVOID *,PULONG,ULONG))
      DetourFindFunction("ntdll.dll","NtFreeVirtualMemory");

void* (__cdecl * Real_malloc_msvcrt)(size_t size)=
      (void* (__cdecl *)(size_t size))
      DetourFindFunction("msvcrt.dll","malloc");

void (__cdecl * Real_free_msvcrt)(void* memblock)=
      (void (__cdecl *)(void* memblock))
      DetourFindFunction("msvcrt.dll","free");

void* (__cdecl * Real_malloc_MSVCR70)(size_t size)=
      (void* (__cdecl *)(size_t size))
      DetourFindFunction("MSVCR70.dll","malloc");;

void* (__cdecl * Real_aligned_malloc_msvcrt)(size_t size, size_t alignment)=
      (void* (__cdecl *)(size_t size, size_t alignment))
      DetourFindFunction("msvcrt.dll","_aligned_malloc");

NTSTATUS (NTAPI * Real_NtWriteVirtualMemory)(HANDLE,PVOID,PVOID,ULONG)=
      (NTSTATUS (NTAPI * )(HANDLE,PVOID,PVOID,ULONG))
      DetourFindFunction("ntdll.dll","NtWriteVirtualMemory");


VOID (NTAPI * Real_RtlFillMemory_ntdll)(PVOID, SIZE_T, BYTE)=
      (VOID (NTAPI * )(PVOID, SIZE_T, BYTE))
      DetourFindFunction("ntdll.dll","RtlFillMemory");
```

APPENDIX C

Code for a sub-set of Function Hooks: The code as been stripped down to core syntax.

```
#if _MSC_VER < 1300
LPVOID WINAPI
Hook_HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes)
#else
LPVOID WINAPI
Hook_HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD_PTR dwBytes)
#endif
{
    LPVOID rv = 0;
    __try {
        rv = Real_HeapAlloc(hHeap, dwFlags, dwBytes);
    } __finally {
        Syelog(---Print Line---);
    };
    return rv;
}

BOOL WINAPI
Hook_HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem){
    BOOL rv = 0;
    __try {
        rv = Real_HeapFree(hHeap, dwFlags, lpMem);
    } __finally {
        Syelog(---Print Line---);
    };
    return rv;
}

NTSTATUS NTAPI
Hook_NtAllocateVirtualMemory(HANDLE ProcessHandle,PVOID
                             *BaseAddress,ULONG ZeroBits,ULONG*
                             RegionSize,ULONG AllocationType,ULONG
                             Protect){
    NTSTATUS rv = 0;
    __try {
        rv = Real_NtAllocateVirtualMemory(ProcessHandle, BaseAddress,
             ZeroBits, RegionSize, AllocationType, Protect);
    } __finally {
            char* T="MEM_COMMIT";
            if(AllocationType==8192) T="MEM_RESERVE";
            if(AllocationType==524288) T="MEM_RESET";
            Syelog(---Print Line---);
    };
    return rv;
}
```

```
NTSTATUS NTAPI
Hook_NtFreeVirtualMemory(HANDLE ProcessHandle,PVOID *BaseAddress,PULONG
                         RegionSize,ULONG FreeType){
    NTSTATUS rv = 0;
    __try {
        rv = Real_NtFreeVirtualMemory(ProcessHandle, BaseAddress,
             RegionSize, FreeType);
    } __finally {
            char* T="MEM_DECOMMIT";
            if(FreeType==32768) T="MEM_RELEASE";
            Syelog(---Print Line---);
    };
    return rv;
}

void* __cdecl Hook_malloc_msvcrt(size_t size){
    void* rv = NULL;
    __try {
        rv = Real_malloc_msvcrt(size);
    } __finally {
        Syelog(---Print Line---);
    return rv;
}

void __cdecl Hook_free_msvcrt(void* memblock){
    __try {
        Real_free_msvcrt(memblock);
    } __finally {
        Syelog(---Print Line---);
}

void* __cdecl Hook_malloc_MSVCR70(size_t size){
      int STIME=TIME;
    void* rv = NULL;
    __try {
        rv = Real_malloc_MSVCR70(size);
    } __finally {
        Syelog(---Print Line---);
    return rv;
}

void* __cdecl
Hook_aligned_malloc_msvcrt(size_t size, size_t alignment){
    void* rv = NULL;
    __try {
        rv = Real_aligned_malloc_msvcrt(size, alignment);
    } __finally {
        Syelog(---Print Line---);
    };
    return rv;
}
```

