

To the Graduate Council:

I am submitting herewith a thesis written by Christopher Thomas Rathgeb entitled "Dynamic Application Level Security Sensors." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree Master of Science, with a major in Computer Engineering.

Gregory D. Peterson

Major Professor

We have read this thesis
and recommend its acceptance:

Brad Vander Zanden

David Icove

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original Signatures are on file with official student records.)

Dynamic Application Level Security Sensors

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Christopher Thomas Rathgeb

May 2010

Copyright © 2010 by Christopher Thomas Rathgeb

All rights reserved.

DEDICATION

This thesis is dedicated to my wife Caroline for all of her love, support, and encouragement; and to my son, Lennox for giving me another yet another reason to further my education.

ACKNOWLEDGMENTS

I would like to thank all of the people who helped and pushed me to complete my Masters of Science degree. First and foremost, I would like to thank my parents for their unconditional love and support. Ever since I was a child, they always told me I could do anything I put my mind to. Secondly, I would like to thank Dr. Craig Shue for critically reviewing draft copies of this work. Thirdly, I would like to thank Mr. Stephen Raub for helping me learn many of these concepts over the past several years. Fourthly, I would not have been successful had it not been for my advisor, Dr. Peterson. Finally, would also like to thank Dr. Vander Zanden and Dr. Icove for taking the time to serve on my committee.

ABSTRACT

The battle for cyber supremacy is a cat and mouse game: evolving threats from internal and external sources make it difficult to protect critical systems. With the diverse and high risk nature of these threats, there is a need for robust techniques that can quickly adapt and address this evolution. Existing tools such as Splunk, Snort, and Bro help IT administrators defend their networks by actively parsing through network traffic or system log data. These tools have been thoroughly developed and have proven to be a formidable defense against many cyberattacks. However, they are vulnerable to zero-day attacks, slow attacks, and attacks that originate from within. Should an attacker or some form of malware make it through these barriers and onto a system, the next layer of defense lies on the host. Host level defenses include system integrity verifiers, virus scanners, and event log parsers. Many of these tools work by seeking specific attack signatures or looking for anomalous events. The defenses at the network and host level are similar in nature. First, sensors collect data from the security domain. Second, the data is processed, and third, a response is crafted based on the processing. The application level security domain lacks this three step process. Application level defenses focus on secure coding practices and vulnerability patching, which is ineffective. The work presented in this thesis uses a technique that is commonly employed by malware, dynamic-link library (DLL) injection, to develop dynamic application level security sensors that can extract fine-grain data at runtime. This data can then be processed to provide stronger application level defense by shrinking the vulnerability window. Chapters 5 and 6 give proof of concept sensors and describe the process of developing the sensors in detail.

PREFACE

This document was written to help readers understand that gaps exist in current defensive techniques. It provides a possible solution to shrink the vulnerability window by filling one such gap. Several of the topics discussed throughout this thesis, such as code injection, reverse engineering, and shellcode examples, are taboo or carry negative connotations. Dynamic-link library injection is often used by attackers to execute code stealthily. Reverse engineering is often used to find attackable vulnerabilities or steal intellectual property. It is possible to take these techniques plus the additions presented in this thesis to create harmful software. For example, an attacker could use dynamic application level security sensors to harvest information, such as passwords, credit card numbers, and other important data. The intentions of this work are not to create a cookbook for developing harmful software but rather to introduce new methods to defend against malicious software. In light of these potential dangers, complete source code for the examples has not been provided.

Contents

1	Introduction	1
1.1	Significance	1
1.2	Approach	4
1.3	Assumptions	4
2	Existing Defensive Techniques	7
2.1	Network Level Security Domain	8
2.1.1	Network Intrusion Detection Systems	8
2.1.2	Network Level Security Domain Summary	10
2.2	Host Level Security Domain	11
2.2.1	System Integrity Verifiers	12
2.2.2	Antivirus Software	15
2.2.3	Host Level Security Domain Summary	18
2.3	Application Level Security Domain	18
2.3.1	Secure Coding Practices	19
2.3.2	Self-Defending Software	23
2.3.3	Static Application Level Sensors	23
2.3.4	Dynamic Application Level Sensors	24
2.3.5	Application Level Security Domain Summary	25
2.4	Conclusions	25

3	Dynamic-Link Library Code Injection	27
3.1	Types of Code Injection	27
3.1.1	SQL Injection	27
3.1.2	Cross-Site Scripting Injection	28
3.1.3	DLL Injection	30
3.1.4	Summary of Types of Code Injection	34
3.2	DLL Injection Techniques	34
3.2.1	Windows APIs	34
3.2.2	Windows Hooks	35
3.2.3	Summary of DLL Injection Techniques	35
3.3	Detecting DLL Injection	36
3.4	Existing Applications of DLL Injection	36
3.4.1	Malware	37
3.4.2	Functionality Additions	39
3.4.3	Summary of Existing Applications of DLL Injection	42
3.5	Dynamic Application Level Sensors Through DLL Injection	42
3.5.1	External Threats	42
3.5.2	Internal Threats	43
3.5.3	Summary of Dynamic Application Level Sensors Through DLL Injection	43
3.6	Conclusions	43
4	Developing Dynamic Sensors	45
4.1	Determine the Extractable Data	45
4.1.1	The Security Threat	46
4.1.2	The Application	47
4.1.3	Summary of Determining the Extractable Data	48
4.2	Reverse Engineer the Application	49
4.2.1	The Reverse Engineering Process	50
4.2.2	Reverse Engineering Tools	50

4.2.3	Reverse Engineering Methods	51
4.2.4	Summary of Reverse Engineering the Application	53
4.3	Automate the Data Extraction	55
4.3.1	Coding the Sensor	55
4.3.2	Signature Matching for Easy Update	57
4.3.3	Summary of Automating the Data Extraction	57
4.4	Analysis and Response	58
4.5	Conclusions	59
5	Experimental Sensor: PuttySensor.dll	61
5.1	Target Application	61
5.2	Reverse Engineering Putty.exe	63
5.3	Implementation	64
5.4	Results	65
5.5	Conclusions	68
6	Experimental Sensor: FunctionTimer.dll	69
6.1	Target Application	69
6.2	Reverse Engineering GuessNumber.exe	71
6.3	Implementation	72
6.4	Attack Vectors	73
6.4.1	Code Tamper Attack	74
6.4.2	Buffer Overflow Attack	74
6.5	Results	78
6.5.1	Results of Code Tamper Attack	79
6.5.2	Results of Buffer Overflow Attack	81
6.6	Conclusions	83
7	Summary and Conclusion	85
7.1	Summary	85

7.2	Future Work	88
7.3	Ethical Implications of this Work	89
7.4	Conclusion	91
	Bibliography	92
	Appendix	98
	A Screenshots of Reverse Engineering Tools	99
	Vita	105

List of Tables

6.1	Observed Execution Times for 32 Runs with a Code Tamper Attack at Run 27 . . .	80
6.2	Observed Execution Times for 32 Runs with a Buffer Overflow attack at Run 32 . .	82

List of Figures

1.1	Unique malware from 2006 to first half of 2009 seen by Avert Labs [28]	2
1.2	Underground malware black market posting of attacks on a website [12]	3
2.1	Three step process taken by defense systems	8
2.2	Architecture of OSSEC showing the three step process [33]	15
2.3	Example of compiler warning that could lead to unexpected behavior	20
3.1	Type of errors used in a full-view SQL injection attack	28
3.2	Cross-site scripting injection example	31
3.3	Thread level concurrency on single and multi-core processors	32
3.4	Result of helloworld.dll being injected into iexplore.exe	33
3.5	Effect of a thread bomb in action	39
3.6	Extended functionality of Task Manager [40]	40
3.7	MMORelay’s chat functionality addition through DLL injection	41
4.1	Reverse engineering feedback loop	50
4.2	Pictorial example of detours	52
4.3	Logic path alteration example	53
4.4	Iterative memory scanning with Cheat Engine	54
5.1	The Putty.exe configuration window	62
5.2	Screenshot of the user32.dll base address and GetDlgItem() function address	64
5.3	Popup box displaying the item id of the user interface element selected	66

5.4	Screenshot of the PuttySensor.txt log file produced by this sensor	67
6.1	GuessNumber.exe prompting the user for their name	70
6.2	Results of a complete cycle of GuessNumber.exe	70
6.3	Assembly code of Foo() prior to the code tamper attack	75
6.4	Assembly code of Foo() after the code tamper attack	76
6.5	Stack before buffer overflow	77
6.6	Stack after buffer overflow	77
6.7	Shellcode executing on the stack	78
6.8	Scatter plot of the timing results of the tamper attack	79
6.9	The watchdog timer detects that Foo() never returns and alerts the user.	81
6.10	Scatter plot of the timing results of the buffer overflow attack	83
A.1	Screenshot of the disassembler, IDA Pro	100
A.2	Screenshot of the memory editor, Winhack 2.0	101
A.3	Screenshot of memory editor and disassembler, Cheat Engine 5.5	102
A.4	Screenshot of the jump calculator, JumpCalc	103
A.5	Screenshot of the compiler, Visual Studio 2008	104

Chapter 1

Introduction

1.1 Significance

Cyberspace is an amorphous global interconnected network of networks that seems to be growing without bounds. Over the last decade, the breadth and complexity of cyberspace has vastly expanded to encompass more than just physically connected computers. The cyberspace domain now extends its reach to mobile devices, social networks, and other physical and virtual elements.

Accompanying this growth is an increase in cybercrime. Cybersecurity policies attempt to defend against attacks, disruptions, and other threats to information infrastructures. Such infrastructures include computers, computer networks, software systems, and the information they contain and communicate [9]. Recent federal policy documents emphasize the importance of cybersecurity to the welfare of modern society. They classify cyberattacks as “a new kind of threat to the safety and well-being of the United States,” and describe America’s failure to protect cyberspace as “one of the most urgent national security problems [26].” Further, these documents outline cybersecurity priorities and discuss the factors that contribute to the difficulty of securing cyberspace [4] [18] [20]. These factors include the increase in quantity and sophistication of cyberattacks and the complexity of the cyberdomain.

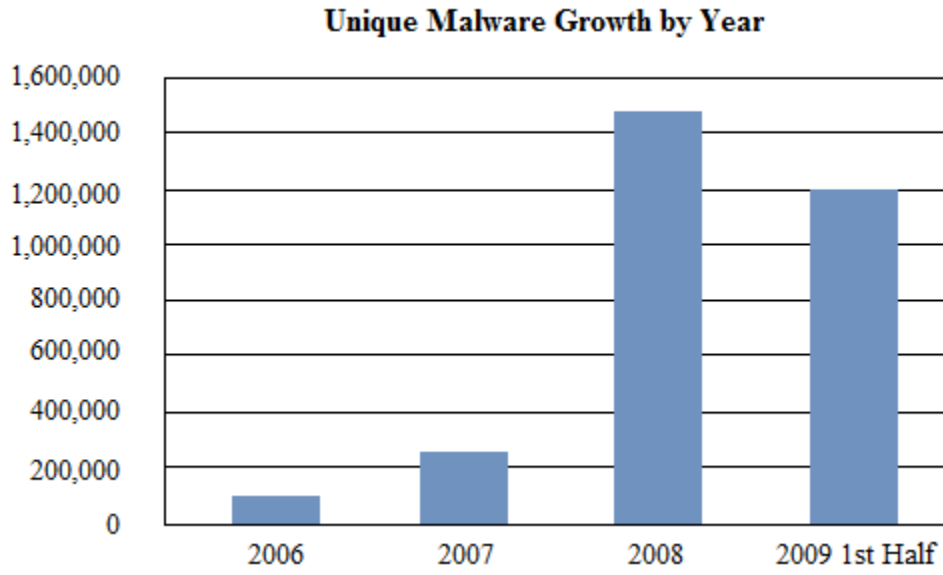


Figure 1.1: Unique malware from 2006 to first half of 2009 seen by Avert Labs [28]

Cyberattacks are growing exponentially in numbers. In 2008 malicious software increased an estimated 350% from levels in 2007 [12]. According to Kaspersky Lab analysts, there were 2,227,415 new malicious programs on the Internet in 2007, a four-fold increase on the results from 2006, with a volume of 354 GB [21]. In the first half of 2009, Avert labs saw almost as much new malware as in all of 2008 [28]. Figure 1.1 shows this growth. Panda labs predicted that malware in 2009 would grow and become more sophisticated and difficult to detect [34]. They claimed cyberspace would see an increase in web-based attacks, attacks through social networks, and attacks on non-Windows platforms such as Mac OS X Leopard, Linux, and iPhone. Each of these claims came true. In January 2009 Twitter suffered attacks that resulted in the compromise of thirty-three high profile accounts including an account belonging to President Barack Obama [39]. In 2009, unlocked iPhones, that is devices that have been unlocked from the manufacture’s proprietary restrictions, met their first worm, which changed the user’s operating system wallpaper to an image of 1980’s pop star Rick Astley with a message that said, “ikee is ever gonna give you up [13].”

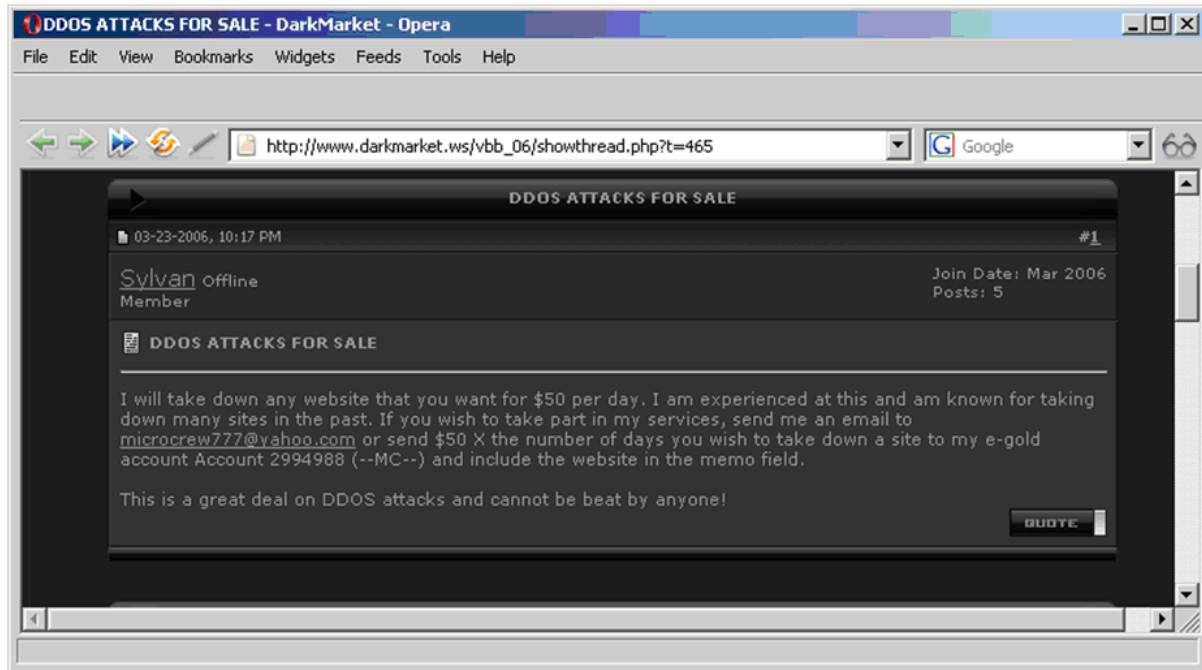


Figure 1.2: Underground malware black market posting of attacks on a website [12]

Second, the increasing sophistication of cyberattacks makes them harder to defend against. For example, malware can detect and neutralize antivirus software before the antivirus software detects the infiltration [30]. Other malware, such as the Storm worm, use polymorphic code that regularly transforms itself to thwart traditional antivirus and intrusion detection [43]. Further, malware authors are adopting software engineering's best practices that allow for faster deployment and higher quality code.

This increase in sophistication is driven by both political and economic agendas. Nation-states, militaries, and foreign intelligence services have the resources to commit acts of cyberwar through the anonymity of the internet. Quality malware can be purchased in underground software black markets. Botnet access can be purchased for as little as \$225, keystroke loggers for \$23, and phishing kits for \$10 [1]. If a hacker desires to take down a website, it would only cost \$50 per day it is down [12]. Figure 1.2 above shows a forum post of a hacker offering a distributed denial of service attack.

Finally, many levels of complexity makes securing the cyberdomain difficult. The information technology infrastructure is a complicated global system of hardware, software, operating systems, data, networks, and people. In Sally Floyd’s paper, *Difficulties in Simulating the Internet*, she attributes the challenges faced on internet modeling and simulation to the heterogeneity and dynamic nature of the networks. She details the internet as changing rapidly and unpredictably [11]. Further, a 2005 Congressional Research Service report for Congress stated that, “the complexity of cyberspace and its components, even within organizations, makes it both difficult to test and to predict how systems will behave under unusual circumstances, such as those that might rise from an unanticipated cyber attack [9].”

1.2 Approach

Many techniques exist for defending against cyberattacks. These techniques include firewalls, least user privileges, network intrusion detection systems, system integrity verifiers, secure coding practices, and vulnerability patching. Combining these elements forms a defense-in-depth security policy. At the network and host level, software-based intrusion detection systems collect data from customized sensors. The collected data is then processed and a response is crafted if any threats are detected. The application level focuses on secure coding practices and vulnerability patching. It lacks the sensors, analysis, and response of the network and host levels.

The granularity of the data collected depends on the security domain protected by the defensive mechanism. The work presented in this thesis expands the application level security domain to include sensors that are dynamically added to the application at runtime. This addition is accomplished through dynamic-link library injection and reverse engineering. The results are sensors that can extract fine-grain data at the application level.

1.3 Assumptions

There are many factors that contribute to the complexity of developing dynamic application level security sensors. These factors include but are not limited to the operating system, processor, and

target application. In order to simplify this complexity, the work presented in this thesis makes several assumptions.

The first assumption is that the operating system of the host machine is some version of Microsoft Windows. As of January 3, 2010, Microsoft Windows has a market share of 92.2%, making it the most common operating system in existence [23]. While the general methods described in this thesis are possible on operating systems other than Windows, the implementations of specific elements may vary. For example, the techniques for process injection differ between Linux and Windows. Additionally, there are differences among Windows versions that contribute to the complexity. For example, the virtual memory location for the start of the stack in Windows XP is static where as it is dynamic in Windows 7. While it is worth noting these differences, they are not extreme enough to warrant an assumption about the version of the operating system. The example presented in Chapters 5 was developed for Windows 7, and the example presented in Chapter 6 was developed for Windows XP.

The second assumption is that the processor of the host machine is either an x86-based Intel or AMD microprocessor. This assumption simplifies reverse engineering because it fixes the instruction set architecture (ISA). There are no assumptions about the number of cores on the system, however, the processor must support threading.

The third assumption is that the target application was developed in C, C++, or x86 assembly. Many newer programming languages, such as Java, Visual Basic .Net, and C#, have virtual run time environments or just-in-time compilers. This results in applications that are innately less vulnerable than traditional C or C++ applications.

The fourth assumption is that the injected binary code is in the form of a dynamic-link library. This simplifies the runtime process injection. It is possible to inject code that is in the form of an executable, string, or precompiled shell code, but these techniques are not addressed in this work.

Finally, the last assumption is that some form of code execution is already running on the system. In the case of stealthy execution, this may be in the form of an exploited vulnerability, and in the case of friendly execution, the code can be manually initiated.

These assumptions will help simplify the complexity of the addition of dynamic application level security sensors. They also focus the efforts of the work presented in this thesis. Keep in mind that many of these assumptions can be relaxed and that the methods can be implemented on other operating systems, processors, and forms of code. The next chapter will discuss the internal workings of several existing defensive techniques and the level at which they operate.

Chapter 2

Existing Defensive Techniques

There are many existing defensive tools, both commercial and open source, such as network intrusion detection systems, virus scanners, and anti-phishing filters. Each of these attempts to protect their particular security domains. Unfortunately attackers continue to capitalize on the copious weaknesses that are innate to these methods. By the time an attack is discovered, if one is discovered at all, the damage has already been done. The focus of this research is to shrink the vulnerability window by adding techniques that strengthen the application level security domain. The chapter discusses the spectrum of existing defensive techniques and highlights the need for application level security methods.

In general there is a three step process that is common in the existing defensive techniques. The first step is to collect data through custom sensors. The sensor then passes the collected data to the data processing unit for analysis. Based on the results of the analysis the defensive system may choose to respond to an event. This three step process, which is shown in Figure 2.1, is a reoccurring theme throughout this thesis, and can be seen in each of the defensive techniques detailed in this chapter. In order to better understand the new cybersensors presented in this research, it is important to understand the inner workings of existing defensive techniques and their security domains.

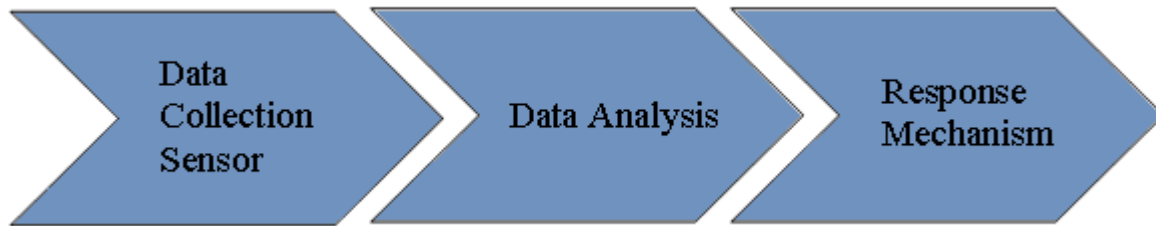


Figure 2.1: Three step process taken by defense systems

2.1 Network Level Security Domain

In the classic defense-in-depth strategy, network level protection systems are the “external layer focus area [29].” They are the first technological line of defense against outside attackers. The network level security domain has both hardware and software defenses that attempt to provide information assurance through a combination of firewalls, proxies, authentication techniques, role-based access controls, and network analyzers. In order to properly compare application level sensors to sensors in other security domains, this section will focus on sensor-based software defenses, particularly network intrusion detection systems.

2.1.1 Network Intrusion Detection Systems

Network intrusion detection systems (NIDS) work by monitoring network traffic. They look for suspect behavior such as denial of service attacks, worm propagations, or attempts to install a backdoor root kit. For example a large number of TCP connection requests to many different ports for a particular machine would indicate a TCP port scan, which is often used by worms attempting to propagate [14]. These intrusion detection systems work by parsing through network packet logs. Their full functionality can be generalized into the previously discussed three step process.

First, a sensor collects data from the network lines. The data is then displayed to the screen for human parsing, logged to disk for analysis later, or passed directly to the NIDS data analysis step. If an attack is successful, log files can be forensically studied in order to build a signature through

the attack sequence.

Next, the NIDS processes the collected data through protocol and content analysis. There are two major ways to perform this kind of analysis. Signature matching attempts to identify known attack sequences by comparing incoming network traffic to a database of previously detected attack sequences. It is not unusual for commercial NIDS to have large signature databases. The drawback to this technique is that it is limited to detecting attacks in the system's repository. Since the quantity of attacks is increasing, these systems quickly become outdated. It is crucial to keep the signatures, also referred to as rules, up to date.

Anomaly-based systems attempt to compare the incoming traffic to a profiled normal baseline for the network. This makes them prime candidates for detecting zero-day attacks. The problem with anomaly-based systems is that if they are trained using data tainted with attacks, they include those attacks in the baseline as normal behavior. This problem can be overcome through training in an isolated clean environment, and using signatures to check the training data.

Finally, the system responds to any detected anomalous behavior. In many cases a response may be as simple as notifying system administrators that something suspicious has been detected. This can be problematic if the NIDS floods the operators with false detections [31]. False positives is a problem all detection systems face, however at the network level the problem is heightened by the volume of data analyzed. It is not uncommon for the system to filter out the IP address of the attacker or terminate the TCP session.

Network intrusion detection systems are powerful tools. However, they are vulnerable to slow attacks and attacks that originate from within. Next, two examples of network intrusion detection systems are described.

Example: Snort - A Network Intrusion Detection System

Snort is an example of one of the most commonly used NIDS. It is an open source system that has over 270,000 registered users and millions of downloads worldwide [46]. It provides a lightweight

framework that uses both signature and anomaly-based inspection methods to perform traffic analysis and packet logging on IP networks. Snort can be configured to run in one of four separate modes. First, it can run in sniffer mode where the sensor simply reads the packets from the network and continuously displays them on the screen. Second, it can run in packet logger mode. In this mode it records all of the network traffic to the hard drive, which is particularly useful for forensically determining the cause of a successful attack. Third, it runs in NIDS mode where the sensor completes the three step process described at the beginning of this chapter. Finally, Snort can run in inline mode where it obtains packets from iptables instead of libpcap [47].

Example: Bro - A Network Intrusion Detection System

Bro is another example of a commonly used network intrusion detection system. Designed for UNIX, its functionality is similar to Snort. Bro introduces some powerful features that are not found in Snort, such as a custom scripting language known as “Bro language.” This scripting language allows users the ability to write or modify policies that govern the sort of activities deemed unacceptable on the network. It also includes a tool called “snort2bro” that converts Snort signatures into Bro signatures and reduces the number of false positives [3]. Even with these network intrusion detection systems, attacks are still successful.

2.1.2 Network Level Security Domain Summary

Network administrators attempt to defend the network level security domain from malicious outsiders through a combination of policy and hardware and software solutions. This section in particular focused on network level intrusion detection systems such as Snort and Bro. These systems are sensor-based software solutions that monitor network traffic in order to pull out anomalous behavior and defend against known cyberattacks. Unfortunately zero-day attacks, slow attacks, and attacks that originate from within are successful at defeating even the best NIDS. Should an attack successfully make it through the network level the subsequent layer of defense lies within the host. The next section will discuss defenses that operate within the host level security domain.

2.2 Host Level Security Domain

In the defense-in-depth paradigm, host level security domain protection falls into the “defending the computing environment” focus area [29]. This is perhaps one of the hardest domains to protect because it encompasses all physical, virtual, and social components tied to a host system. This is such a broad domain that the effects of a successful attack range in cost from inconsequential to catastrophic.

The physical elements tied to this domain include the hardware of the host system itself and physical devices controlled by the host. For example, the Easytune 5 software allows users to fine tune the speed of their CPU fan in order to allow for performance over-clocking. The existence of this software shows that malware could be written to disable the CPU fan of a computer system which could result in physical hardware failure due to overheating. In the case of a host that controls external physical devices the effects could be far more devastating. For example, supervisory control and data acquisition (SCADA) systems are host machines that control critical infrastructure such as power generators and water treatment facilities [41]. The compromise of a SCADA system could have damaging cascading effects.

Virtual elements tied to the host level security domain include critical data, which may contain intellectual property, and software running on the machine. Blackhat hackers, viruses, worms, trojans, and malicious insiders threaten all of these elements. An attacker could commit costly identity theft by compromising critical data such as credit card or social security numbers. Disgruntled employees and other malicious insiders threaten the intellectual property of businesses which can cost millions of dollars. According to an insider threat study conducted by the National Threat Assessment Center and CERT Coordination Center, losses from insider threats for several companies studied were in the tens of millions of dollars each [22]. Malware in the form of viruses, worms, and trojans attack software in order to destroy systems and facilitate information theft.

The social elements tied to the host level security domain include user computer education, usage tendencies, and reputations. Uneducated users who download and execute unknown attachments

sent to them via email create many new attack vectors for outsiders. Educating these users to safely operate their hosts is as important as running software to detect and remove viruses. In the earlier example of insider threat versus a business, social elements such as damage to the company's reputation are also a risk.

The growing influx of social networking websites has changed the usage tendencies of the average user. People are instantly connected to their friends and family through technology. Attackers use this to their advantage to help spread chaos in the form of spam email and malware. Should a hacker compromise someone's account on one of these sites, they can propagate malware to everyone connected to that victim's social network.

The increased number of attacks combined with the volume of potential attack vectors is overwhelming. Host level security measures attempt to protect all of these different elements through education, system integrity verifiers, and antivirus software. Again, in order to properly compare application level sensors to sensors in other security domains, the following section will focus on the sensor-based software solutions used to protect this domain, such as host-based intrusion detection and antivirus software.

2.2.1 System Integrity Verifiers

System integrity verifiers are more commonly referred to as host-based intrusion detection systems (HIDS). In this section both terms will be used interchangeably. The term "system integrity verifier" better describes the functionality of this defensive mechanism, while the term "host-based intrusion detection system" better describes the security domain. These systems monitor changes to system and log files, the registry, and processes running on the host [14]. They are designed to work alongside of network intrusion detection systems. While a NIDS protects the communication channels between different hosts on a network, a HIDS protects the host itself.

System integrity verifiers follow the same three step process described at the beginning of this chapter. First, data is collected from customized sensors. System integrity verifiers particularly rely

on data from several different sensors at the same time. For example, one sensor may monitor which processes are activated, and the length and activity levels of each process. Meanwhile a second sensor may monitor access logs to files located in the “C:\\Windows\\System32” directory. The fact that these systems collect data from many different sensors is beneficial in that it fully allows a system integrity verifier the ability to probe the system with seemingly limitless possibilities. However, it also sheds light onto the complexity of a working host. The combination of multiple versions of different software and ranging computer skills and security awareness opens up a large number of potential vulnerabilities. The exorbitant size of the possible attack space is overwhelming, and determining where to place sensors is challenging.

Next, the collected data is processed. The type of processing depends largely on the collected data. For example, changes that are made to a system file could use two types of processing. One type of processing may be checksum verification, while a second type may be to check this access against anomaly based detection rules. Another example is a sensor that monitors port activity. Such a sensor could be implemented at either the network or host level. The processing stage for this may determine when specific ports are properly or improperly accessed [19].

Finally, the third step is to respond to any suspect behavior on the system. Possible responses are notifying administrators, killing a process, blocking port access, or other actions that protect against the threat.

Example: A HIDS vs. The Blaster Worm

The Blaster worm is a computer worm that exploits a buffer overflow in the DCOM RPC service on Windows XP and Windows 2000 machines. It surfaced in late 2003 and spread by spamming itself to random IP addresses [2]. A system integrity verifier could be used to detect the Blaster worm. In order to demonstrate this it is important to know how the Blaster worm works. According to an analysis performed by eEye Digital Security the Blaster worm had the following behavior [10]:

1. Exploit RPC DCOM buffer overflow that binds a system level shell to port 4444 in order to set up the command channel from the infecting agent and the vulnerable target which is used

to transmit msblast.exe via TFTP.

2. Create the registry value “HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\windows auto update” with the value “msblast.exe”, which causes the executable to be automatically run each time the machine boots.
3. Create a mutex named “BILLY” as a way of notifying new instances of Blaster that the target has already been infected.
4. Initialize Winsock in order to check that the new host has network access.
5. Seek out new hosts by scanning IP addresses.

Knowing the functionality of the Blaster worm makes it easy to see how specific sensors could easily detect it. A sensor that monitors unauthorized access to ports, particularly port 4444, would detect this worm. Scanning for changes to the registry or initializations of Winsock would also detect Blaster. Another host level method for detecting this type of attack would be the antivirus software approach, which is to seek out a code signature that would match msblast.exe on the disk. All these sensors could help defend against the Blaster worm.

Example: OSSEC A Host Based Intrusion Detection System

OSSEC is an example of an open source cross platform general purpose host-based intrusion detection system used by internet service providers, governments, universities, and even commercial companies. It combines log file analysis, file integrity checking, policy monitoring, and rootkit detection to provide real time active response [33]. Its architecture makes use of the three step process described throughout this chapter. A small client program, called an “agent”, is installed on the computer to monitor and collect information. These agents are the customized sensors described in the first stage of the three step process. Next, the information collected from agents is passed to the “manager”, which is a program residing on the server that centrally performs the second step, data analysis, for all of the agents connected to the OSSEC server. This is a useful feature because it provides the ability to perform correlation analysis between agents. From this analysis

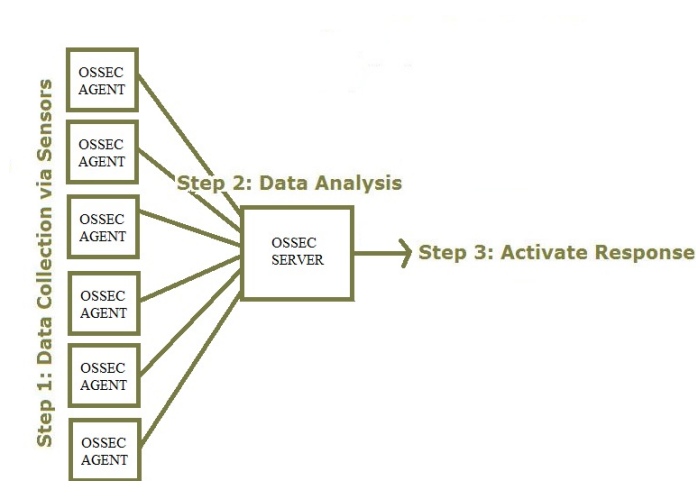


Figure 2.2: Architecture of OSSEC showing the three step process [33]

the system performs the third step by activating a response when something happens. This may be in the form of blocking an attack or emailing an alert to a system administrator [33]. Figure 2.2 shows the architecture of OSSEC and how it uses the three step process. OSSEC has several other features, not described here, that assist the general three step process, such as agentless monitoring, configurable alerting, and virtualization support. Even with all of these features attacks are still successful. This is the reason defense-in-depth is used. What one level of protection does not catch the next level should.

2.2.2 Antivirus Software

Antivirus software is a host level security domain protection mechanism that attempts to combat malicious code and software by scanning the file system for potentially harmful files. Antivirus software also follows the three step process. The sensor data step is the collection of binary files to be scanned. The data analysis step is the process of searching the files for code signatures that are known to be harmful. Should the antivirus software find a match, it fulfills the third step by taking the appropriate action that matches the threat. In most cases the action is to delete the bad file and notify the user.

Antivirus software works by matching code signatures. While signature matching is highly successful at detecting specific attacks, it fails to detect new viruses and polymorphic code, which is code that mutates through obfuscation. In order to be effective, antivirus software must maintain an updated signature for each potential threat.

Example: Code Signature Matching and Polymorphic Code

The following example has been crafted to help understand how virus scanners work and why they can easily detect previously seen code but not new or mutating code. Take the following snippet of C++ code that loops a compound addition ten times:

Code: C++ Implementation

```
int a = 3;

for(int i=0;i<10;i++) {
    a = a + i;
}
```

When a program is compiled into an executable, machine readable code is created. The assembly code for this particular C++ sequence is shown below.

Code: x86 Assembly Representation

```
.text:0041138E  mov [ebp+var.8], 3      // Assign 3 to a variable. In our case 'a'
.text:00411395  mov [ebp+var.14], 0     // Assign 0 to loop variable 'i'
.text:0041139C  jmp short loc.4113A7    // Jump to first loop condition check.
.text:0041139E  mov eax, [ebp+var.14]   // Move value of variable 'i' into register eax
.text:004113A1  add eax, 1             // Add 1 to register eax
.text:004113A4  mov [ebp+var.14], eax   // Move new eax value back into variable 'i'
.text:004113A7  cmp [ebp+var.14], 0Ah   // Loop condition check. Compare 'i' to value 10
.text:004113AB  jge short loc.4113B8    // If 'i' is >= 10, exit loop
.text:004113AD  mov eax, [ebp+var.8]    // Move value of variable 'a' into register eax
.text:004113B0  add eax, [ebp+var.14]   // Add value of variable 'i' to register eax
.text:004113B3  mov [ebp+var.8], eax    // Move new value of eax into variable 'a'
.text:004113B6  jmp short loc.41139E    // Goto next loop iteration
```

This assembly code can also be translated into a hexadecimal representation of what the computer actually interprets as seen in the highlighted code below.

Code: Hexadecimal Representation of Assembly

```
.text:00411380  FF FF B9 36 00 00 00 B8 CC CC CC CC F3 AB C7 45
```

```
.text:00411390 F8 03 00 00 00 C7 45 EC 00 00 00 00 EB 09 8B 45
.text:004113A0 EC 83 C0 01 89 45 EC 83 7D EC 0A 7D 0B 8B 45 F8
.text:004113B0 03 45 EC 89 45 F8 EB E6 33 C0 5F 5E 5B 8B E5 5D
.text:004113C0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC
.text:004113D0 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
```

Static pieces of the code, such as instruction opcodes and integer values, and the sequence in which they occur can be used to construct a specific code signature for this C++ code snippet. Memory addresses, the registers, and other variable pieces of information can be represented by don't care values. The signature will be constructed from the hexadecimal representation where variable elements are replaced with an "X" to represent a don't care value.

Code: Hexadecimal Signature Sequence

```
"C7 XX XX 03 00 00 00 C7 XX XX 00 00 00 00 EB XX 8B XX XX 83 XX XX 89 XX XX 83 XX XX 0A 7D
XX 8B XX XX 03 XX XX 89 XX XX EB XX"
```

Now that a signature has been created, this code snippet can be detected by a program that traverses through an executable looking for a match to this particular byte sequence. This illustrates how a virus scanner can detect malicious code segments that have previously been encountered and have signatures. It is also easy to see how malware that has not been seen before will slide past this method of detection.

If this code were to become mutated through polymorphism or static obfuscation the signature would not match. For example, the following code example is an assembly representation containing trivial nop obfuscation.

Code: x86 Assembly Representation with Trivial Obfuscation

```
.text:0041138E mov [ebp+var_8], 3 // Assign 3 to a variable. In our case 'a'
.text:00411395 mov [ebp+var_14], 0 // Assign 0 to loop variable 'i'
.text:0041139C jmp short loc_4113A7 // Jump to first loop condition check.
.text:0041139E mov eax, [ebp+var_14] // Move value of variable 'i' into register eax
.text:004113A1 add eax, 1 // Add 1 to register eax
.text:004113A4 mov [ebp+var_14], eax // Move new eax value back into variable 'i'
.text:004113A7 cmp [ebp+var_14], 0Ah // Loop condition check. Compare 'i' to value 10
.text:004113AB jge short loc_4113B8 // If 'i' is >= 10, exit loop
.text:004113AD mov eax, [ebp+var_8] // Move value of variable 'a' into register eax
.text:004113B0 add eax, [ebp+var_14] // Add value of variable 'i' to register eax
.text:004113B3 add eax, 10 // add 10 to register eax
.text:004113B6 sub eax, 10 // sub 10 from register eax
```

```
.text:004113B9  mov [ebp+var.8], eax  // Move new value of eax into variable 'a'
.text:004113Bc  jmp short loc_41139E  // Goto next loop iteration
```

In the above code the two inserted instructions cancel each other out and the resulting functionality is the same as the original function. These two lines of assembly code however, change the hexadecimal representation in a way that would not match the original signature.

This is a trivial example that shows the strengths and weaknesses of antivirus detection systems. In the underground malware black market it is not uncommon for the same pieces of code to be reused. This means signature-based detection methods should be able to detect the malicious code. However, malware is becoming increasingly complex due to techniques like polymorphism and obfuscation. Ten lines of assembly can be dynamically turned into hundreds or thousands of lines of functionally equivalent code in order to avoid antivirus detection.

2.2.3 Host Level Security Domain Summary

The host level security domain is incredibly far reaching. The dynamic nature of a host system and the user controlling it creates many opportunities for attacks. Attackers can quickly capitalize on the vulnerabilities that surface. This makes defending these systems a challenge. System integrity verifiers attempt to probe a system's resources to determine if the host is functioning properly. They do this through a combination of sensor data extracted at the host level and custom analysis. Antivirus software seeks out malicious code and attempts to block it from harming the system. Well-designed malware can avoid detection entirely or even disable the antivirus software. The next section will discuss the application level security domain and measures it takes to detect and defend against cyberthreats.

2.3 Application Level Security Domain

The application level security domain encompasses defenses that are related to software applications running on a host machine. It subsumes the methods programmers take to secure the code they develop. These methods include sensors statically coded into the software, self-defending mecha-

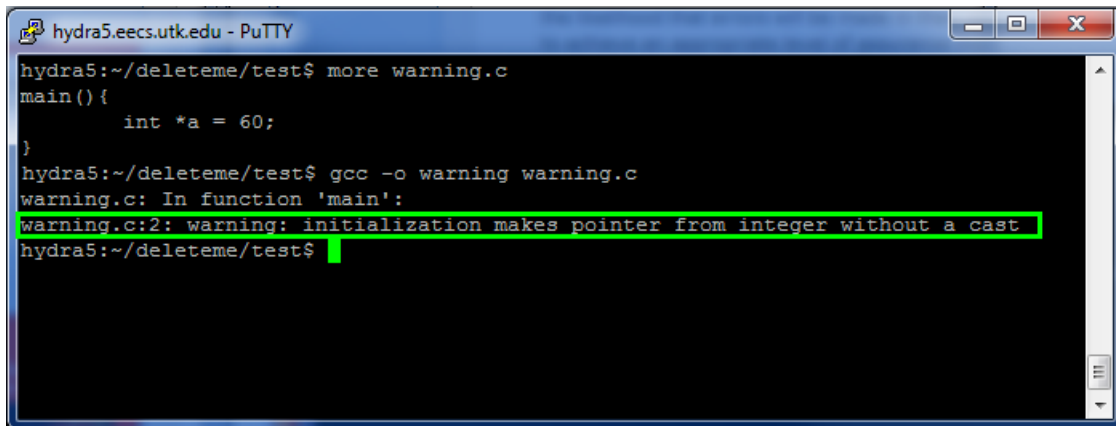
nisms, and other coding practices resulting in safe applications. This level also extends beyond the development process into the runtime environment. The runtime environment includes the virtual memory associated with the application’s process and the state of the application while it is executing.

Robust application level security is rare due to the nature of the software development cycle. Programmers are more concerned with getting applications to work than getting them to work securely. Many times security is pushed aside during the development cycle due to the extra monetary overhead of verifying a software system. This is one of the reason that seventy-five percent of vulnerabilities and ultimately hacks come from the application level security domain [27]. There exists a large gap in the number of vulnerabilities that manifest from application level security and the number of measures to protect against these vulnerabilities. This section discusses several of the techniques that can be used to defend the application level security domain. It also introduces the focus of this thesis, dynamic application level sensors.

2.3.1 Secure Coding Practices

The primary reason vulnerabilities manifest is the lack of secure coding practices. There are several contributing factors that lead to this shortcoming such as budget, complexity, lack of policy, ignorance, or simply laziness. Each of these must be overcome in order to develop secure software. CERT publishes a list of the top ten secure coding practices and recommends following them during the development cycle to greatly reduce the number of vulnerabilities that arise from an application [44].

Input validation is the first practice [44]. Validating all input from untrusted sources can eliminate the majority of software vulnerabilities. These types of vulnerabilities are flaws that result from “implicit assumptions made by the application developer about the application input [36].” Buffer overflows are an example of an invalid assumption about the maximum size of an input [36]. In the case of web-based applications, failure to validate input is the reason SQL injection and cross-site scripting (XSS) attacks are successful. The following is an example of a basic SQL

A screenshot of a terminal window titled 'hydra5.eecs.utk.edu - PuTTY'. The terminal shows a user at the prompt 'hydra5:~/deleteme/test\$' typing 'more warning.c'. The content of 'warning.c' is displayed: 'main() { int *a = 60; }'. Then, the user types 'gcc -o warning warning.c'. The compiler output is shown: 'warning.c: In function 'main': warning.c:2: warning: initialization makes pointer from integer without a cast'. The warning message is highlighted with a green box. The prompt 'hydra5:~/deleteme/test\$' is visible at the bottom.

```
hydra5:~/deleteme/test$ more warning.c
main() {
    int *a = 60;
}
hydra5:~/deleteme/test$ gcc -o warning warning.c
warning.c: In function 'main':
warning.c:2: warning: initialization makes pointer from integer without a cast
hydra5:~/deleteme/test$
```

Figure 2.3: Example of compiler warning that could lead to unexpected behavior

injection. Assume the application that fails to check inputs is constructing an SQL query string such as `“SELECT * FROM users WHERE name = “ + userName + ““;”`. Should the user input their name as `“‘a’ OR ‘t’=‘t’“` the resulting SQL query string would appear to be `“SELECT * FROM users WHERE name = ‘a’ OR ‘t’=‘t’;”` which would return more information than should be displayed to the individual executing the attack [48]. The same technique can be used to bypass password authentications on sites that did not validate the input.

Heeding compiler warnings is the second practice that can help reduce the number of software vulnerabilities [44]. Many developers ignore compiler warnings. Compiling with the highest warning levels available and fixing the results that the compiler flags can reduce the number of potential attack vectors. A good example of this is a C or C++ compiler that allows the user to write a value into a variable that should be a pointer. This forces the pointer to point to some arbitrary location in memory. Syntactically this may be acceptable but most compilers will throw a warning for this type of action. Ignoring this type of warning could cause an application to have potentially unexpected behavior or unanticipated vulnerabilities. For example Figure 2.3 shows the warning produced for the C++ code `“int *a = 60;”` when compiled with gcc at standard warning levels.

Architect and design for security policies is the third secure coding practice that CERT describes [44]. They suggest creating a software architecture that allows the developer to implement

and enforce security policies. Moreover, they give the following example: “if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set [44].” In the given example, if one subsystem were to become compromised the other subsystems would be able to continue operations. In addition, wherever communication is involved between the sub systems data sanitization should be implemented.

Keeping designs simple is the fourth secure coding practice [44]. By creating uncomplicated designs fewer errors surface. More complex designs lead to a larger number of errors that can be harder to find. Complexity is often unavoidable for functions or applications. For this reason reducing complexity is relative to how much complexity can be reduced.

Defaulting to deny is the fifth secure coding practice [44]. The idea behind this practice is to not trust the user. When determining whether or not to allow access to a system, rather than assuming the user should have access unless they give an indication that they should not, assume access is not granted unless the user proves they should have access.

Adherence to the principle of least privilege usage is the sixth secure coding practice [44]. Running every application with the smallest amount of privileges required to operate results in dilution of successful attacks. Attackers who successfully exploit an application with low privileges are not able to do as much damage as an attacker who exploits an application with system level or root privileges.

Sanitizing data sent to other systems is the seventh secure coding practice [44]. Sanitizing data passed to subsystems prevents attacks from potentially invoking unused functionality through context misunderstandings [44]. In multi-component systems inputs and outputs may be handled in different ways. In order to ensure the target subsystem functions properly data sanitization should be handled before input is passed to it.

Practicing defense in depth strategies is the eighth secure coding practice [44]. If an upper level defense becomes compromised a lower level defense layer should attempt to defeat attack.