```
BOOL WINAPI Hook_WriteProcessMemory(HANDLE hProcess,LPVOID
lpBaseAddress,LPCVOID lpBuffer,SIZE_T nSize,SIZE_T* lpNumberOfBytesWritten){
      int STIME=TIME;
    BOOL rv = true;
    __try {
        rv = Real_WriteProcessMemory(hProcess, lpBaseAddress, lpBuffer,
             nSize, lpNumberOfBytesWritten);
    } __finally {
        Syelog(---Print Line---);
    };
    return rv;
}

NTSTATUS NTAPI
Hook_NtWriteVirtualMemory(HANDLE ProcessHandle,PVOID BaseAddress,PVOID
                          Buffer,ULONG NumberOfBytesToWrite){
    NTSTATUS rv = 0;
    __try {
        rv = Real_NtWriteVirtualMemory(ProcessHandle, BaseAddress,
             Buffer, NumberOfBytesToWrite);
    } __finally {
        Syelog(---Print Line---);
    };
    return rv;
}
```

BIBLIOGRAPHY

APIMON (2006), "API Monitor," http://www.apimonitor.com/

Bazaz A. (2006), "A Framework for Deriving Verification and Validation Strategies to Assess Software Security," Unpublished PHD thesis, Virginia Polytechnic Institute and State University, VA.

BlindView Co (2006), "Strace," http://www.bindview.com/

Bosch, R.P. Jr. (2001), "Using Visualization to Understand the Behavior of Computer Systems," Ph.D. Dissertation, Stanford University.

CERT (2006), "Computer Emergency Response Team," http://www.cert.org/

ETF (2007), "eTForecasts," http://www.etforecasts.com

Foster, J.C., V. Osipov, N. Bhalla, and N. Heinen (2005), "Buffer Overflow Attacks," Syngress Publishing, Inc./Rockland, MA.

Graff, M.G. and K.R. Van Wyk (2003), "Secure Coding," O'Reilly & Associates Inc./Sebastopol, CA.

Harris, S., A. Harper, C. Eagle, J. Ness, and M. Lester (2005), "Gray Hat Hacking: The Ethical Hacker's Handbook," McGraw-Hill/Emeryville, CA.

Hoglund, G. and J. Butler (2006), "Rootkits," Pearson Education Inc./Upper Saddle River, NJ.

Hunt, G. and D. Brubacher (1999), "Detours: Binary Interception of Win32 Functions," Proceedings of the 3rd USENIX Windows NT Symposium, pp. 135-143, http://research.microsoft.com/sn/detours/

IWS (2007), "Internet World Stats," http://www.internetworldstats.com

Kaspersky Labs (2007), "Viruslist.com," http://www.viruslist.com

Microsoft Corp. (2007a), ".Net Library," http://msdn2.microsoft.com/en-us/library/default.aspx

Microsoft Corp. (2007b), "Memory Management: What Every Driver Writer Needs to Know," http://www.microsoft.com/whdc/driver/kernel/mem-mgmt.mspx

Microsoft Corp. (2007c), "Managing Virtual Memory in Win32," http://msdn2.microsoft.com/en-us/library/ms810627.aspx

NTinternals.net (2006), "Undocumented Functions for Windows," http://undocumented.ntinternals.net/

Rose, N.J (2000), "Hilbert-Type Space-Filling Curves," NC State University, http://www4.ncsu.edu/~njrose/

Shneiderman, B. (1996), "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations," Proceedings of the 1996 IEEE Conference on Visual Languages 336-343.

Software Verification (2006), "Memory Validator," http://www.softwareverify.com/cpp/memory/

Vostokov D. (2007), "Dump2Memory," http://www.dumpanalysis.org/

Wikipedia (2007), "The free encyclopedia," http://en.wikipedia.org/wiki/Main_Page

WINE (2007), "The Wine API Guide," http://source.winehq.org/WineAPI/