Use effective quality assurance techniques [44]. This is the ninth secure coding practice and one that is often overlooked. High quality assurance testing can be effective at finding and neutralizing vulnerabilities before software is deployed for usage. Validation and verification are a critical stage in the software development cycle that is often overlooked. In complex systems exhaustive testing may not be feasible. For this a combination of random and end case test vectors and boundary scanning should be used. Quality testing should take place on different platforms with common software configurations.

Adopting a secure coding standard is the tenth and final practice published by CERT [44]. Adherence to strict coding standards helps identify abnormalities in code that could potentially lead to vulnerabilities. An example of this is to consistently verify the success of a dynamic memory allocation.

These ten coding practices are preventative measures that help fortify code against attackers. When combined with knowledge of common insecure functions such as `strcpy()`, which will be shown to be dangerous in Chapter 6, these methods can greatly reduce the number of vulnerabilities in a program. Unfortunately, these methods are only truly effective if followed during the development cycle. When a vulnerability is discovered after a software system has been deployed, other defensive measures must be taken into consideration.

Since 2003 Microsoft has released its security updates on the second Tuesday of each month known as "Patch Tuesday [35]." Vulnerability patching is the reactive measure software developers take to fix mistakes they made in the original development of their software and prior patches. Quickly patching known vulnerabilities is a necessity for securing the application level security domain.

2.3.2 Self-Defending Software

Self-defending and self-healing software is code developed to detect, defend, and recover from attacks. This resilience is most commonly used by host-based intrusion detection systems and antivirus software. The increasing sophistication of malware has led to malicious code that is capable of seeking out defensive mechanisms and deactivating or functionally disabling them. This can be accomplished in several ways, first by attempting to kill a process through a library call to `TerminateProcess`, `TerminateThread`, `ExitProcess`, or `SuspendThread` [24]. Another way is to inject code into the process of the antivirus application that modifies the logical functionality of the program at run time. This is potentially a way of functionally disabling the antivirus software even though it is still technically running. In a 2004 study that tested the ability of several common antivirus packages' to recover from attack attempts, Kaspersky labs' antivirus software was the only one capable of surviving attacks launched even with administrator privileges. Several other antiviral applications were easily defeated and others had mixed results. With local administrator privileges F-Secure and Pestpatrol were deactivated, while Panda and Trend Micro's antivirus protection was automatically restarted after 30 seconds. Grisoft and Symantec continued functioning though their processes were killed [45]. This study shows that several of the tested antivirus software packages are capable of self-defense.

This kind of defense is ideal not only for antivirus software but also for general purpose application development. This will require sensors that actively seek vulnerabilities and attacks in progress, in an effort to automatically fix and recover from them. It is unrealistic to expect a host level security system to perform this sort of fine grain monitoring on all the different applications running on a system. For that reason, these sensors need to be built into the applications themselves or added dynamically at runtime.

2.3.3 Static Application Level Sensors

Static application level sensors are coded into the program during the development cycle. They report information related to functional operation of the application at runtime. For example,

web applications deployed in Apache often report information pertaining to database connections, access attempts, and other functionality. The problem with these static application level sensors is they often are being used improperly. Most of the reports created from these application level events are used only for debug purposes and the logs created are never examined unless absolutely necessary. If an attacker is attempting a brute force password attack on an Apache web application sensors could easily be developed into the system to detect this. More application level sensors need to be implemented during the development cycle. Also the three step process that is seen at the other security level domains needs to be implemented. Rather than simply logging these events to a file, the application should run automatic analysis to discover any inconsistencies in expected execution as it runs. The result of this analysis could then trigger a response, possibly in the form of a code sequence that immediately neutralizes a suspected attack in progress.

2.3.4 Dynamic Application Level Sensors

Many applications running on a computer system do not readily have their source code available to add static event sensors. This is the case with common proprietary applications, such as the Microsoft Office suite or with legacy applications in which source code is no longer available. The work presented in this thesis proposes a solution to this problem that is based on the combination of reverse engineering and dynamic-link library (DLL) code injection, which will be discussed further in Chapter 3. The basic idea is to create an execution thread inside an application's process that dynamically deploys customized sensors. The locations of these sensors are found by reverse engineering the application and discerning where critical data and functions reside in the virtual memory space. After these sensors are deployed analysis is run on the data they produce to develop active responses to potential threats. It is understandable why reverse engineering is required for legacy software but in the case of proprietary software these type of defenses should be built into the application. If the vendor fails to do this they should at least provide the proper hooks to add them dynamically.

2.3.5 Application Level Security Domain Summary

The majority of software vulnerabilities manifest within the application level security domain. Yet it is this domain that is the least protected. Most of the defenses in this domain are focused on avoiding common coding mistakes that lead to vulnerabilities. The combined knowledge of common exploits with a secure coding practice can greatly reduce the number of problems in a program. However this is not a total solution because as the complexity of architectures increases unexpected behavior can emerge. The next application level defensive measure deals with developing patches in a timely manner for code that is known to be vulnerable. Major software companies have been doing this for many years. “Patch Tuesday,” the second Tuesday of each month, has become the official exploit correction day for Microsoft. Unfortunately, counting on developers to not make mistakes and simply fixing them when they do is the reason attackers always have the upper hand. Attackers show developers what needs to be patched by exploiting the vulnerabilities prevalent in the software. The problem is that the damage done in the initial attack can be costly.

New application level defensive mechanisms need to further explore the concept of self-healing code. Code that can recover from an attack is impressive in of itself but extending that capability to develop code that can monitor its own health and deploy dynamic defenses depending on the results of that health monitor would be a significant technical advancement. This capability could be accomplished through application level sensors. The functionality could be built into new programs and dynamically added to existing programs at run time.

2.4 Conclusions

There are many existing techniques for defending the many security domains. When combined these form the standard defense in depth idea of cybersecurity. The strongest defenses follow the three step process described throughout this chapter. Step one is to acquire data through a customized sensor. Step two is to analyze that data, and step three is to respond to any threats that the analysis has detected.

At the network level security domain the three step process is seen in many of the network based intrusion detection systems. The sensors collect packet information that is then passed on to signature and anomaly based analysis systems. From there the potential problems are filtered out and administrators are notified. Should defenses at this level fail the subsequent layer of defense is the host level security domain.

The host level has many different software based solutions for protecting a workstation, such as host based intrusion detection systems and antivirus software. Each of these methods follows the three step process: collect, analyze and respond. In the case of the host based intrusion detection system, sensors come from many different sources such as process analytics, file system monitors, and many other locations.

The next level of system defense is the application level where the three step process is not well implemented. Developers are given the responsibility to not make mistakes in their code that could lead to problems. If a mistake is made it is patched after deployment. The defensive measures are centered more around coding properly the first time and less on detecting problems as or before they are exploited. One possible solution is static and dynamic application level sensors. The advent of multiprocessor systems opens the door for processor cores that are dedicated to securing the applications running on a system. Threads of execution can be spawned inside the virtual memory space of processes running on the system in order to perform security analysis. The next two chapters will explore the required technologies to efficiently implement dynamic application level sensors.

Chapter 3

Dynamic-Link Library Code Injection

Dynamic-link library injection is the sequence of steps taken to forcibly load code contained in a DLL into the virtual memory space of another process. Attackers have been using this technique for many years to execute malicious code without detection. For that reason code injection has acquired negative connotations. However, this same technique can be used defensively to develop dynamic application level security sensors. Chapter 3 discusses the different types of code injection and particularly details DLL injection.

3.1 Types of Code Injection

Code injection comes in several forms such as SQL, XSS, and DLL injection. Each of these methods will be explained further, but the form this thesis will focus on is DLL injection. Understanding the similarities and differences of other injection methods is important to the understanding of how DLL injection is a unique class of code injection and how it can be used defensively.

3.1.1 SQL Injection

SQL injection is a form of injection intent on attacking databases. Attacks can result from the failure to validate user input. Attackers use SQL injection in an attempt to insert, delete, modify, or extract data. In 2004, this type of attack was used to collect 40 million credit card numbers from a CardSystems, Inc database [32].

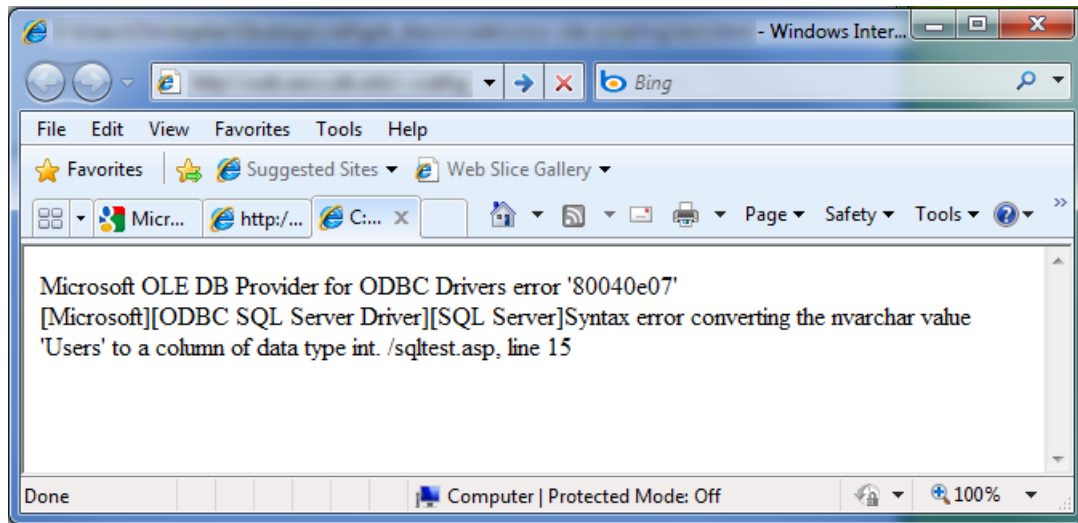


Figure 3.1: Type of errors used in a full-view SQL injection attack

In general there are two common types of SQL injection attacks: full-view attacks and blind attacks. Full-view attacks probe the system with inputs and look at the error messages reported to determine how to formulate their attack. Figure 3.1 shows an example of the kind of output used in full-view attacks. Blind SQL injection attacks are used when web applications are developed in such a way that prevents error messages from being displayed. Table names and contents are not known in these types of attacks, therefore a blind SQL attack probes the system with boolean queries to extract information one character at a time [32]. An example of an SQL injection attack can be found in the discussion of input validation in Chapter 2 on page 17.

SQL injection attacks are fairly noisy and easy to detect. Many times, defending against them is as easy as input validating. In other cases, parsing web server log files or building signatures for NIDS or HIDS can defend against these types of attacks.

3.1.2 Cross-Site Scripting Injection

Cross-site scripting injection attacks are another variant of code injection that relies on a web application's failure to validate user input. Eighty-four percent of the web-based vulnerabilities documented by Symantec during the last six months of 2007 were cross-site scripting vulnerabilities

[52]. The intended targets of these attacks are the users who visit a website after the attack has occurred. XSS attacks are successful due to the nature of web browsers. When a user accesses a webpage, the HTML code is downloaded and interpreted by the web browser. Scripting languages can be embedded into the website’s markup language in several ways, such as `<script>` tags and attributes such as, `style` or `src`. On webpages that store and display user input such as blogs, auctions, or comment sections, attackers can input `<script>` tags with malicious script code. If the site fails to validate the attacker’s input, other users that view the site later download the HTML with these `<script>` tags containing malicious code.

XSS attacks are difficult to protect against for several reasons. First, neither encryption nor firewalls can protect against them. Second, there are many ways to trick browsers into executing the malicious code, and third, these attacks are easily disguised [53].

Example: Cross-Site Scripting

The following example is of a basic XSS injection. An HTML page prompts the user to input their name. When the user presses submit, it sends the value of name to a PHP script that generates a custom welcome page with the user’s name. The file “input.html” is the markup for the initial page that prompts the user for their name.

Code: input.html

```
<html>
  <body>
    Please Enter Your Name.
    <form name="input" method="post" action="name.php">
      <p>Name: <input type="text" name="name" size="70"/></p>
      <p><input type="submit" value="Submit" /></p>
    </form>
  </body>
</html>
```

The following code segment shows a valid input string that would result in a basic cross-site scripting attack.

Code: input string

```
<p onmouseover=javascript:alert(911) > Christopher T. Rathgeb </p>
```

When the user hits submit on the webpage, the input string is sent to “name.php” which embeds the user input into the customized page.

CODE: name.php

```
<?php
    echo “ <html> <body>”;
    echo “Welcome: ”. $_POST['name'];
    echo “</body> </html>”;
?>
```

The markup for the page that “name.php” generates is seen below. The embedded javascript can be seen inside it. When the user moves their mouse over the name a popup window is thrown to the screen.

Code: generated HTML page

```
<html>
    <body>
        Welcome:
        <p onmouseover=javascript:alert(911)> Christopher T. Rathgeb </p>
    </body>
</html>
```

This sequence of events can be seen in Figure 3.2.

3.1.3 DLL Injection

The SQL and XSS injection methods are both targeted towards web-based applications. DLL injection differs in that it targets the processes of applications running on a system. It forces the execution of precompiled code inside the virtual memory space of another application’s process. This process is accomplished using multi-threading.

Threads are alternate execution streams that run concurrently or pseudo-concurrently with their parent process. Whether or not the execution streams are concurrent or pseudo-concurrent depends on factors such as, the processor, operating system, and thread model. Essentially they are lightweight processes that share state, memory address space, and other resources within their parent process. Multi-core processors offer concurrency by tasking each core with different threads

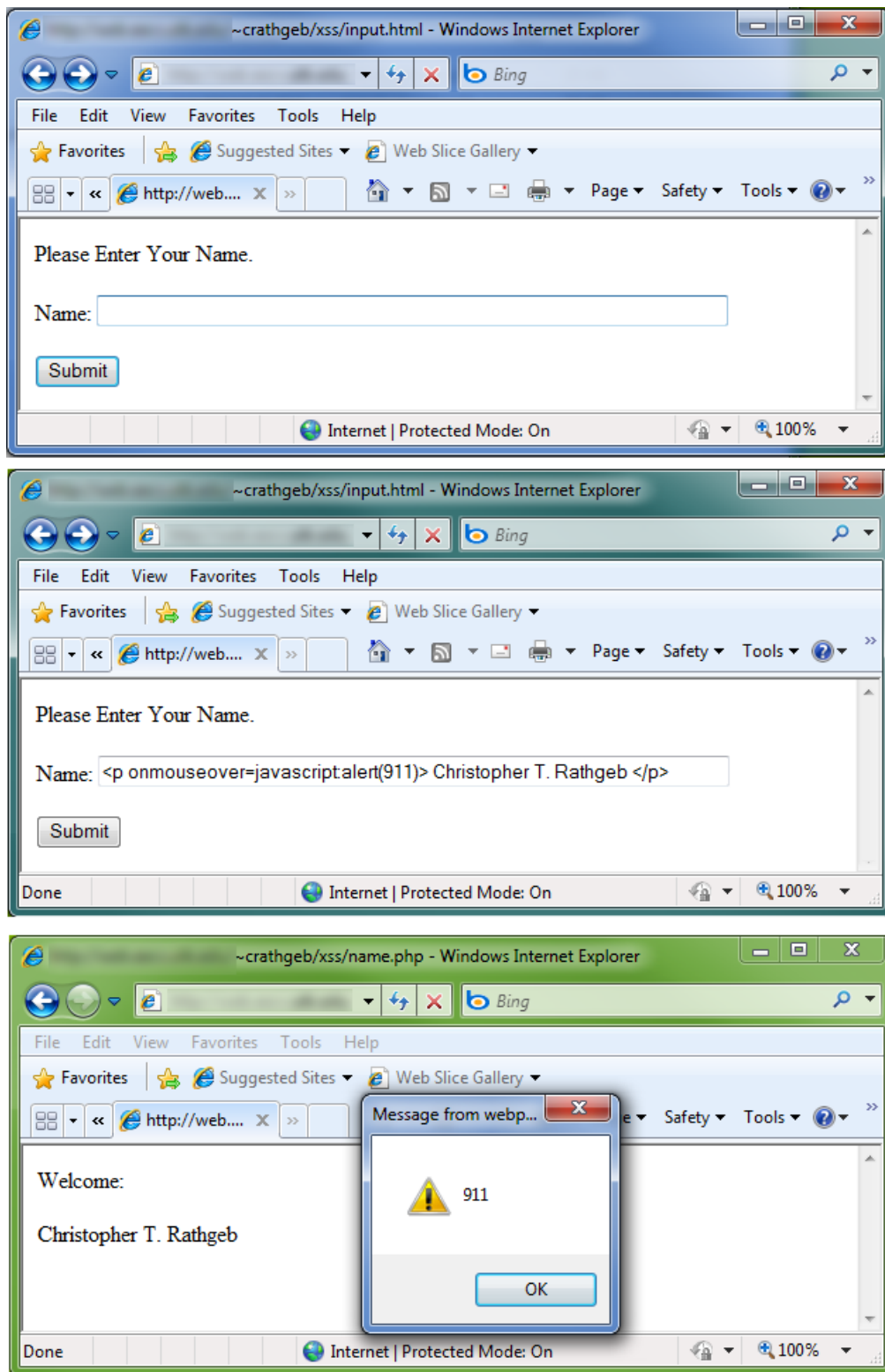


Figure 3.2: Cross-site scripting injection example

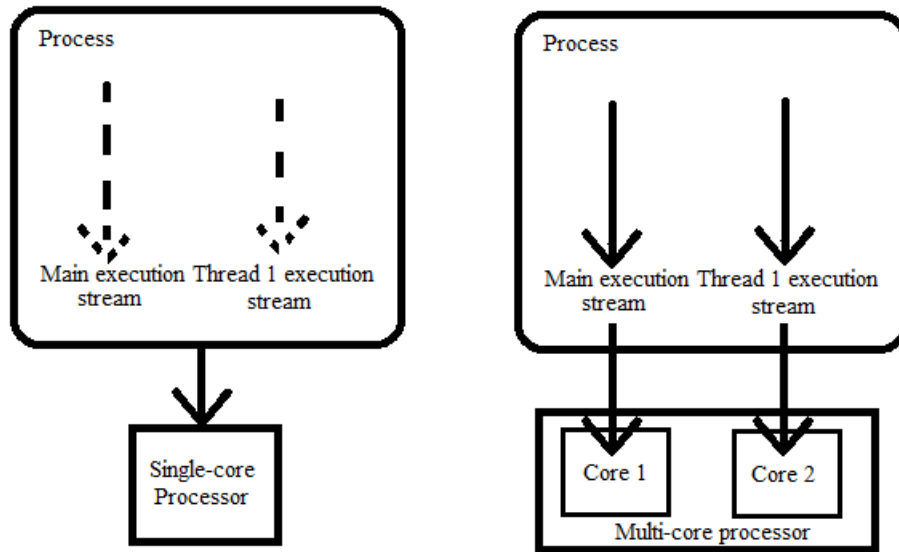


Figure 3.3: Thread level concurrency on single and multi-core processors

of execution. On the other hand, single-core processors offer pseudo-concurrency by multiplexing the processor between execution streams. This concept can be seen in Figure 3.3.

Threads are the ideal construct for DLL code injection, because after the code of a DLL is loaded into the memory of the target process, a thread can be spawned within the host process. This allows the code to execute. Later in this chapter several methods of this technique will be detailed.

Example: Injecting “Hello World!” into Internet Explorer

In the following example, helloworld.dll is injected into iexplore.exe. This results in the DLL simply displaying a message box that says ”Hello World!” The code for helloworld.dll can be seen below.

CODE: helloworld.cpp

```
bool APIENTRY DllMain(HMODULE hModule,DWORD ul_reason_for_call, LPVOID lpReserved){
    MessageBoxA(NULL,“Hello World!”,“Hello World!”,MB_OK);
    return true;
}
```

Figure 3.4 is a screenshot of the output when injected into an active instance of iexplore.exe.

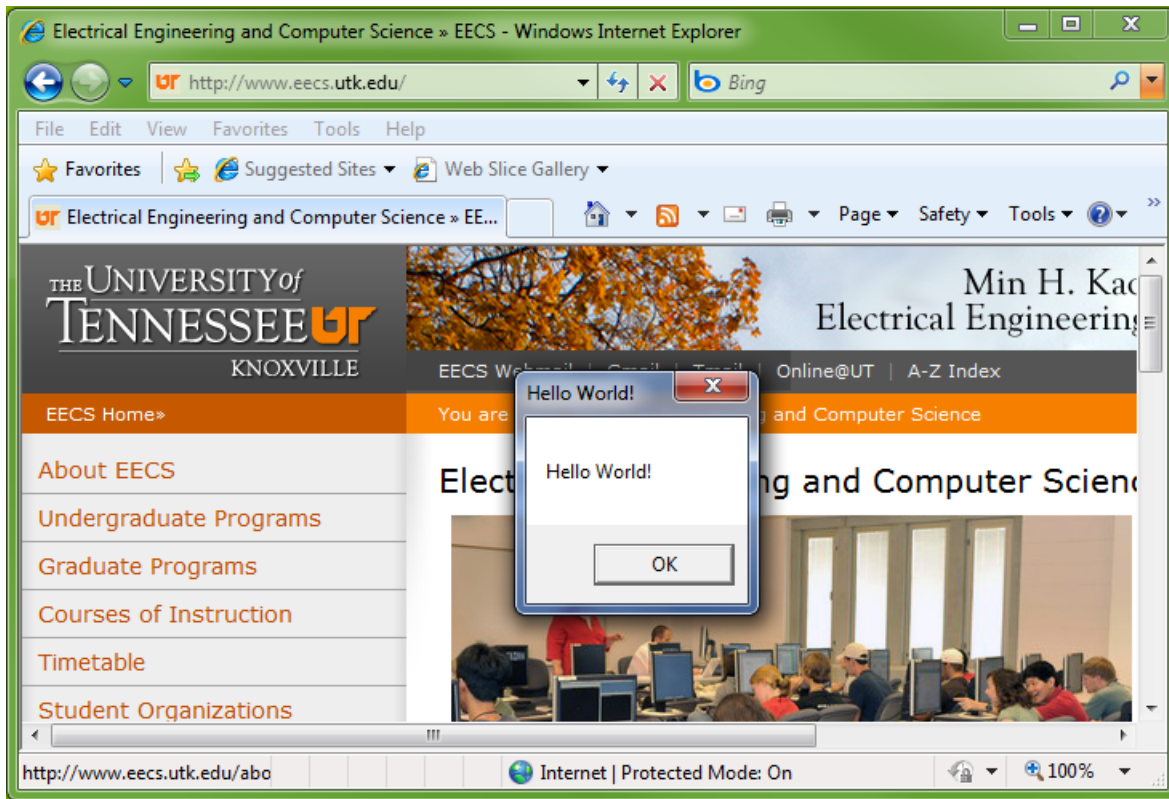


Figure 3.4: Result of helloworld.dll being injected into iexplore.exe

3.1.4 Summary of Types of Code Injection

Understanding the basics of other forms of code injection is crucial to understanding DLL injection, which is the method used in this thesis. SQL injection attacks exploit databases through user input that has not been validated. This method can be used to insert, delete, modify, or extract data from databases. XSS injection exploits user input to embed malicious scripts inside the HTML markup downloaded by the user when they view a webpage. Finally, dynamic link library injection targets the processes of applications running on a computer system. It forces the execution of precompiled code within the virtual memory space of the host process. The next section will discuss several of the techniques used for implementing DLL injection.

3.2 DLL Injection Techniques

There are many methods of DLL injection such as Windows APIs, Windows hooks, code caves, and reflective injection [8] [7]. Of these methods, the two most common techniques are Windows APIs and Windows Hooks. This section will provide a general overview of these two methods. More detail and source code implementations of these methods can readily be found on the internet. As stated in the introduction these methods assume some form of execution is already on the system. In the case of malware the execution is usually accomplished through email attachments, Trojan horses, or some sort of vulnerability such as a buffer overflow. In the defensive mechanisms presented in this work, execution can be manually initiated.

3.2.1 Windows APIs

DLL injection can be achieved through the correct combination of Microsoft Windows APIs. The WinSpy project on www.codeproject.com provides the following sequence of API calls to inject a DLL into a target process [25]. This information is common knowledge and used by many “script kiddies” to perform DLL code injection.

1. Retrieve a `HANDLE` to the remote process (`OpenProcess`) [25].
2. Allocate memory for the DLL name in the remote process (`VirtualAllocEx`) [25].

3. Write the DLL name, including full path, to the allocated memory (`WriteProcessMemory`) [25].
4. Map the DLL to the remote process via `CreateRemoteThread` and `LoadLibrary` [25].
5. Wait until the remote thread terminates (`WaitForSingleObject`); this is until the call to `LoadLibrary` returns. Put another way, the thread will terminate as soon as `DllMain` (called with reason `DLL_PROCESS_ATTACH`) returns [25].
6. Retrieve the exit code of the remote thread (`GetExitCodeThread`). Note that this is the value returned by `LoadLibrary`, thus the base address (`HMODULE`) of our mapped DLL [25].
7. Free the memory allocated in Step #2 (`VirtualFreeEx`) [25].
8. Unload the DLL from the remote process via `CreateRemoteThread` and `FreeLibrary`. Pass the `HMODULE` handle retrieved in Step #6 to `FreeLibrary` (via `lpParameter` in `CreateRemoteThread`) [25].
9. Wait until the thread terminates (`WaitForSingleObject`) [25].

3.2.2 Windows Hooks

The next technique commonly used to perform DLL injection is the Windows hooks method. This method works through the `SetWindowsHookEx()` function. This function hooks Windows messages for local and remote processes. A hook is another word for a detour, which will be described in detail in Chapter 4. Hooks are a way of triggering code segments to be executed whenever a function or message is called. The injected DLL has a filter function that is added to a linked list of existing hooks. When the Windows message is triggered, this list is traversed and the provided functions are executed. Therefore, the DLL injection is handled implicitly through the hooks [16].

3.2.3 Summary of DLL Injection Techniques

The Windows APIs and Windows hooks techniques are two of the most commonly used DLL injection methods. This section examined the general procedure taken by these two methods. The

next section will discuss techniques used to detect DLL injection.

3.3 Detecting DLL Injection

Detecting a DLL injection depends on two factors, the method of code injection and the measures the programmer went to in order to avoid detection.

The first factor depends on the method through which the code has been injected. Certain methods are more detectable than others. For example, the Windows API injection method is easily detected in several ways. It explicitly calls `LoadLibrary()`. A functional hook placed on this API call could be used to catch this method. A second method uses the Windows Process API, which provides the functionality to enumerate loaded modules for a given process. The enumerated list can be scanned for injected DLLs [42]. For each of these methods of detection, a countermeasure exists to avoid detection.

For this reason, the second factor of detecting DLL injection depends in part on the programmer's determination to avoid detection. In the example of the enumerated module list, the injected library can remove itself from the enumeration lists. Many DLL injection techniques are detected by file scanners and antivirus software because of their hard disk activity. These detection methods can be evaded by loading dynamic-link libraries directly from memory without disk interaction. Reflective DLL injection is an example of a technique that accesses the disk making it harder to detect [8].

The research presented in this thesis does not implement any of the anti-detection mechanisms described in this section. However, stealthy execution is possible if it is required.

3.4 Existing Applications of DLL Injection

DLL injection, like other forms of code injection, generally has a negative connotation. These negative connotations stem from its association with malicious software. DLL injection allows the user to force the execution of their code inside another process with that process's access privileges.

Once inside the process, it shares the state, memory address space, and resources with the host process. Many viruses and worms have used this technique to execute code without detection. While there are many ways this technique can be used improperly, there are several legitimate usages as well.

3.4.1 Malware

There are many features of DLL injection that make it attractive to malware such as viruses and worms. First, this form of code injection is relatively easy to use. “Script kiddies” can perform an Internet search and obtain working code for different methods of injection. Second, as described in the previous section, the ability to avoid detection methods allow for stealthy execution. Stealth is one of the most attractive features of DLL injection to malware. In general, the longer malware lives undetected the more damage it does. Third, while some methods are more publicly known than others, there are many different techniques for process injection. This is beneficial to malware because varying injection methods can help avoid detection by code signature. Fourth, the nature of threads grants malware access to the memory of the host process. This access can be used to steal information or modify the logical flow of the application. Fifth, when malware is successfully operational within a process, it assumes the access privileges of that process. Should a piece of malware find its way into a system or administrator-owned process, it can effectively perform operations at that access level. Finally, malware can mask its functionality and hide from network and host-based intrusion detection systems by cleverly attaching to the right processes. For example, some forms of malware attempt to connect to a central server. If malware attempts to connect through notepad, it could raise suspicion. However, if it uses a web browser to connect, it is more likely that an intrusion detection systems would perceive this as normal activity [6].

Example: Thread Bomb

The thread bomb attack is a relatively harmless piece of malware created for the purpose of this example. This attack injects into a process and spawns off the maximum number of threads allowed. Each created thread enters an infinite loop, which quickly exhausts the computer’s resources. The

following is the code that does this.

CODE: threadbomb.cpp

```
#include <windows.h>
#include <stdlib.h>

#define MAX_THREADS 8192
#define HALF_SECOND 500
void ThreadBomb();

bool APIENTRY DllMain(HMODULE hModule,DWORD ul_reason_for_call, LPVOID lpReserved) {
    for(int i=0;i<MAX_THREADS;i++)
        CreateThread(0,0,(LPTHREAD_START_ROUTINE)ThreadBomb,0,0,NULL);
    while(1) {}
}

void ThreadBomb() {
    while (1) {
        Sleep(HALF_SECOND);
    }
    ExitThread(0);
}
```

The results can be seen in Figure 3.5. The thread bomb does no real harm to the system but illustrates how easy it is to create malware with process injection. This type of attack can quickly disable a system and require a reboot. If code were added to this attack to automatically run upon reboot the thread bomb could become much more malicious.

Example: The Storm Worm

The Storm computer worm surfaced in early 2007 and targeted computers running variants of Microsoft Windows. It creates a peer-to-peer distributed botnet capable of launching DDoS, spam, and phishing attacks. The fact that the Storm worm is peer-to-peer and not centralized makes it difficult to destroy [49].

In March of 2008, Offensive Computing, LLC. performed a malware analysis on a variant of the Storm worm. The particular strand of the worm loads a driver into the kernel which injects the Storm worm into services.exe. The worm begins by enumerating the process list to determine the id of services.exe. Next, the payload of the worm is decrypted with a simple XOR cipher. After that

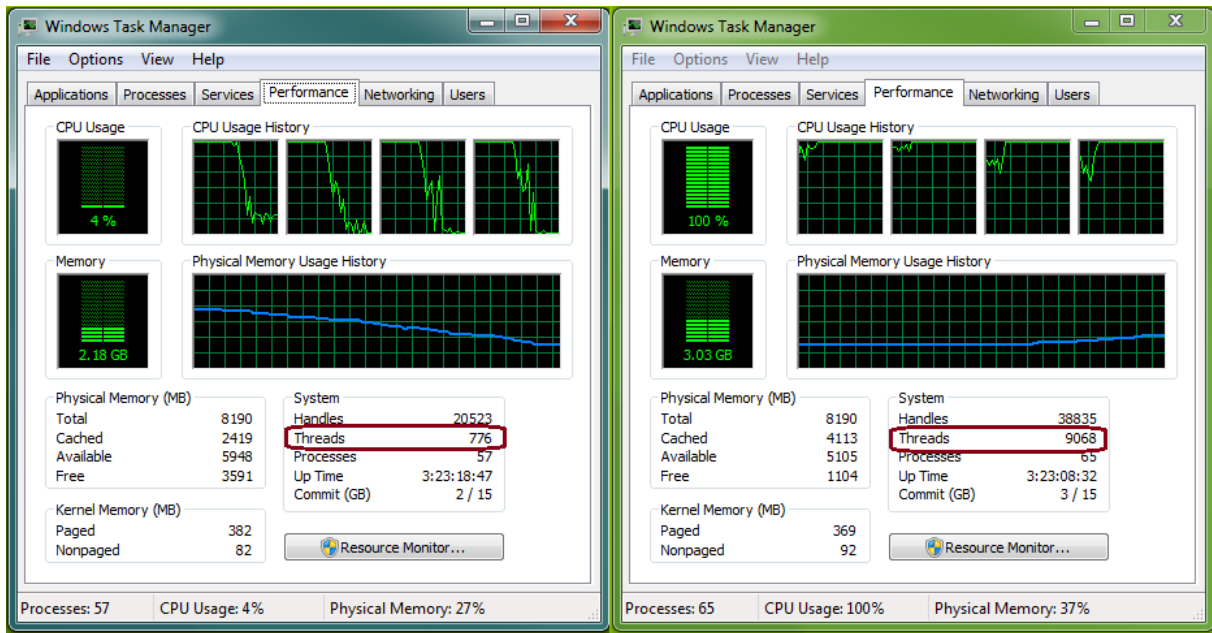


Figure 3.5: Effect of a thread bomb in action

the worm attaches to the process space and allocates memory through the `KeAttachProcess` and `ZwOpenProcess` functions. The code is then inserted into the running process [38]. In the case of the Storm worm, the inserted code is in executable form not DLL form; however, the end result is the same.

3.4.2 Functionality Additions

Thus far code injection has been discussed in terms of malware. However, not all code injection is malicious. DLL injection can be used to add functionality to software systems in which source code is not available. Such systems may include legacy code, lost code, or proprietary applications. For example, Company X has a custom software system they use to do statistical analysis on client data. One day they decide they want the output of their statistical analysis to be web accessible, but they do not have the source code to the software system anymore. DLL injection allows them to insert the necessary functionality. If they were to inject a DLL into the process that harvested the output of the statistical model and put it into an SQL database they could write a web application that displays the data through SQL queries.

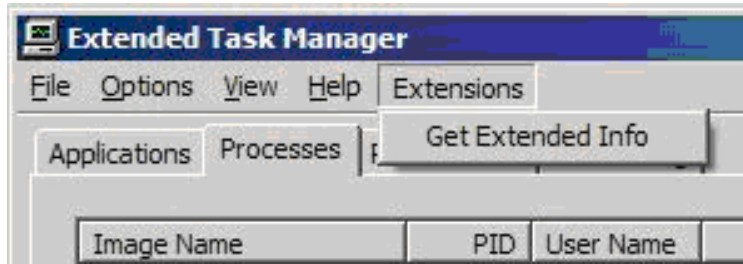


Figure 3.6: Extended functionality of Task Manager [40]

Example: Extending Task Manager

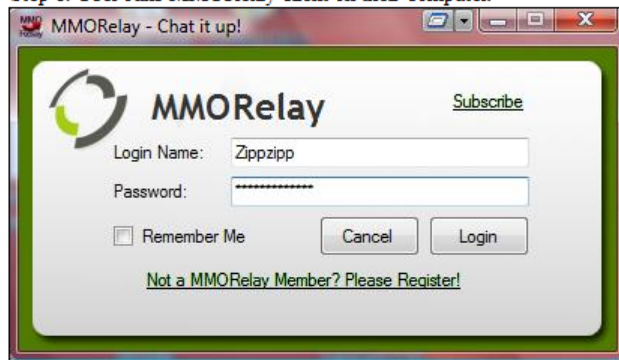
In the TaskEx project on www.codeproject.com, DLL injection is used to extend the functionality of the Windows Task Manager. First, an application sits in the background waiting for instances of task manager to popup. When an instance is discovered, it injects a DLL that installs an extra menu named “Extensions” to the application’s main menu. From here additional information can be obtained about particular processes [40]. A picture of the extended functionality addition can be seen in Figure 3.6.

Example: MMORelay

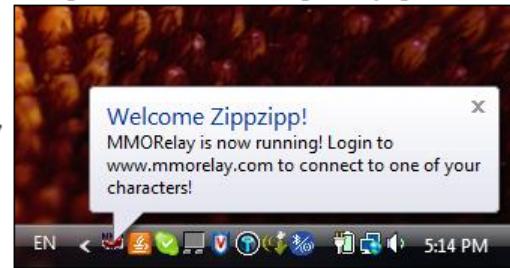
MMORelay is a software system that I developed in 2008 to add functionality to massive multiplayer online role playing games. MMORelay is an application the user runs that connects to the video game and relays the chat functionality of to a web interface. This web interface can be accessed and interacted with from any web browser around the world. The end result is the added ability of a user to chat with their virtual video game friends without being at their computer.

The system adds the relay functionality to the game through DLL injection. The injected code detects output to the chat window of the game and sends it off to the web server. The web server queues the messages in a database. When the user connects to the website he or she pulls the messages from the database and can actively respond to them remotely. The response is sent back to the person’s computer and ultimately put into the game.

Step 1: User runs MMORelay client on their computer.



Step 2: MMORelay establishes a connection from the video game to webserver and begins relaying chat.



Step 3: User logs into website to chat with thier video game friends.

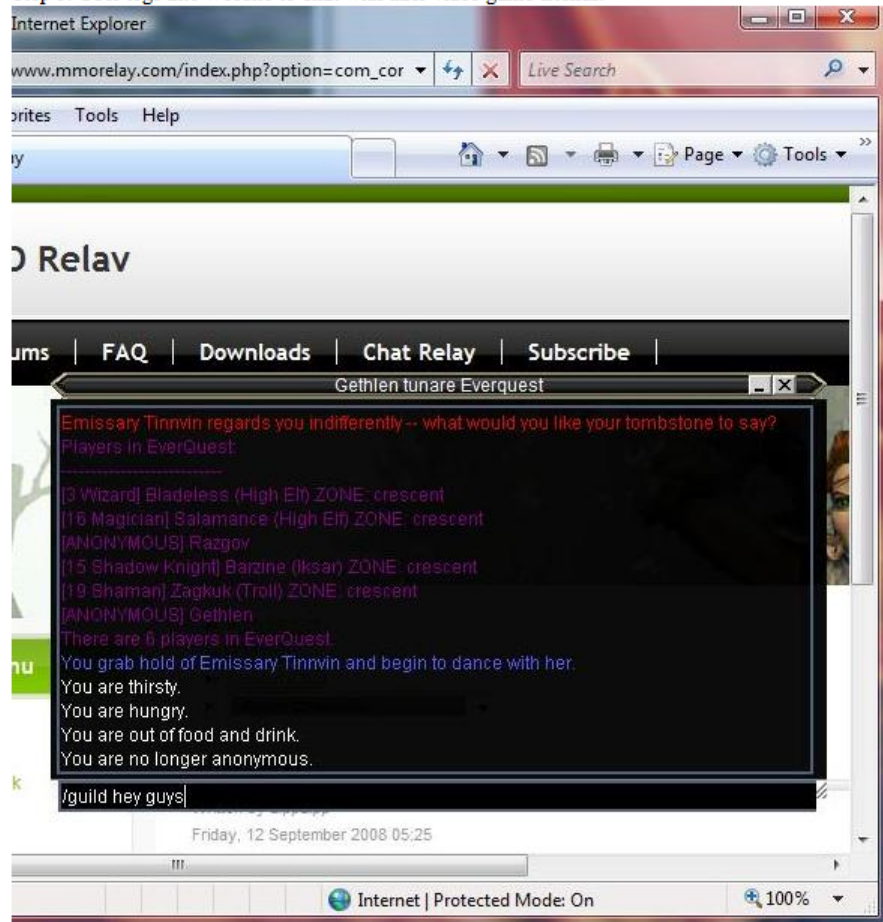


Figure 3.7: MMORelay's chat functionality addition through DLL injection

3.4.3 Summary of Existing Applications of DLL Injection

Dynamic-link library injection is a powerful method that can be used by software with both malicious and legitimate intent. Malware uses this method because it provides a simple, robust, and stealthy way of executing malicious code within the address space of another process. This allows the malware to assume the privileges of the host process and hide from intrusion detection systems. On the other hand, DLL injection can also be used legitimately to extend the functionality of software that has no source code available. MMORelay is an example of a software system that uses DLL injection to extend the functionality of a video game. It provides web-based interoperability to the chat features of the game. Another example of a legitimate use of DLL injection is the functional extensions added to task manager, which can be seen in Figure 3.6 [40]. The next section discusses dynamic application level sensors as a new application of DLL injection application.

3.5 Dynamic Application Level Sensors Through DLL Injection

Dynamic application level sensors can be implemented through DLL injection. The same legitimate methods used to add functionality to applications can be used to add dynamic sensors to applications. These sensors have a unique advantage over host level security domain defenses. They execute within a thread of the application, which gives them access to the application's state, virtual memory, and resources. This information can be extracted by the injected DLL and used to help secure the system against external and internal threats.

3.5.1 External Threats

External threats are constantly probing computer systems looking for any sign of weakness. Dynamic application level sensors are uniquely positioned to defend against these threats. For example, a sensor could be setup to collect data on the normal executional flow of a program. This data could then be used to build a profile for the program logic flow. Should the program deviate from this executional flow, such as a buffer overflow exploit used to install malicious shell code, the sensor data returned would reflect this abnormality and the system performing analysis would see

the deviation. Another example is a sensor that detects potential buffer overflow, memory leaks, or arithmetic overflow and dynamically patches them. For example, a sensor reports to the analysis station that an unsafe `strcpy()` is being executed. The response mechanism then deploys a functional detour that replaces the unsafe function with a safe version.

3.5.2 Internal Threats

As discussed in Chapter 2, the insider threat problem costs businesses tens of millions of dollars each year [22]. Dynamic application level sensors could help protect against malicious insiders. The state information of applications and usage patterns of insiders can be monitored at run time, through DLL injected sensors, to profile normal usage. Should a malicious insider begin acting outside these normal patterns, the application level anomaly detection system built from dynamic application level sensors would detect it. The example sensor presented in Chapters 5 is targeted at malicious insiders.

3.5.3 Summary of Dynamic Application Level Sensors Through DLL Injection

DLL injection provides the required constructs to implement dynamic application level sensors. These sensors can be used to probe an application in ways that were previously not possible. Application level security domain defenses can be built to protect against threats to the network and computer system, both external and internal. Chapters 5 and 6 will explore example implementations of dynamic application level sensors.

3.6 Conclusions

There are several types of code injection that can be used in malicious ways, such as SQL, XSS, and DLL injection. However, DLL injection can also be used legitimately to add dynamic sensors. The idea behind DLL injection is to force a thread of execution inside a targeted host process. This thread has access to the state, virtual memory, and resources of its parent process and therefore has a unique opportunity to collect fine grain data and process it for defensive security applications.

The next chapter will discuss the sequence of steps taken to develop one of these custom application level sensors.

Chapter 4

Developing Dynamic Sensors

Dynamic application level sensors extract critical information from processes at run time. Developing these sensors is a multi-stage process. The first stage is to determine the extractable data. This primarily depends on two factors, the security threat and target application. The second stage is to reverse engineer the target application. This stage is a challenging feedback loop of static and dynamic binary analysis. The goal is to extract memory locations for critical data elements and functions that can be logically detoured. The third stage uses the extracted memory locations discovered in the reverse engineering stage to automate the data extraction in the form of a DLL that is injected into the application. Finally, the fourth stage is to analyze the data and respond to any threats. This chapter will describe each step of this multi-stage process; however, it will concentrate primarily on the first three stages. This is due to the fact that the focus of this thesis is not on the analysis and response but rather the sensor elements.

4.1 Determine the Extractable Data

Identifying what type of data is collectable is the first stage of developing dynamic application level sensors. This directs the reverse engineering stage. In order to effectively begin reverse engineering, the programmer needs to know what valuable information to seek. Determining the type of data to collect primarily depends on two factors: the security threat and targeted application.

4.1.1 The Security Threat

The first factor that influences what type of data to collect is the security threat. For example, the type of data needed to defend against insider threat differs from the type of data required to defend against malware.

An insider threat detection sensor could collect information pertaining to a user's usage tendencies for a particular application, such as all possible user input for an application. The input could be then used to model how the user typically interacts with that particular application. In the analysis stage an anomaly detection algorithm could profile the users's normal behavior within the application, and then respond to any behavior that is abnormal. Chapter 5 creates the sensor portion of this defense on the application `putty.exe`, which is a client SSH program.

A malware detection sensor could extract runtime information on the binary execution of software functions that are known to be malware free. Statistical analysis could provide a threshold for expected runtimes of these functions. Should a form of malware attempt to nullify or inject malicious code into one of these code segment, the sensor would report timings outside of acceptable mathematic standard deviations. Additionally, this type of defense would be able to protect against polymorphic code, because polymorphism tends to increase the size of the malware code. This is done in an attempt to obfuscate the true functionality from signature matching methods and reverse engineering. The increased size of the malware code would result in longer timing analysis making it appear even more abnormal. This sensor has been created and is detailed in Chapter 6.

The first factor to developing dynamic sensors that can potentially detect these security threats is determining what to collect. Deciding which application the sensor will collect data from is the next factor that needs to be considered.

4.1.2 The Application

Ascertaining which applications to develop sensors for is the next factor in determining what data to collect. Selecting the application has several dependencies, such as how specific the sensor is to the application, what applications are available to run, and the language in which the application was originally coded.

The first dependency is how specific the sensor is to the application. If a sensor is highly specific to a particular application then the application must fit the sensor's requirements. For example, a sensor could be developed to collect information on the keywords or phrases entered into an internet search. Text analysis could then be run on the keywords for phrases such as "how to build a bomb" or "how to plant a computer virus." The requirements for such an application are that the application must be able to perform Internet searches. The most likely choice of application would be a web browser, however any program that has a `iframe` embedded into it also meets this specific constraint. Some programs embed dynamic advertisements inside `iframes` of applications.

Generic sensors can be applied to any application that fits a set of constraints. For example, a constraint may be that the application must render graphics to the screen using Microsoft's DirectX. This could be the case for games or other graphic intensive applications. A sensor that could have this particular constraint would be one that collects data about what is being drawn to the screen through the render function of DirectX.

The second dependency of the choice of application to collect data from is the set of available applications, particularly the subset of applications that run regularly. This subset may change depending on the users of a system. For example, an accountant is more likely to use a program like Quicken than a computer programmer. It may be of particular interest to the employer of this accountant to install application level sensors that monitor the accountant's usage of this program. For this reason, the company could develop a sensor that canonically captures and logs all possible user input to Quicken. The sensor not only captures the inputs but also the biometric timing data of the inputs. An anomaly detection algorithm, then processes the extracted data and builds

a standard usage baseline for the accountant's interactions with Quicken. This baseline includes what operations the accountant normally executes within the program, the standard transitions from one action to the next, and the biometric timings related to those transitions. If the accountant leaves his desk to go to lunch and forgets to lock his door, the company does not need to worry about disgruntled coworkers attempting to modify the books to steal money. The application level sensor captures, logs, and processes all of the interactions to Quicken. A properly trained anomaly detection system would be able to tell that the accountant was not the individual using Quicken during the lunch break. There would also be a log file showing the sequence of events inside the program.

The final dependency on the choice of application for which to develop sensors is the language that the application was originally coded. As stated in the introduction, the work presented in this thesis makes several assumptions. These assumptions include that the applications these sensors are dynamically added to run on Microsoft Windows and are programmed in a language that can be disassembled into x86 assembly, such as C or C++.

4.1.3 Summary of Determining the Extractable Data

Determining the type of data that is collectable helps pinpoint what to look for in the reverse engineering stage. There are two main factors that help decide what type of data to collect. The first factor is the type of security threat. The data collected for insider threat detection is different than the data collected for malware detection. The second factor is choosing which application to attempt to collect data from. This decision is dependant on how specific the sensor is to the application, the subset of applications that are run on the system, and the language in which the application was coded. Once the application and type of data are determined, the next step is to reverse engineer the binary to extract the critical data and functions. This stage will be discussed in the next section.

4.2 Reverse Engineer the Application

Reverse engineering is the second stage of the multistage process. It is the sequence of steps taken to break down an application to machine readable code for static and dynamic analysis. The goal is to extract critical memory locations for function detours and data. Much like code injection, the term “reverse engineering” has acquired a negative connotation. This is due to the existing motives of many individuals who reverse engineer software. Such motives include, finding vulnerabilities in code in order to developing malware, analyzing existing malware in order to build a defense, and commercially gaining a competitive advantage by learning the “trade secrets” of a competitor.

However, reverse engineering principles can be used constructively in order to dynamically add defensive sensors to applications. Much like analyzing existing malware, the intention is beneficial to the computer owner. As discussed in earlier chapters, the types of software that require reverse engineering are those for which source code is not available, such as proprietary, legacy, or lost code. In cases where code is available, such as open source or in-house programs, reverse engineering is not required as the sensors can be compiled directly into the application.

In the case of legacy or lost code, reverse engineering is a necessary measure. However, in the case of proprietary software, cooperative reverse engineering should not be necessary. Companies should either provide the source code to their products or guarantee the security of their code. At the very least they should provide the proper hooks for adding security dynamically after the software has been deployed. Unfortunately, due to the nature of business and lack of policy to govern proprietary software security requirements, proprietary software must be reversed as well.

The goal of the reverse engineering stage is to extract memory locations for critical data elements and functions that can be hooked. This section will focus on the reverse engineering process, and detail the tools and methods used to extract the required data for the security sensor.

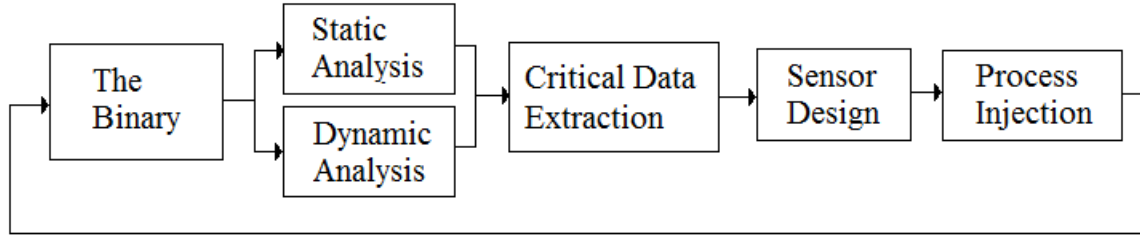


Figure 4.1: Reverse engineering feedback loop

4.2.1 The Reverse Engineering Process

The reverse engineering process is a challenging feedback loop of static and dynamic binary analysis, critical data extraction, sensor design, and process injection. Static analysis is the process of disassembling the application into machine code and studying it while it is in a not running state. On the other hand, dynamic analysis studies the application while it is in a running state. Breakpoints are inserted and the dynamic state of the registers, the stack, and programmatic flow are studied. From this static and dynamic analysis data is extracted and a sensor designed. The reason the reverse engineering process is a feedback system is that often finding the targeted information is not trivial. Intermediate step sensors are developed to provide more information and gain further insight to the functionality of the program. These intermediate step sensors are injected into the application to assist in finding the targeted information in future dynamic analysis steps. This reverse engineering feedback loop ends when the targeted information is discovered. This feedback loop can be seen in Figure 4.1.

4.2.2 Reverse Engineering Tools

As with any job, having the right set of tools can make the task easier. For reverse engineering, the proper set of tools should consist of a disassembler, a hex editor, a jump calculator, and a compiler. The following is the set of tools used to perform the work presented in this thesis.

1. IDA Pro is an interactive disassembler and debugger. It supports over 50 processor families and allows the user to analyze the assembly code of an application both statically and dy-

namically. When used dynamically, it runs the application in debug mode, which allows the user to add breakpoints and step through the program's assembly code line-by-line. At each step, dynamic information, such as the state of the registers and stack, can be analyzed [17].

2. Winhack 2.0 is a traditional memory editor. It displays the virtual memory for each process and allows modification at run time.
3. Cheat Engine is a more sophisticated memory editor. It has additional functionality to iteratively scan memory for specific values, freeze memory locations from value changes, snoop code that access specific memory locations, and find references to pointers. The full functionality of Cheat Engine is not limited to the described capabilities and can be found online [5].
4. JumpCalc is a program used to calculate the hexadecimal representation of a modified branch instruction for the x86 instruction set architecture. This is useful when changing the conditions of a branch. For example, turning a conditional jump, such as `JNE`, to an unconditional jump, such as `JMP`, changes the hexadecimal representation. This will be discussed in further detail in the next subsection.
5. Visual Studio 2008 is a compiler. It is used to develop the intermediate step sensors and ultimately the final sensor.

Screenshots of each of these tools in action can be found in Appendix A. The next section will focus on several of the methods used while reverse engineering an application.

4.2.3 Reverse Engineering Methods

The dynamic binary analysis portion of the reverse engineering feedback loop is where knowledge of the code layout meets line-by-line execution. It allows the reverse engineer to study the state of the application at a given instance of its execution. However, the dynamic analysis section is not limited to running in debug mode. The combination of other methods can prove to be quite helpful. Function detouring, logic path alterations, iterative memory scanning, and the outputs

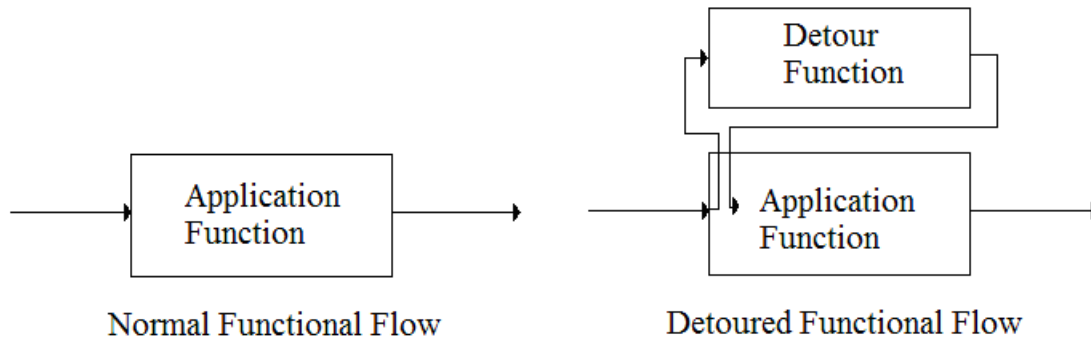


Figure 4.2: Pictorial example of detours

from previous intermediate step sensors can all be used to help extract the critical data required of the reverse engineering.

Function detouring, also known as a wedge or a hook, is the first method that can help extract critical data. It is the process of “intercepting arbitrary Win32 functions on x86 machines [15].” Detours work by replacing the first few instructions of the targeted function with unconditional jumps to a detour function. The original function is saved in a “trampoline.” After the dynamically added detour function is finished, the trampoline is used to resume execution as normal [15]. Microsoft research presented this concept and library in the Proceedings of the 3rd USENIX Windows NT Symposium in 1999. They claimed that the typical reasons to intercept functions is to add functionality, modify returned results or insert instrumentation for debugging or performance profiling [15]. This work proposes a new reason, which is to use function detours to insert application level cybersecurity sensors. Figure 4.2 shows a pictorial representation of function detours.

Logic path alterations can be used to help extract critical data. They are subtle modifications to the application’s code at runtime that force a branch one direction or another. Logic path alterations are ideal for determining the functionality of assembly functions. For example, a reverse engineer observes the behavior of a function and decides to replace a conditional jump, such as `JNZ`, with an unconditional jump, such as `JMP`. The reverse engineer does this in order to observe the new behavior of the function within the application. This task can be accomplished with the JumpCalc

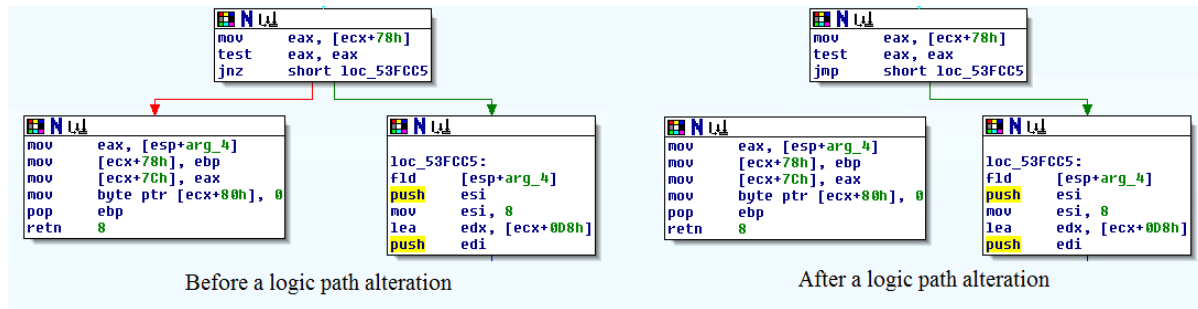


Figure 4.3: Logic path alteration example

tool. This tool calculates the hexadecimal representation of the new assembly instruction `JMP`. Then the reverse engineer uses a memory editor such as WinHack 2.0, to change the memory location of the `JNZ` instruction to the calculated hexadecimal representation of the new instruction. After this logical path alteration, the branch is always taken. This can be seen in Figure 4.3.

Iterative memory scanning is the process of scanning memory for some criteria. After the results are returned, the search is refined and performed again on the returned result set from the previous search. This process is continued until the location of the targeted data is pinpointed. The tool Cheat Engine provides this functionality. This is particularly useful for finding the memory location of data that can be manipulated in the original program. For example, as the user punches text into Notepad++, the text is stored into memory. Iterative memory scanning could be used to pinpoint the start of this storage. Figure 4.4 shows this process.

4.2.4 Summary of Reverse Engineering the Application

The process of reverse engineering is a feedback loop. Multiple passes through the loop allow for different static and dynamic analysis. Methods such as function detouring, logic path alterations, and iterative memory scanning can be used to help understand the full functionality of an application. This understanding allows for the extraction of the targeted types of critical data. The goal of reverse engineering is to find the memory address of important functions and data constructs within the application. The next step is to develop the sensor into a DLL that can be injected. The DLL applies detours to the important functions to act as a checkpoint and log data that passes

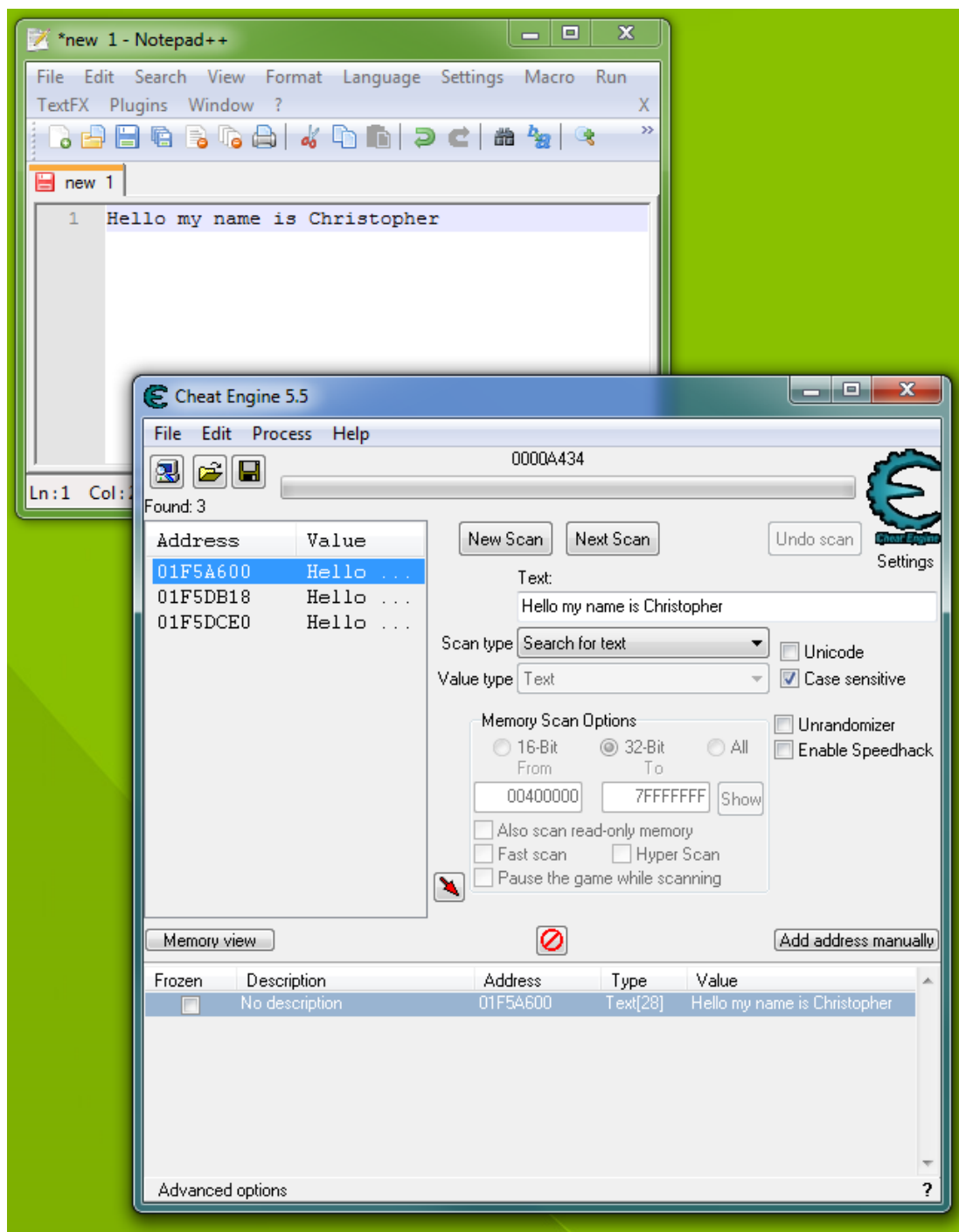


Figure 4.4: Iterative memory scanning with Cheat Engine

through the function. The sensor then references the memory addresses of important data and log the contents for further analysis. The next section will discuss the automation of the data extraction.

4.3 Automate the Data Extraction

The reverse engineering stage provided the required memory addresses to apply function hooks and extract the targeted sensor data. The final stage of the multistage sensor design process focuses on automating the data extraction from the application. This is accomplished by developing a DLL that can be injected into the application's process at run time. This section will discuss coding the actual sensor, the need for a development and deployment framework, and how to automatically update the sensor when a software update is released.

4.3.1 Coding the Sensor

Sensors are dynamic-link libraries coded in unmanaged C++. Unmanaged C++ allows direct access to pointers and in-lined assembly. When a sensor is injected into an application, the DLL entry point is called. The following code segment shows the entry point for a DLL:

Code: DLL entry point

```
bool APIENTRY DllMain( HMODULE hModule,DWORD ul_reason_for_call,LPVOID lpReserved){
    switch (ul_reason_for_call){
        case DLL_PROCESS_ATTACH:
            // create a thread of execution for the function ThreadFunc
            CreateThread(0,0,(LPTHREAD_START_ROUTINE)ThreadFunc,0,0, 0);
            break;
        case DLL_THREAD_ATTACH:
            break;
        case DLL_THREAD_DETACH:
            break;
        case DLL_PROCESS_DETACH:
            threadloop = false;
            break;
    }
    return true;
}
```

The `ThreadFunc()` begins by calling an `InitializeSensor()` function. This function can be used

to add any detours to the application. `ThreadFunc()` then enters a while loop that waits for a signal to terminate. The while loop keeps the thread alive and periodically calls a pulse function that can be used to perform operations that need to be called regularly. The frequency of this pulse can be changed and is set to 100ms in the next code segment. Finally, the thread function calls `ShutdownSensor()`. This is the point that detours are removed and the sensor performs necessary garbage collection.

CODE: ThreadFunc()

```
void ThreadFunc() {
    InitializeSensor();
    while (threadloop)
    {
        PulseCall();
        Sleep(100);
    }
    ShutdownSensor();
    ExitThread(0);
}
```

One of the goals of the reverse engineering stage is to extract the memory addresses of important functions that can be detoured. The following library function can be used to add a detour:

CODE: The addition of a detour

```
DetourFunctionWithEmptyTrampoline((PBYTE)cp_trampoline,(PBYTE)0x413424,(PBYTE)cp_detour);
```

The first argument to this function is the address of the trampoline. This trampoline holds the assembly code that was overwritten to add an unconditional jump to the detour function. The second argument to this function is the extracted memory address that is being detoured, and the third argument is the address to the detour function. Function detouring is useful for extracting the arguments passed to a function.

The second goal of the reverse engineering stage is to extract the memory addresses of critical data. Accessing this critical data is fairly simple in code. In order to find this information a programmer declares a pointer for the data type and casts the memory address into the pointer. Next, they reference the variable to obtain the data stored at that location.

Finally, it is important to code the logic that directs the data pulled from the sensor to the analysis center. Where the data is sent after it has been gathered depends on several factors. The first factor is whether or not processing happens in real time. If not, the data can be logged to a file. If the data is processed in real time, then the second factor of where to send the data is to decide where the processing will take place. If the analysis happens inside the dynamic-link library, then the logic for the analysis needs to be coded in as well. If the analysis happens in a separate application on the host machine then some form of interprocess communication needs to be added to the DLL. If the analysis happens on another machine, either a centrally located server or distributed analysis computing cluster, then some form of network communication, such as sockets, needs to be added to the DLL.

4.3.2 Signature Matching for Easy Update

When a vulnerability patch, feature enhancement, or any other software update is released, the changed binary needs to be reversed again. To facilitate this process, signature matching, the same technique antivirus scanners use to seek out malicious code, can be used. In the case of the memory address of a function that is detoured, a signature of the function can be made. In the case of the memory address of critical data, the signature of a section of code that accesses that critical data can be constructed. If the proper signatures are constructed, pulling out the new memory addresses is as simple as running a program that takes the binary and signature as an input and produces the new memory addresses as an output. If the update modifies the section of code for which a signature has been developed then signature scanning will not work. Through personal experience, this is not the case the majority of the time.

4.3.3 Summary of Automating the Data Extraction

There are two main techniques for accessing the data. The first technique is to apply a detour on a critical function to see how often it is called and what data passes through. The second technique is to cast the memory address of critical data into a pointer and access the data stored at the location. Both of these techniques can be used to efficiently sniff out sensor data. After coding

the sensor, logic needs to be coded into the DLL to direct the sensed data to the analysis center. Finally when a binary is updated, signature scanning can be used to automate the extraction of the necessary memory addresses for this stage.

4.4 Analysis and Response

While the focus of this thesis is on the sensor step of the three step process described in Chapter 2, it is worth discussing the types of analysis and response that can be performed on the sensor data. The analysis and response steps of the three step process are the core of the application level cybersecurity defense. They give the extracted sensor data meaning. This also means they serve no purpose without sensor data. There are many forms of analysis, such as signature, statistical, and biological based methods. There are also many different types of responses, such as automatic patching of vulnerable codes, killing an infected process, and notifying authorities of insider threats. This section will briefly discuss some of the analysis and response methods that can be applied to the extracted sensor data.

The process of scanning for code signatures was described in Chapter 2. This method can be applied dynamically at the application level to code segments of running processes to detect specific forms of malware or shell code that may have modified an application. Again, signature based methods are effective for finding specific code segments but fail to polymorphism.

There are many biologically inspired analysis techniques that can be implemented on the data set, such as genetic algorithms, artificial neural networks, and particle swarm optimization. These methods look to nature for potential solutions. Genetic algorithms are evolutionary algorithms used for optimization and search problems. Genetic algorithms could potentially be used to search for classes of malware on a system. Artificial neural networks are a collection of nodes that simulate biological neurons that can be trained to classify, filter, and cluster data. These neural networks could potentially be used to differentiate between normal and anomalous behavior of an insider. Particle swarm optimization is an application of swarm intelligence that looks for an optimal so-

lution based a fitness function and several constraints. These constraints could be the global best of a swarm, personal best of an individual in the population, or the local best of a small cluster of elements in a swarm. Particle swarming analysis could be used to cluster insider behavior.

Another potential form of analysis is statistical analysis. Statistical analysis takes a quantitative approach on the data. These methods attempt to extract knowledge by looking to the frequencies, means, medians, and standard deviations of events. Statistical methods attempt to find correlations between data sets. To do this it may perform regression analysis and attempt to find a function that can fit the data with the least amount of error. Another approach would be to attempt to classify groupings of data using naive Bayes classifiers. Each of these methods could be implemented to understand the standard behavior of the internal workings of an application.

After analysis has been performed, there are many potential responses, including no action, killing a process, denying access to a user, and notifying authorities. Each action will depend on the nature of the threat. In the case of malware that has infected a system, it may be useful to attempt to disable the malware by killing the process. In the case of an insider, the proper response may be to notify the system administrators or managers. If analysis determines someone is trying to hack a password, the system could temporarily disable the section of code that controls the password check and lock the users ability to attempt to authenticate.

Different forms of analysis and response can be performed on the collected data. Ultimately, the form of analysis depends on the nature of the threat and the type of data collected, and the response depends on the results of the analysis. Experimentation with different forms of analysis and response will be future work.

4.5 Conclusions

This chapter has focused on the multi-stage process of developing an application level domain security sensor. The first stage is to determine what kind of data to collect. This decision depends primarily on two factors, the security threat and application. The type of data collected is different

if the security threat is an insider than if the security threat is malicious code. Choosing what data to collect also depends on the application for which sensors are designed. The chosen application should belong to the subset of programs that are on the system. Depending on the threat, it may even belong to the subset of applications that the particular user runs.

The next stage is to reverse engineer the application. The intended goal is to extract the memory addresses of functions that will be detoured and critical data. It is important to emphasize that this is a form of cooperative reverse engineering that should not be required in the case of proprietary applications. Proprietary applications should either release the source code, guarantee security, or provide third party developers the proper hooks to add security after deployment.

The third stage is to automate the data extraction. This is accomplished by coding a dynamic-link library that can be injected into an application at runtime. Several constructs exist for extracting the necessary data from the process at run time, such as functional detours and casting the memory address of a critical data element into a pointer. In the event of an application update, code can be added to search out the signatures of the memory addresses extracted in the reverse engineering stage.

Finally, this chapter discussed several of the techniques that can be used to perform analysis and active response. The discussions of these techniques are intentionally brief because the focus of this thesis is on the sensor step of the three step process. Different methods of analysis and response will be explored in future work. The next two chapters will detail the experimental results of developing sensors and deploying them.

Chapter 5

Experimental Sensor: PuttySensor.dll

This chapter will discuss the implementation and results of the dynamic application level sensor PuttySensor.dll. PuttySensor.dll is a dynamic link library that extracts user input data from the target application. The collected data is then logged to a file. The type of attack this sensor could potentially address is the insider threat. The user input data could be used to create a profile of how the user interacts with the target program. After this profile has been created, analysis could determine whether a user's interaction with the application is within their normal usage patterns. The first section will discuss the target application for the developed sensor.

5.1 Target Application

The target application for this sensor is Putty.exe. Putty.exe is an open source telnet and SSH client for Microsoft Windows and Unix. The program is used to remotely login and control computers across a network [50]. The application has many features that are controlled by a dialog window. After the user has configured the session with their required options, they can connect to the remote machine. At this point, a terminal window is opened and the configuration control window is closed. The sensor developed in this chapter extracts only the users interaction with the configuration window. Figure 5.1 shows a screenshot of the configuration window user interface of Putty.exe. The left hand side of the window has a treeview control that displays different configuration pages, and the right hand side has all of the configurations that fall into the selected page's category.

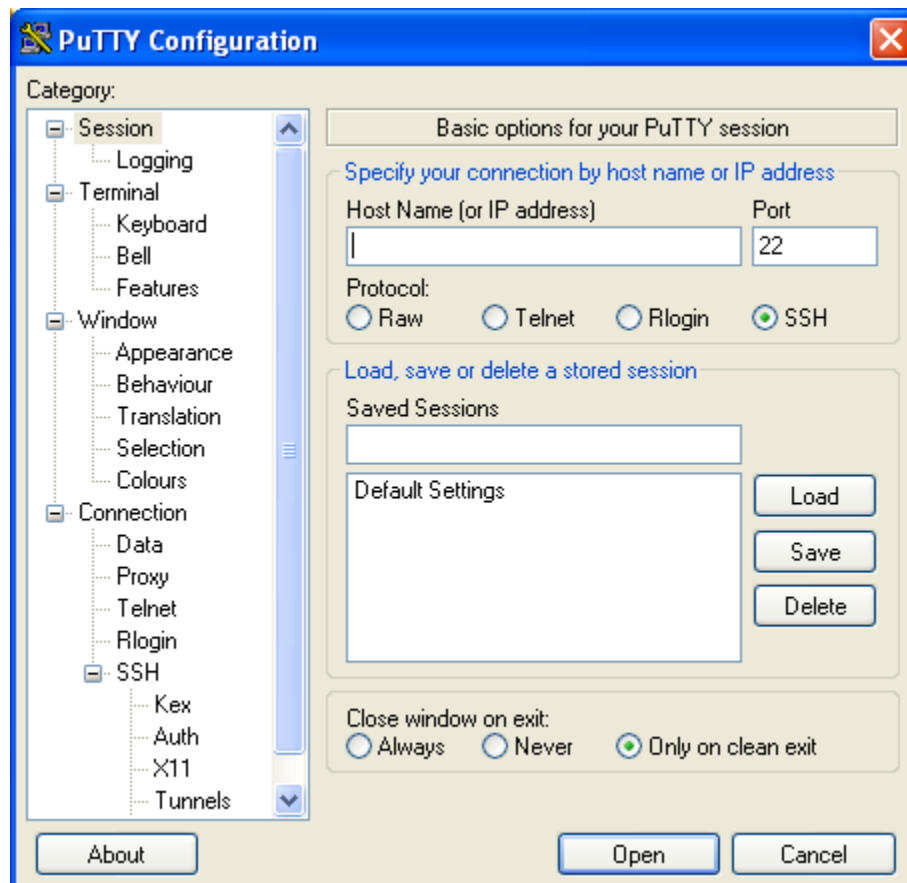


Figure 5.1: The Putty.exe configuration window

5.2 Reverse Engineering Putty.exe

The second step of the three step process described in Chapter 2 is to reverse engineer the target application. The Putty.exe application is a dialog window and makes many calls to Windows API functions in user32.dll. Of all the functions called, `GetDlgItem()` and `IsDlgButtonChecked()` are of particular interest. `GetDlgItem()` is called once for every user interface item to be displayed on the page each time the user changes between pages, and `IsDlgButtonChecked()` is called every time the user manipulates a checkbox or radio button. In order to extract the information necessary to create a dynamic application level sensor, intermediate sensors need to be developed. These sensors collect the input values passed to the functions.

The API documentation details the inputs and outputs of these two functions. Both `GetDlgItem()` and `IsDlgButtonChecked()` take two parameters as input. The first parameter is an `HWND` pointer, and the second parameter is the dialog item ID. This item ID is the targeted information for the sensor.

Detours are applied to the two library functions, and the input parameters are extracted to create the sensor data. In this example, the two target functions that will be detoured live within the user32.dll library. This system library is loaded when the application is launched. Because the target functions exist within a DLL themselves, the memory addresses for the target detour functions need to be extracted dynamically. This is done by obtaining the base pointer of the library within the virtual memory and calculating the effective address of the function to be detoured. In order to calculate the effective address, the memory offset of the target function needs to be extracted. This is accomplished by finding the base address for user32.dll in memory and the address for the `GetDlgItem()` function in memory and subtracting the difference. Figure 5.2 shows the IDA Pro screenshot of each of these memory addresses. The offset needed is calculated by subtracting 0x77650000 from 0x77690BC5. The result is 0x40BC5. The following code segment shows the dynamic effective address calculation using this offset to acquire the address of the target function.

```

usp10.dll:7749D000 ; [00040000 BYTES: 0
user32.dll:77650000 ; -----
user32.dll:77650000
user32.dll:77650000 ; [00001000 BYTES:
.....

user32.dll:77690BC5 user32_GetDlgItem proc near
user32.dll:77690BC5
user32.dll:77690BC5
user32.dll:77690BC5 ; FUNCTION CHUNK AT user32.dll
user32.dll:77690BC5 ; FUNCTION CHUNK AT user32.dll

```

Figure 5.2: Screenshot of the user32.dll base address and GetDlgItem() function address

Code: Dynamic effective address calculation of target function

```

DWORD dllhandle = (DWORD)GetModuleHandle("user32.dll");
checkDlgButtonFunction = dllhandle + 0x40bc5;
DetourFunctionWithEmptyTrampoline((PBYTE)GetDlgItem_Tramp,(PBYTE)checkDlgButtonFunction,
(PBYTE)GetDlgItem_Detour);

```

This process is repeated for `IsDlgButtonChecked()`, with a resulting offset of `0x2c927`. From this an intermediate sensor is developed and injected into the application. The detours are applied when `InitializedSensor()` is called, and the sensor's detour function, `GetDlgItem_Detour()`, intercepts the `GetDlgItem()` function each time it is called. The input parameters are then harvested and used to develop the final sensor.

The output of the final sensor is an event log of the users's interaction with the application. These events are determined based on the item IDs passed to the two API functions. The next section will discuss how the item IDs of the user interface pieces are used by intermediate sensors and eventually turned into the final event log sensor.

5.3 Implementation

PuttySensor.dll extracts the input events of users within the application Putty.exe. The sensor logs the data to a file called PuttySensor.txt. In order to capture the user input events, intermediate sensors are used to discover the item IDs of items on each page. These items IDs are used in two

ways in the final sensor.

First, `GetDlgItem()` is called for each user interface item when the page is changed. The intermediate sensor intercepts the function call with a detour and keeps a list of all of the item IDs belonging to each page. Using these lists of IDs, hashes are computed for each page. The final sensor implements the same detour and calculates a run time hash. The sensor then compares this hash to the hash created with the intermediate sensor to determine which page the user selects. When the determination is made, the event results are written to the log file.

Second, `IsDlgButtonChecked()` is called every time the user manipulates a radio button or a check box. The input ID of the first call to `IsDlgButtonChecked()` is the ID of the newly selected radio button or check box. The intermediate sensor is used to develop a list of user interface elements and their IDs for each page. Using this information, the final sensor can query the results from the intermediate sensor to determine which user interface piece is being activated. When the determination is made, the event results are written to the log file. Figure 5.3 shows a popup menu displaying the item ID for the user interface element, Telnet radio button, after it has been selected.

5.4 Results

The end result is a log file containing the input events of the application. The log file contains the time of the event and the event message. Figure 5.4 is a screenshot of this log file after several events have been captured.

The next step is to develop a tool that can take this log data and profile the user's typical behavior within this application in real time. The sensor could easily be modified to pass this data directly to the analysis tool.

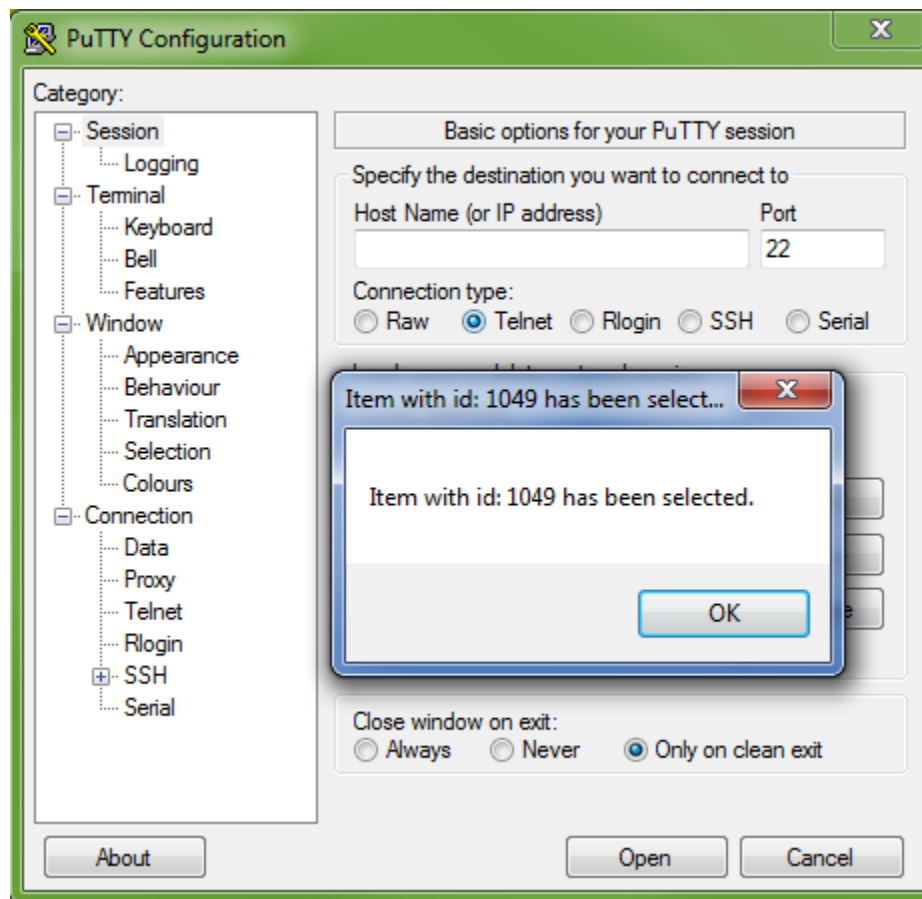


Figure 5.3: Popup box displaying the item id of the user interface element selected

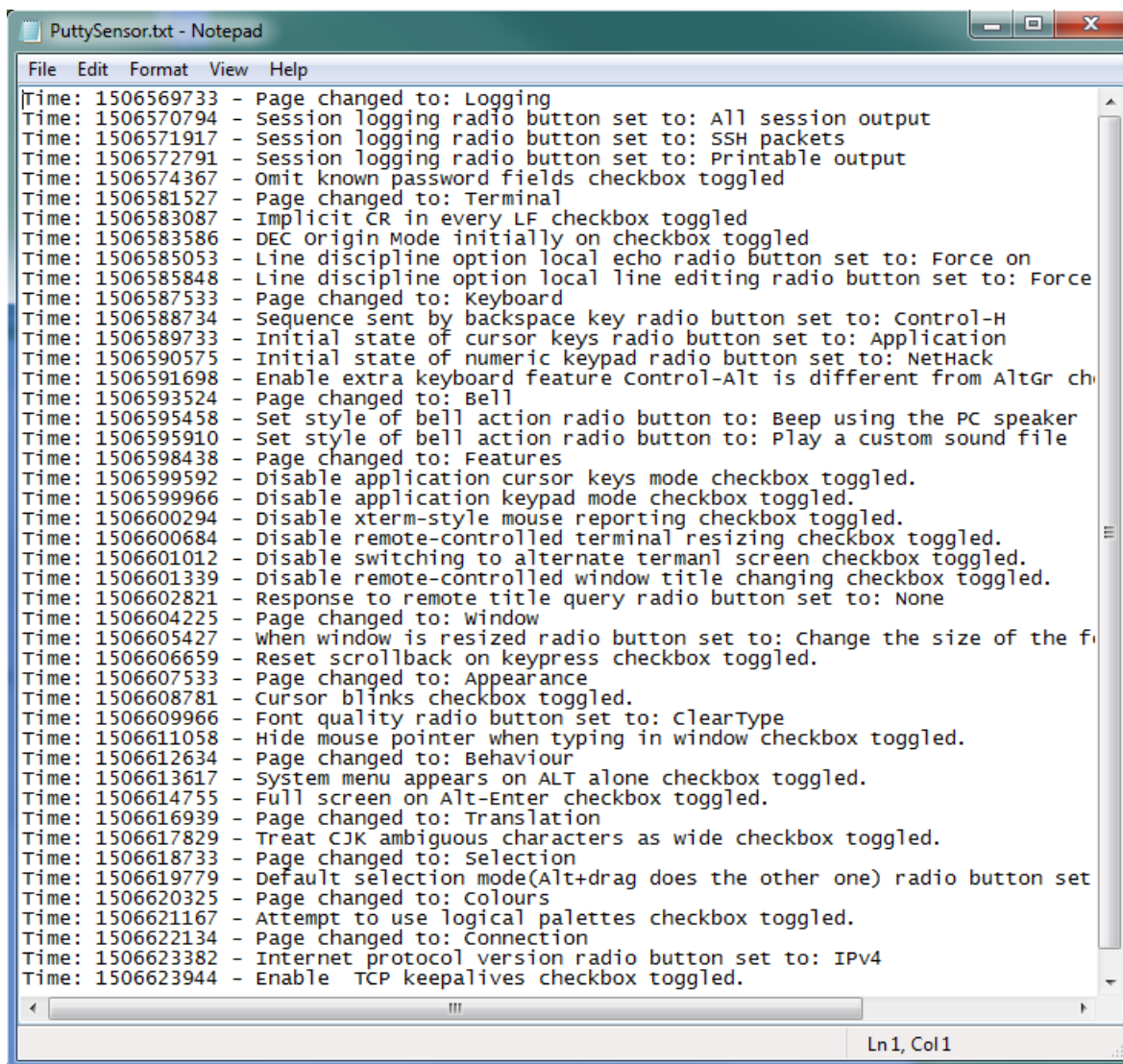


Figure 5.4: Screenshot of the PuttySensor.txt log file produced by this sensor

5.5 Conclusions

This chapter discusses `PuttySensor.dll`, which is an application level security sensor that attaches to `Putty.exe` and extracts data describing the user's interaction with the application. The sensor creates a time stamped event log of the input to the program. `Putty.exe` is an open source telnet and SSH client for Microsoft Windows and Unix. It is used for remote access to computers across the Internet [50]. The sensor this chapter describes tracks the input to the configuration dialog box of `Putty.exe`. Figure 5.1 on page 62 shows the user interface of this configuration window.

Using detours applied to the API functions `GetDlgItem()` and `IsDlgItemChecked()` an intermediate sensor is designed to extract the user interface element ID passed to each of these functions. First, the effective addresses of these functions are calculated using the base address of the `user32.dll` module in memory and an offset. Then, the offset is determined by subtracting the address of the target function from the base address. Next, the intermediate sensor is injected into the application, and these functions are detoured. Finally, the user interface item IDs are collected from the application's detoured API calls. The item IDs gathered from calls to `GetDlgItem()` are used to create a hash that identifies which options page has been selected. The item IDs collected from calls to `IsDlgItemChecked()` help determine the user interface element that has been manipulated on the current page.

Ultimately, the output of this sensor could potentially train an anomaly detection system to combat the insider threat problem. This is a task that could be explored in future work. The next chapter will show an application of dynamic sensors that targets the malicious code threat.

Chapter 6

Experimental Sensor: FunctionTimer.dll

This chapter will discuss the implementation and results of the dynamic application level sensor FunctionTimer.dll. FunctionTimer.dll is a dynamic-link library that extracts timing data on specific functions within the targeted application. After timing data is collected, the results are logged to a file. Additionally, manual analysis is performed on the log data. The attack threat this sensor addresses is malicious code that modifies the functionality of the application. The hypothesis is that the difference in timing results of the unmodified code and modified code will be statistically noticeable. The first section will discuss the target application for the developed sensor.

6.1 Target Application

GuessNumber.exe is the targeted application, and it was developed for the purposes of this experiment. This application is a simple number guessing game. It begins by prompting the user for their name and then asks them to guess a number between 1 and 10. Afterwards, the application generates a random number and compares it to the users guess. If the numbers match, the user wins. If they do not match, the user loses, and the correct answer is displayed. This process is looped until the user closes the application. GuessNumber.exe is unique because it is coded in such a way that makes it vulnerable to a buffer overflow attack.

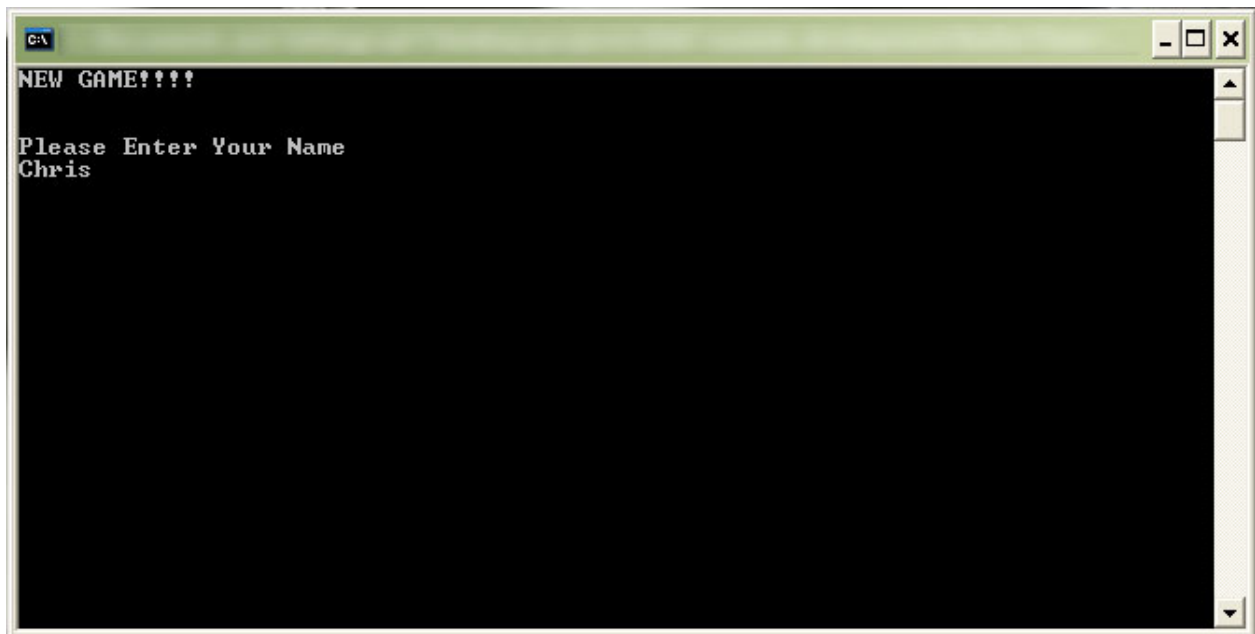


Figure 6.1: GuessNumber.exe prompting the user for their name

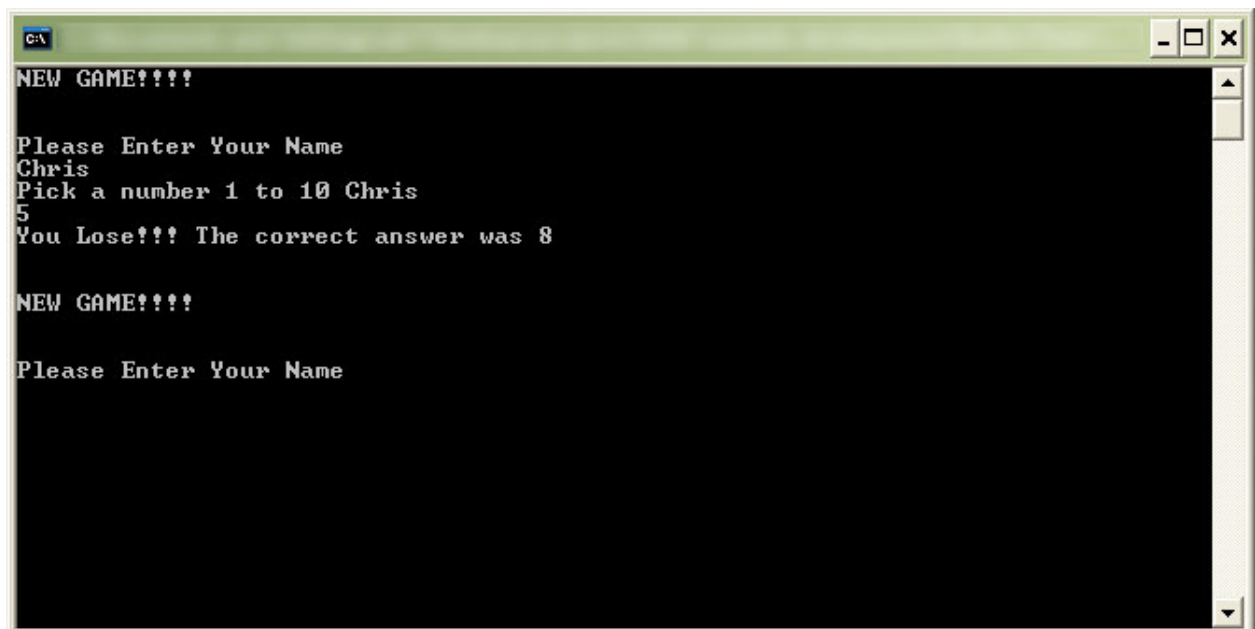


Figure 6.2: Results of a complete cycle of GuessNumber.exe

The following source code shows the targeted function that contains this vulnerability.

Code: Function Foo of GuessNumber.exe

```
void Foo(char *buf){
    char buffer[10];
    for (int i=0;i<10000;i++){
        strcpy(buffer, buf);
    }
}
```

This function creates a static character array buffer of size 10 and performs an unsafe `strcpy()` into that buffer. The source string is passed to the function as an argument. An extra loop exists to extend the normal execution time for this function. Figure 6.1 and 6.2, on the previous page, show the functionality of this game.

6.2 Reverse Engineering GuessNumber.exe

The GuessNumber.exe binary is reverse engineered in order to extract the memory address from the function `Foo()`. This memory address is used to apply a functional detour that times the execution of `Foo()`. GuessNumber.exe is simple enough that the location of `Foo()` was easily found by searching the binary for the string “Pick a number 1 to 10.” This string is printed to the screen after the targeted function is called. The following code segment is the assembly representation of `Foo()`.

Code: x86 Assembly Representation of Foo()

```
.text:0042D5A0  sub_42D5A0          // start of function
.text:0042D5A0  var_10 = dword ptr -10h // variable int i
.text:0042D5A0  var_C = byte ptr -0Ch  // variable buffer
.text:0042D5A0  arg_0 = dword ptr 8    // function arg char *buf
.text:0042D5A0
.text:0042D5A0  push ebp            // preserve frame pointer
.text:0042D5A1  mov ebp, esp        // mv stack ptr into frm ptr
.text:0042D5A3  sub esp, 50h        // make room for stack frame
.text:0042D5A6  push ebx            // push ebx to stack
.text:0042D5A7  push esi            // push esi to stack
.text:0042D5A8  push edi            // push edi for stack
.text:0042D5A9  mov [ebp+var_10], 0  // initialize int i to 0
.text:0042D5B0  jmp short loc_42D5BB // jump to loop init loop cmp
.text:0042D5B2  mov eax, [ebp+var_10] // put var i into eax
.text:0042D5B5  add eax, 1          // increment eax
.text:0042D5B8  mov [ebp+var_10], eax // put eax back into var i
```

```

.text:0042D5BB    cmp [ebp+var_10], 2710h // compare i 10000d (2710h)
.text:0042D5C2    jge short loc_42D5C6    // if i >= 10000d jump out of loop
.text:0042D5C4    jmp short loc_42D5B2    // else continue loop
.text:0042D5C6    mov eax, [ebp+arg_0]    // move char *buf into eax
.text:0042D5C9    push eax                // push to stack for upcoming fnc call
.text:0042D5CA    lea ecx, [ebp+var_C]    // load address of buffer into ecx
.text:0042D5CD    push ecx                // push to stack for upcoming fnc call
.text:0042D5CE    call sub_42B703         // call to strcpy()
.text:0042D5D3    add esp, 8              // clean up stack after strcpy()
.text:0042D5D6    pop edi                 // restore edi from stack
.text:0042D5D7    pop esi                 // restore esi from stack
.text:0042D5D8    pop ebx                 // restore ebx from stack
.text:0042D5D9    mov esp, ebp            // move frame ptr into stack ptr
.text:0042D5DB    pop ebp                 // restore ebp from stack
.text:0042D5DC    retn                    // function ends

```

This is indeed the target function, and the memory address of the function is 0x42D5A0. The next section describes how this memory address is used to implement the sensor.

6.3 Implementation

The reverse engineering stage provided the necessary information to develop the sensor. When the dynamic-link library is injected into the process, the `InitializeSensor()` function is called. This function deploys a detour on the `Foo()` function of the application using the extracted memory address, 0x42D5A0. The code for this detour is shown in the following code segment:

Code: Function detour added to 0x42D5A0

```
DetourFunctionWithEmptyTrampoline((PBYTE)cp_tramp,(PBYTE)0x42D5A0,(PBYTE)cp_detour);
```

When the `Foo()` function is called within the application, execution is redirected to the `cp_detour()` function. The detour function uses the x86 assembly instruction `RDTSC` to capture the number of processor ticks since reset. Next, the detour function calls `Foo()`. To determine the number of processor ticks for `Foo()`, the `RDTSC` instruction is called again after `Foo()` returns. Then the result is subtracted from the initial call. At this point, the clock frequency can be used to determine the execution time of the function with nanosecond scale precision. Should `Foo()` fail to return, the `PulseCall()` function, which runs within the sensors thread, can act as a watchdog timer and calculate the timing difference.

Advances in CPU architectures have diluted the accuracy of the RDTSC timing method. First, in multi-core architectures, each core has its own time stamp counter (TSC) register. This is overcome by the fact that the timing is executed within the injected sensor, which runs as a thread on a particular core. Second, the method does not account for context switches, which can inflate the timing of the function. This problem is overcome in the analysis stage by computing the average timing of the `Foo()` function. Third, out of order execution can cause the RDTSC instruction to not issue when expected. This can be solved by adding a serializing function like `CPUID`. Finally, many laptops have power saving technologies that speed up or slow down the CPU frequency to conserve the battery. When this occurs, the timing results will be off potentially skewing the sensor results. The computer on which these experiments were run on does not have this power saving feature [51].

After timing results are calculated, the sensor logs the data for analysis. At this point the sensor could be developed to automatically forward the data to the analysis center. However, the focus of this thesis is on the dynamic application level sensors and not the analysis and response steps. Future work will explore analysis and response in more detail. Additionally, the sensor implements a watchdog timer that functions at the millisecond scale to determine if `Foo()` has failed to return from execution. This is important considering one of the attacks described in the next section prevents `Foo()` from returning.

6.4 Attack Vectors

This section details two attack vectors for the targeted function `Foo()`. The first attack vector is a code tamper attack. The code of the targeted function is manipulated in such a way that modifies the functionality, resulting in abnormal timings. The second attack vector is a buffer overflow attack. The return address of the stack frame is overwritten resulting in the execution of shellcode on the stack. In this case, `Foo()` does not return.

6.4.1 Code Tamper Attack

The first type of attack developed for GuessNumber.exe is a code tamper attack. The functionality of the code is altered in memory at run time. There are several ways this could happen, such as the injection and redirection of code execution or the actual modification of the existing code. The attack developed for GuessNumber.exe modifies the existing code of the function. It changes the number of loop iterations from 10000 to 30000. This is accomplished through an API call to `WriteProcessMemory()`. This function call writes a buffer to a specified memory location in a targeted process. In this case, the targeted process is the GuessNumber.exe process. Figure 6.3 shows the graph view of the assembly code before the modification, and Figure 6.4 shows the assembly code after the modification.

6.4.2 Buffer Overflow Attack

Buffer overflow vulnerabilities are the result of an application reading or writing beyond the end of an array. Attackers exploit these vulnerabilities to force the execution of malicious shellcode on a remote system. There are several techniques to exploit a buffer overflow, such as arc injection, pointer subterfuge, and heap smashing. However, the most common technique is the “stack smashing” approach. This is when the overrun buffer writes into the stack frame of the current function and replaces the return address with the address of their shellcode [37].

The function `Foo()` contains a “stack smashing” buffer overflow vulnerability. This is the result of the small static buffer and the unsafe string copy function, `strcpy()`. Shellcode can be inserted in place of the name to force a buffer overflow. The following is an example of shellcode that will cause the buffer to overflow and overwrite the return function of the current stack frame.

Code: Hexadecimal representation of buffer overflow shellcode for function Foo()

“\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\xEB\xF0\xFC\xFE\x12”

Figures 6.5 and 6.6 show the stack before and after the shellcode has been pasted in for the name.

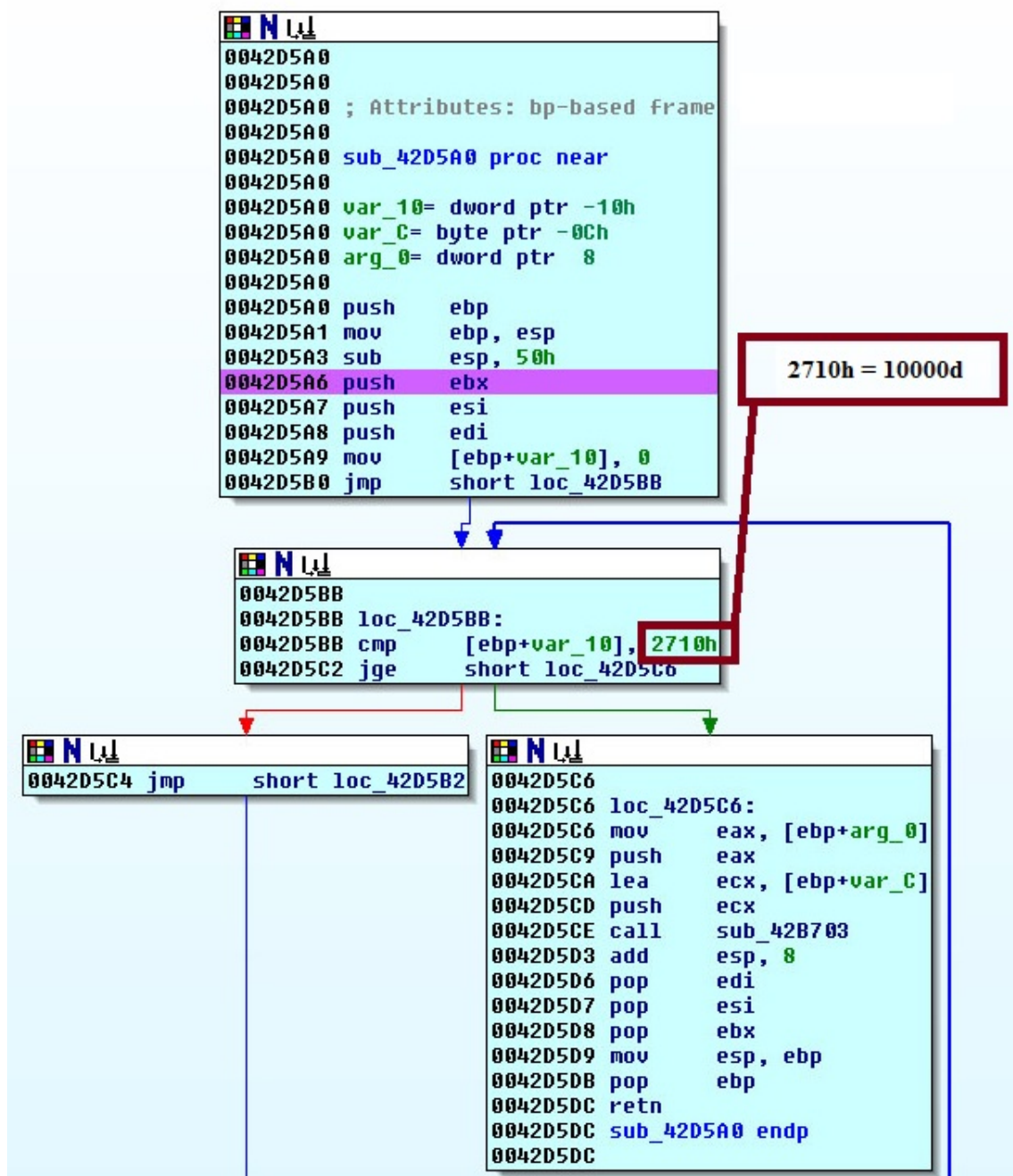


Figure 6.3: Assembly code of Foo() prior to the code tamper attack

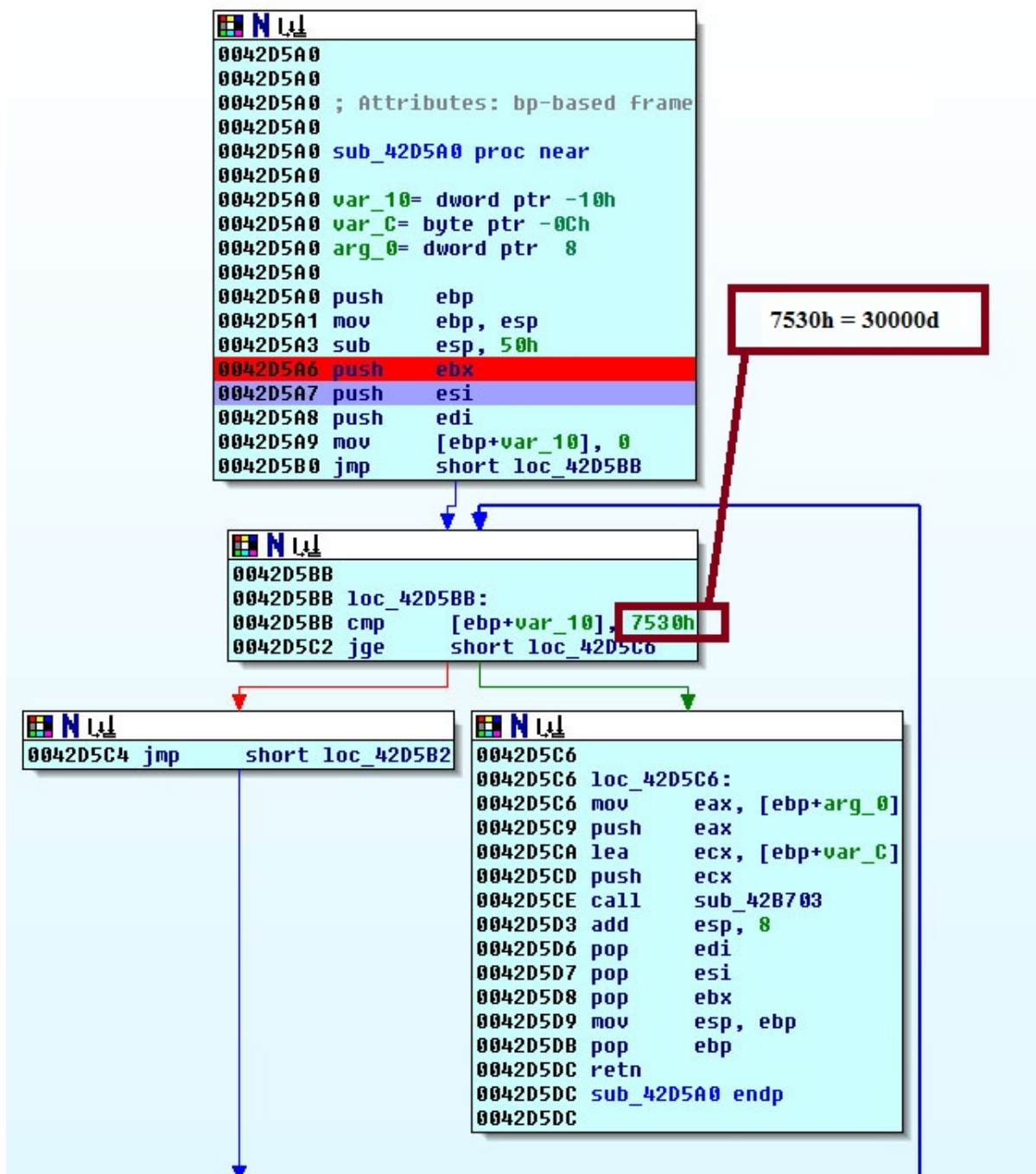


Figure 6.4: Assembly code of Foo() after the code tamper attack

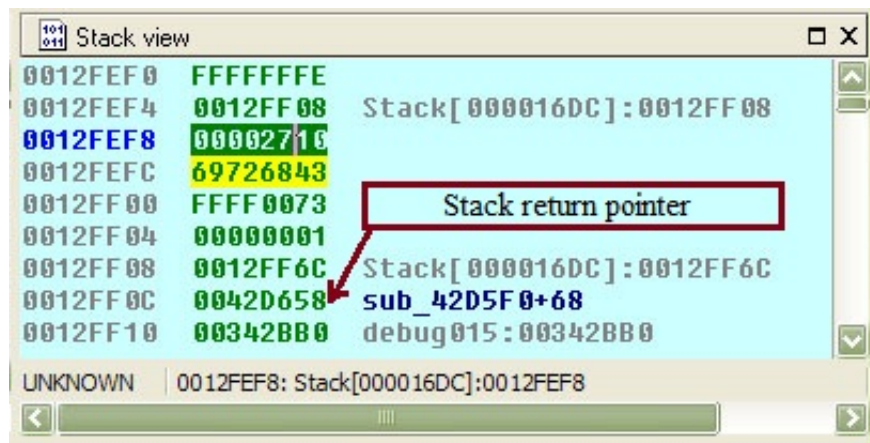


Figure 6.5: Stack before buffer overflow

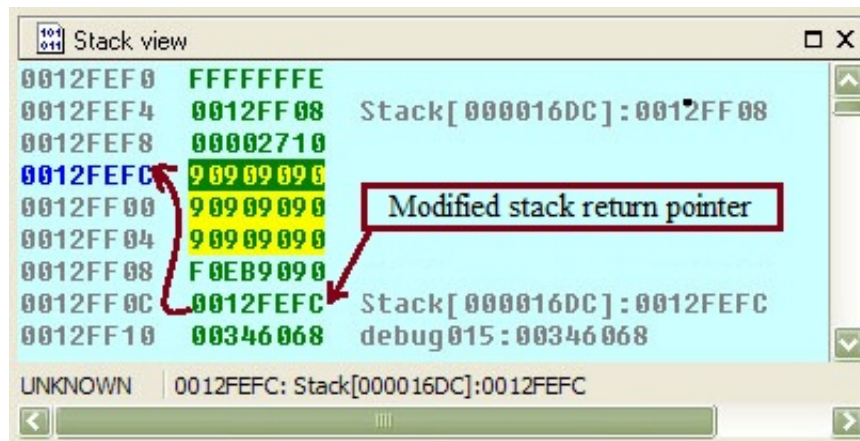


Figure 6.6: Stack after buffer overflow

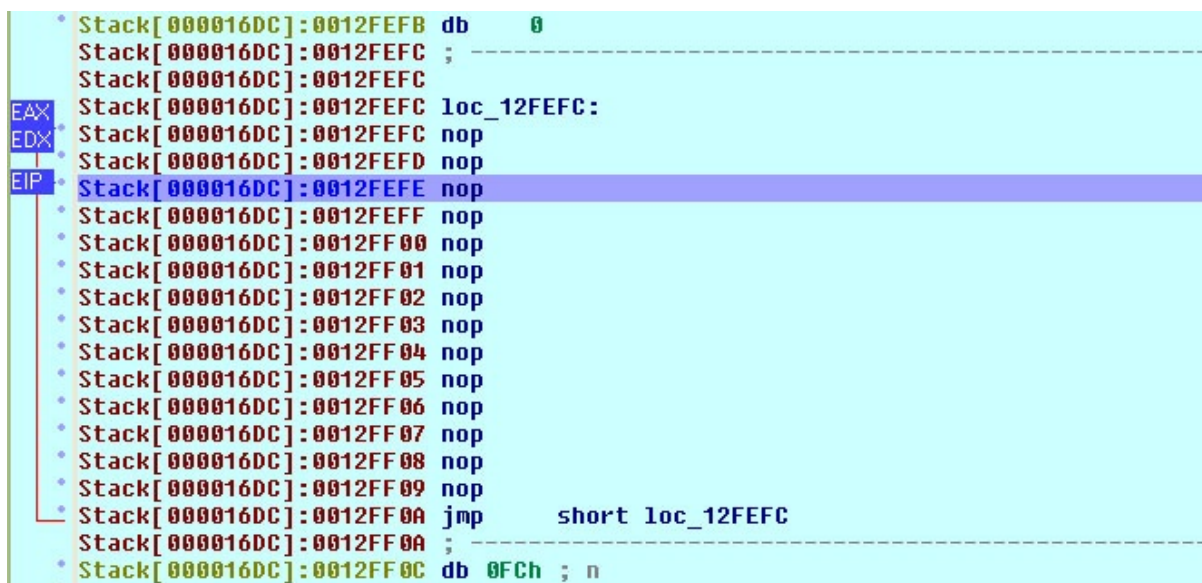
Code: ASCII representation of buffer overflow shellcode for function Foo()

```
"EEEEEEEEEEEEEEEEδ≡=■^R"
```

The ASCII representation of the shell code, which is shown in the above code segment, is pasted into GuessNumber.exe when the user is prompted for their name. The shellcode begins executing after the Foo() function returns. The code is returned to the new overwritten return address. A breakpoint was inserted to allow line by line execution. The shellcode does nothing more than hang execution in a loop on the stack. Figure 6.7 shows the assembly instruction sequence executing on the stack.

6.5 Results

The sensor is injected into the GuessNumber.exe process and the timing data is accumulated. Next, the attacks are executed and the sensor collects data through these events. This section will detail the results of the dynamic application level sensor FunctionTimer.dll.



```
* Stack[000016DC]:0012FEFB db 0
Stack[000016DC]:0012FEFC ; -----
Stack[000016DC]:0012FEFC
Stack[000016DC]:0012FEFC loc_12FEFC:
Stack[000016DC]:0012FEFC nop
Stack[000016DC]:0012FEFD nop
Stack[000016DC]:0012FEFE nop
Stack[000016DC]:0012FEFF nop
Stack[000016DC]:0012FF00 nop
Stack[000016DC]:0012FF01 nop
Stack[000016DC]:0012FF02 nop
Stack[000016DC]:0012FF03 nop
Stack[000016DC]:0012FF04 nop
Stack[000016DC]:0012FF05 nop
Stack[000016DC]:0012FF06 nop
Stack[000016DC]:0012FF07 nop
Stack[000016DC]:0012FF08 nop
Stack[000016DC]:0012FF09 nop
Stack[000016DC]:0012FF0A jmp short loc_12FEFC
Stack[000016DC]:0012FF0A ; -----
Stack[000016DC]:0012FF0C db 0FCh ; n
```

Figure 6.7: Shellcode executing on the stack

6.5.1 Results of Code Tamper Attack

The results of the code tamper attack supported the hypothesis. Prior to the code tamper attack, the timing results were approximately 30,500 ns. After the code tamper attack, the timing results jumped to approximately 90,500 ns. This can be seen in Table 6.1 on the next page.

The average normal time for this experiment is 30,534 ns with a standard deviation of 880 ns. When the tamper attack is launched, the average time for the tamper timings is 90701 ns. This represents a z-score that is 68.39 standard deviations away. The hypothesis is that the difference in timing results of the unmodified code and modified code is statistically noticeable. These results indicated that this is indeed the case. Figure 6.8 shows the scatter plot of the timing results from the tamper attack. This analysis was produced from the log files of the FunctionTimer.dll sensor. While the analysis performed on the produced data set was done by hand, the eventual goal is to perform automatic analysis in real time and use the results to actively respond to threats.

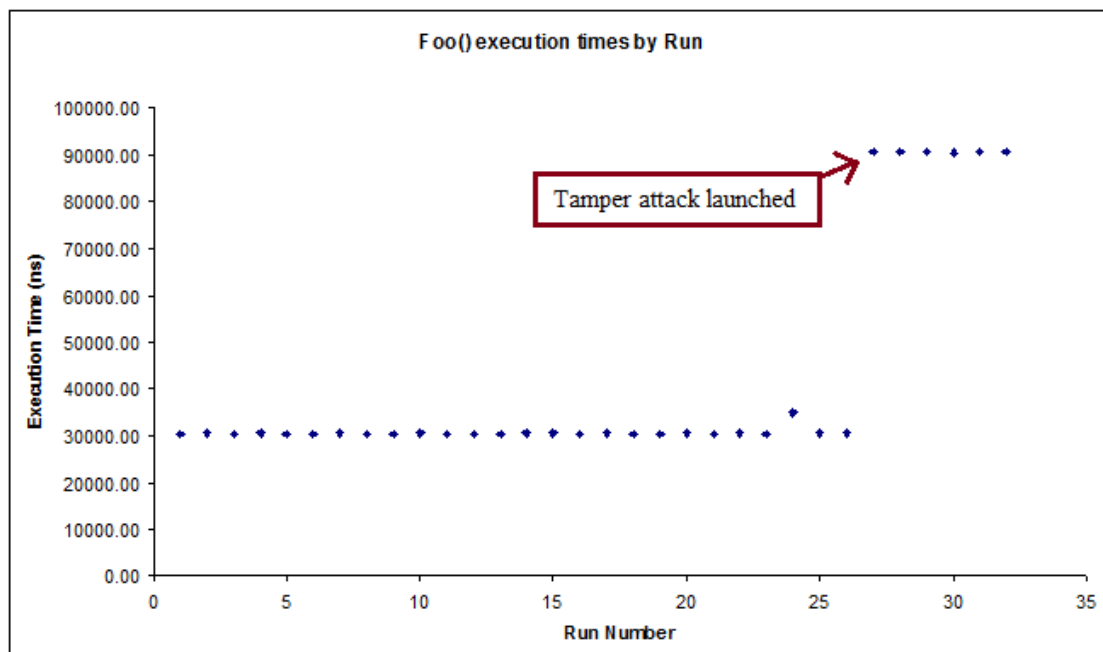


Figure 6.8: Scatter plot of the timing results of the tamper attack

Table 6.1: Observed Execution Times for 32 Runs with a Code Tamper Attack at Run 27

Run	Time(ns)	Observation
1	30674.26	Normal
2	30565.97	Normal
3	30481.74	Normal
4	30385.48	Normal
5	30538.90	Normal
6	30301.25	Normal
7	30307.27	Normal
8	30385.48	Normal
9	30454.67	Normal
10	30490.77	Normal
11	30379.46	Normal
12	30301.25	Normal
13	30499.79	Normal
14	30421.58	Normal
15	30295.23	Normal
16	30415.56	Normal
17	30301.25	Normal
18	30295.23	Normal
19	30620.12	Normal
20	30385.48	Normal
21	30385.48	Normal
22	30394.50	Normal
23	30064.47	Normal
24	30478.73	Normal
25	30400.52	Normal
26	30789.00	Normal
27	90744.31	Tamper Attack
28	90687.16	Tamper Attack
29	90774.39	Tamper Attack
30	90596.91	Tamper Attack
31	90750.33	Tamper Attack
32	90654.07	Tamper Attack

6.5.2 Results of Buffer Overflow Attack

The results of the buffer overflow attack were also as expected. The buffer overflow causes the program to hang in a loop on the stack. As a result execution never returns from the function `Foo()`. This causes the periodic check to see if the function has returned to fail. When this check fails, the sensor calculates the time since the start of `Foo()` and updates the event log. The user is also notified of the event with an alert. This alert can be seen in Figure 6.9. Finally, after the attack has occurred, the application needs to be restarted because execution is stuck in the shellcode.

The timing results of normal execution and the timeout of the `Foo()` function can be seen in Table 6.2. The results are as expected; an average normal execution time of 30,697 ns with a standard deviation of 1237 ns. The z-score of the timed out buffer overflow attack represents a z-score of 19821.59 standard deviations. Figure 6.10 shows the scatter plot of the execution time versus run number.



Figure 6.9: The watchdog timer detects that `Foo()` never returns and alerts the user.

Table 6.2: Observed Execution Times for 32 Runs with a Buffer Overflow attack at Run 32

Run	Time(ns)	Observation
1	30593.04	Normal
2	30301.25	Normal
3	30490.77	Normal
4	30550.93	Normal
5	30391.50	Normal
6	30316.29	Normal
7	33258.28	Normal
8	30496.78	Normal
9	30295.23	Normal
10	30394.50	Normal
11	30400.52	Normal
12	30484.75	Normal
13	36871.08	Normal
14	30415.56	Normal
15	30475.72	Normal
16	30304.26	Normal
17	30292.23	Normal
18	30298.24	Normal
19	30391.50	Normal
20	30448.65	Normal
21	30406.54	Normal
22	30328.32	Normal
23	30568.98	Normal
24	30466.70	Normal
25	30295.23	Normal
26	30301.25	Normal
27	30292.23	Normal
28	30322.31	Normal
29	30298.24	Normal
30	30385.48	Normal
31	30457.68	Normal
32	24582366.57	Buffer Overflow Attack

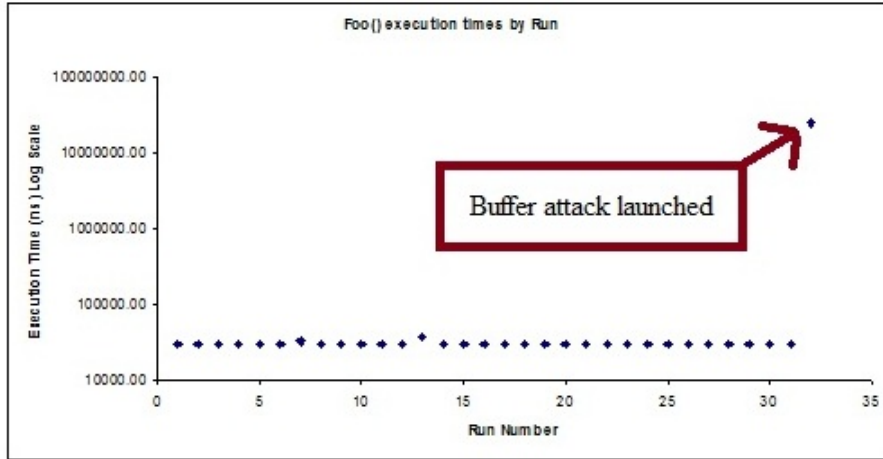


Figure 6.10: Scatter plot of the timing results of the buffer overflow attack

6.6 Conclusions

This chapter discussed the implementation and results of the dynamic application level sensor FunctionTimer.dll. This sensor collects timing data on the execution time of the target function. In the case of FunctionTimer.dll, the target function is `Foo()`, which resides in `GuessNumber.exe`. `GuessNumber.exe` is a guessing game application that was developed specifically for the purposes of these experiments. The memory address of the function `Foo()` is extracted by reverse engineering the binary. Using this memory address, a sensor is developed and deployed. This sensor uses the TSC register to calculate the number of ticks since reboot before and after the `Foo()` function is called. From this and the processor frequency, the timing of the function is calculated and recorded. Additionally, a periodic timer checks to make sure the `Foo()` function has returned.

Two experiments tested the hypothesis that the difference in timing results of the unmodified code and modified code will be statistically noticeable. The sensor is injected into the application at run time and normal timing data is collected. After a certain number of runs, two different attacks are launched on the application. The first attack is a code tamper attack that modifies the functionality of the `Foo()` function. This causes it to take three times longer to execute. The sensor data verifies this, and the change point is easily identifiable. The resulting execution time has a z-score of 68.39, which is statistically significant. The second attack is a buffer overflow attack that

overwrites the return pointer of the current stack frame, causing shellcode to execute on the stack. As a result, `Foo()` never returns from execution, and the watchdog timer of the sensor captures this event. The user is notified, and the time captured by the watchdog timer is logged. The execution time of `Foo()` is theoretically infinity since it never finishes execution. However, the watchdog is set in place to prevent this.

The work presented in this thesis focuses on the sensors and not the analysis or response. Some of the future work in this area will be to automatically perform analysis on the collected data. The `FunctionTimer.dll` sensor shows another the type of data that can be collected with an application level security sensor.

Chapter 7

Summary and Conclusion

The work presented in this thesis introduced the concept of dynamic application level sensors produced via reverse engineering and implemented via process injection. Dynamic application level security sensors provide the opportunity to use a finer granularity of data to make informed decisions about potential cyberthreats. Adding these new sensors to the existing defense in depth scheme can further strengthen the overall security of cyberspace. First, this chapter will summarize the existing defense techniques and examine where dynamic application level sensors fit in the overall security design. Second, it will recap on the technologies required to develop and implement these fine grain sensors. Third, it will review the results of the two examples, PuttySensor.dll and FunctionTimer.dll. Fourth, it will discuss the ethical implications of this work, and finally it will conclude with future work in the area.

7.1 Summary

Cyberspace is a large dynamic network of hardware, software, operating systems, data, and people. The sophisticated complexity of this network makes protecting it an ambitious task. Cyberattacks are growing exponentially in numbers. Along with this increase in the number of attacks is an increase in the sophistication of the attacks. Attackers are constantly looking for new ways to compromise systems. The motives behind these attacks include revenge, sabotage, terrorism, money, and political purposes.

Threats manifest at many different security domains throughout cyberspace. To combat these threats, layered defenses, such as defense in depth security strategies have become the standard. The combination of hardware and software security measures at the different levels of the cyberspace domain has provided the best protection currently available. The network level offers firewalls, switches, and network intrusion detection systems based on signatures and anomaly detection algorithms. The host level offers protection via antivirus scanners, system integrity verifiers, and other security measures that monitor activity on a given computer system. The common theme between the network and host level security domains, is that they follow a generic three step process. First, they collect data from sensors. Second, they process the data, and finally, they respond to any potential threats. This three step process is lacking at the application level. Defenses at this level focus on secure coding practices and patching vulnerable code. Unfortunately, this is not sufficient to protect against the complex attacks common in cyberspace. The work presented in this thesis targets application level security. It proposes adding the three step process at the application level starting with sensors. The sensors are dynamically added to the application at run time using a form of code injection.

There are several forms of code injection, such as SQL, XSS, and DLL methods. Of the forms described in Chapter 3, dynamic-link library injection is unique, because it allows code to execute within the virtual memory space of a targeted process. Execution within the virtual memory space of a remote process grants the DLL access to run time application data that is not normally accessible to other applications. There are many ways to implement DLL injection. Of these methods, the two most common are Windows APIs and Windows hooks. Windows APIs work by issuing the proper sequence of functions provided by the operating system. Windows hooks work by hooking messages for local and remote processes. Detecting code injection partially depends on the method of injection. Some forms of injection are easier to detect than others. Chapter 3 detailed different ways to detect code injection, but there are stills ways to avoid detection if stealthy execution is a required feature of the injection.

Existing applications of dynamic-link library injection are malware, functionality additions, debugging, and performance profiling. The Storm computer worm was an example of a piece of malware that used a form of code injection similar to DLL injection to execute stealthily. MMORelay is an example of DLL injection developed to add new functionality to an existing software system. The work presented in this thesis introduces a new application of dynamic-link library injection focused on cybersecurity. This new application injects security modules into applications at run time to help protect them. In order to accomplish this, the sensors must be developed.

Developing dynamic application level sensors is a multi-stage process. The first stage is to determine what data needs to be collected. This determination is dependant on the targeted security threat and the application chosen to defend against. The next stage is to reverse engineer the application to extract the memory addresses for target functions that can be detoured and critical data that can be directly accessed. The reverse engineering process is a feedback cycle of static and dynamic analysis that results in intermediate sensors that can assist in the next iteration of dynamic analysis. The end result is the necessary memory addresses for the next stage, which is to automate the data extraction.

Data extraction is accomplished by writing the actual sensor. The sensor implements any function detours and applies any processing required to report the targeted data. At this stage, the logic can be added to direct the output of the sensor to the analysis and response steps of the three step process. This thesis does not focus on the analysis and response steps, however, future work will likely explore these steps. This process was implemented on two sensors to show the kind of data that can be extracted with dynamic application level sensors.

Chapter 5 introduced PuttySensor.dll, which is an application level sensor that targets the insider threat. It collects user input data to the program Putty.exe, which is an open source SSH and Telnet client for Windows and UNIX. The putty sensor detours `GetDlgItem()` and `IsDlgButtonChecked()` in order to determine what the user is doing within the application. Whenever a user changes pages or manipulates a user interface element, the sensor reports the action in

an event log. An anomaly detection system could train on the output of this sensor to profile the normal behavior of the user within the target application. It could then respond to any deviations of this profiled normal behavior.

Chapter 6 introduced the next sensor which was `FunctionTimer.dll`. This sensor collected timing data on the execution of functions within an application. The hypothesis was that the timing results of the unmodified code and modified code would be statistically noticeable. To test this hypothesis, the application `GuessNumber.exe` was created. The program prompts the user for their name and a number between 1 and 10. The program itself was coded in such a way to be vulnerable to several types of attacks. The experiment launched two attacks on the program. The first attack tampered with the logic of a timed function. The result was a timing that took three times longer than normal to execute. This longer execution time represented a z-score of over 68 standard deviations away. The second attack was a buffer overflow that smashed the return pointer of the stack frame and forced the execution of shellcode on the stack. The result was the target timed function never returned, triggering the watchdog timer the sensor implemented. The attacks were noticeable through change points in the timing logs the sensor produced. There is much potential future work in this field. The next section will detail several directions in which this work could go.

7.2 Future Work

The work presented in this thesis opens the door for many possibilities of additional work. First, different forms of analysis and response need to be explored in an effort to make use of the powerful fine-grain data dynamic application level sensors provide.

Second, many more sensors could be designed. This would allow for a robust defense at the application level. Ideally, sensors would run within each process on the system. They would monitor the health of the process and defend against threats both internal and external to the host and particular application.

Third, a robust framework should be developed to facilitate the design of these sensors. This framework should have two parts. The first part is the development center, which would provide APIs for rapidly creating and testing sensors. It would also help automate the reverse engineering process as much as possible. The second part is the deployment functionality of the framework. The deployment functionality actually performs the DLL injection and keeps track of which sensors are loaded in which applications and processes. It would allow system administrators control over the sensors, analysis, and response. A large part of the framework should be a high quality visualization capability that assists system administrators in understanding what the status of the system is at any given time. This visualization capability would also have the necessary user interface controls to launch or remove sensors to the system.

Finally, additional work could focus on new methods of stealthy execution. This is of particular interest because targeting the insider threat means the system should be transparent to the user. In general, a system such as the one proposed in this thesis would be invisible to the average user. However, advanced users could still track down and disable the system. To prevent this, new methods of stealthy executions could be explored in future work.

The goal of the reverse engineering stage is to extract memory locations for critical data elements and functions that can be hooked. This section will focus on the reverse engineering process, and detail the tools and methods used to extract the required data for the security sensor.

7.3 Ethical Implications of this Work

The work presented in this thesis has ethical implications that need to be considered. The preface noted that the techniques described in this document could potentially be used to develop harmful sensors. Is it right to develop a framework that facilitates code injection and application level data mining, even if its intended use is to defend? Are the ethical implications of this same as the invention of dynamite? Dynamite is extremely useful in applications, such as clearing rock for the development of roads and the demolition of buildings. However, it can also be used to do harm.

Other ethical implications to consider involve privacy. For Example:

- If Person X owns a computer and Person Y uses it, is it right to develop a sensor to monitor Person Y's activity?
- What limits of privacy should Person Y expect?
- How does the relationship between Person X and Person Y effect the ethics?
- What if Person Y is a minor and Person X is their guardian?
- What if Person X is a public library providing free internet access?
- What if Person X is a company that wishes to monitor their employee?
- What obligation does Person X have to keep the information confidential? How confidential?
- Can another employee access the collected data if they are working on behalf of the employer?
- Should the employer monitor the actions of the employee in charge of monitoring other employees?
- Should Person X disclose to Person Y that they are being monitored?
- If so, how much detail should be disclosed about what is being monitored?
- The type of data that could potentially be collected through application level security sensors is personally identifiable information, such as passwords, social security numbers, and credit card numbers. How should this data be handled?

These are just a few of the ethical questions that should be considered when developing and implementing dynamic application level security sensors. Ultimately, there are no definitive answers to these questions. The answers depend on the situation and the individuals involved. There are many code of ethics that can be applied to each of these scenarios, but in the end the answer depends on the interpretation.

7.4 Conclusion

The need for stronger application level security domain defenses is apparent. Secure code practices and vulnerability patching are not sufficient. Attackers are regularly probing enterprise networks looking for vulnerabilities. If they are successful, the results can be costly. The work presented in this theses proposes the use of dynamic-link library injection and reverse engineering to dynamically add security sensors to applications at run time. These sensors further fortify the defense-in-depth security strategy. They can be used to access a previously untapped source of fine-grain data and should be incorporated into normal operations of system administrators. It is not the absolute solution to the rising cyber security problems, but it is a previously missing piece of the solution.

Bibliography

- [1] M. Alvarez “Online Black Market Worth \$267 Million”, *Atelier BNP Paribas Group* November 26, 2008 [Online], <http://www.atelier-us.com/e-business-and-it/article/online-black-market-worth-267-million>
- [2] Blaster, Accessed: February 8, 2010 [Online] [http://en.wikipedia.org/wiki/Blaster_\(computer_worm\)](http://en.wikipedia.org/wiki/Blaster_(computer_worm))
- [3] Bro, Accessed: February 6, 2010 [Online] <http://bro-ids.org>
- [4] G. W. Bush, “The National Strategy to Secure Cyberspace”, *The White House*, February 2003.
- [5] Cheat Engine, *Hex-Rays*, Accessed: February 16, 2010 [Online], <http://www.cheatengine.org/>.
- [6] Daphne, “How Bypass firewall with Process Injection”, January 3, 2009 [Online], <http://www.abyssec.com/blog/2009/01/how-bypass-firewall-with-process-injection/>.
- [7] Darawk, “Dll Injection Tutorial”, *The Edge of Nowhere* Mar 18, 2006 [Online Forum], <http://www.edgeofnowhere.cc/viewtopic.php?p=2483118>.
- [8] S. Fewer, “Reflective DLL Injection”, *Harmony Security*, Version 1.0 October 31, 2008.
- [9] E. A. Fischer, “Creating a National Framework for Cybersecurity: An Analysis of Issues and Options”, *CRS Report for Congress*, February 22, 2005.
- [10] eEye Digital Security, August 11, 2003 [Online], <http://research.eeye.com/html/advisories/published/AL20030811.html>
- [11] S. Floyd and V. Paxson, “Difficulties in Simulating the Internet”, *IEEE/ACM Transactions on Networking*, Vol.9, No.4, pp. 392-403, February 2001.
- [12] F-Secure Labs, “F-Secure IT Security Threat Summary for the Second Half of 2008”, 2008 Q4 [Online], <http://www.f-secure.com/en.EMEA/security/security-lab/latest-threats/security-threat-summaries/2008-4.html>.

- [13] A. Greenberg, “The iPhone’s First Worm”, November 8, 2009 [Online], <http://www.forbes.com/2009/11/08/iphone-virus-attack-technology-security-rickrolling-cybersecurity.html>.
- [14] R. Graham, “FAQ: Network Intrusion Detection Systems”, Version 0.8.3, March 21, 2000.
- [15] G. Hunt and D. Brubacher “Detours: Binary Interception of Win32 Functions”, *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.
- [16] Iczelion, “Tutorial 24: Windows Hooks”, Accessed: February 13, 2010 [Online], <http://win32assembly.online.fr/tut24.html>.
- [17] IDA Pro, “The IDA Pro Disassembler and Debugger”, *Hex-Rays* Accessed: February 16, 2010 [Online], <http://www.hex-rays.com/idapro/>.
- [18] INFOSEC, “Hard Problem List”, *INFOSEC Research Council*, November 2005.
- [19] Internet Security Systems, “Network-vs. Host-based Intrusion Detection, A Guide to Intrusion Detection Technology”, October 2, 1998.
- [20] Interagency Working Group on Cyber Security and Information Assurance, “Federal Plan for Cyber Security and Information Assurance Research and Development”, *National Science and TEchnology Council*, April 2006.
- [21] Kaspersky Labs, “Kaspersky Lab forecasts ten-fold increase in new malware for 2008”, April 9, 2009 [Online], <http://www.kaspersky.com/news?id=2075756291>.
- [22] M. Keeney, E. Kowalski, D. Cappelli, A. Moore, T. Shimeall, S. Rogers, “Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sensors”, *National Threat Assessment Center and CERT Program Software Engineering Institute*, May 2005.
- [23] G. Keizer, “Windows Market Share Slide Resumes” *Computer World* January 3, 2010 [Online], http://www.computerworld.com/s/article/9142978/Windows_market_share_slide_resumes.
- [24] J. E. Kerivan, “Self-Defending Security Software”, *MILCOM 2005*, October 2005.

- [25] R. Kuster, “Three Ways to Inject Your Code into Another Process”, *The Code Project* July 25, 2003 [Online], <http://www.codeproject.com/KB/threads/winspy.aspx>.
- [26] J. R. Langevin, M. T. McCaul, S. Charney, H. Raduege, and J. A. Lewis, “Securing Cyberspace for the 44th Presidency”, *Report of the CSIS Commision on Cybersecurity for the 44th Presidency*, December 2008.
- [27] T. Lanowitz, “Now Is the Time for Security at the Application Level”, *Gartner*, December 1, 2005.
- [28] D. Marcus, “Malware Is Their Business...and Business is Good!”, *Avert Labs McAfee* July 22, 2009 [Online], <http://www.avertlabs.com/research/blog/index.php/2009/07/22/malware-is-their-businessand-business-is-good/>.
- [29] National Security Agency, “Defense in Depth: A practical strategy for achieving Information Assurance in today’s highly networked environments.”, *Information Assurance Solutions Group*,
- [30] Net-Security, “Trojan attacks antivirus software”, June 27, 2006 [Online], http://www.net-security.org/malware_news.php?id=655.
- [31] D. Newman, J. Snyder, and R. Thayer, “Crying wolf: False alamrs hide attacks”, *Network World*, June 24, 2002 [Online], <http://www.networkworld.com/techinsider/2002/0624security1.html>.
- [32] M. G. Nystrom, “SQL Injection Defenses”, *O’Reiliy Media, Inc.*, March 26, 2007
- [33] OSSEC, Accessed: February 10, 2010 [Online] <http://www.ossec.net/>
- [34] PandaLabs, “PandaLabs’ 2009 Predictions: Malware Will Increase In 2009”, December 21, 2008 [Online], <http://www.prweb.com/releases/2008/12/prweb1772314.html>.
- [35] Patch Tuesday, Accessed: February 11, 2010 [Online] http://en.wikipedia.org/wiki/Patch_Tuesday

- [36] T. Pietraszek, and C. V. Berghe, “Defending against Injection Attacks through Context-Sensitive String Evaluation”, *Proceedings of Recent Advances in Intrusion Detection (RAID2005)*, September 2005.
- [37] J. Pincus and B. Baker, “Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns”, *IEEE Security and Privacy*, pages 20-27, July 2004.
- [38] D. Quist, “Storm Worm Process Injection from the Windows Kernel”, *Offensive Computing*, March 9, 2008.
- [39] J. R. Raphael, “Twitter Hack: How It Happened and What’s Beign Done”, *PCWorld* Jan 5, 2009 [Online], http://www.pcworld.com/article/156359/twitter_hack_how_it_happened_and_whats_being_done.html.
- [40] Rocky Pulley, “Extending Task Manager with DLL Injection”, *The Code Project* May 19, 2005 [Online], <http://www.codeproject.com/KB/threads/taskex.aspx>.
- [41] SCADA, Accessed: February 8, 2010 [Online] <http://en.wikipedia.org/wiki/SCADA>
- [42] Skape and J. Turkulainen, “Remote Library Injection”, *nologin*, April 6, 2004.
- [43] B. Schneier, “Gathering ‘Storm’ Superworm Poses Grave Threat to PC Nets”, *Wired* October 4, 2007 [Online], <http://www.wired.com/politics/security/commentary/securitymatters/2007/10/securitymatters.1004>.
- [44] R. Seacord, and M. Sebor, “Top 10 Secure Coding Practices”, *CERT*, February 13, 2010.
- [45] E. Skoudis, “Your desktop antivirus product may be leaving you wide open to attack”, *SearchSecurity* June 1, 2004 [Online], http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci967559_mem1,00.html?ShortReg=1&mboxConv=searchSecurity_RegActivate_Submit&&ShortReg=1&mboxConv=searchSecurity_RegActivate_Submit&.
- [46] Snort, Accessed: February 6, 2010 [Online] <http://www.snort.org>
- [47] Sourcefire, Inc, “SNORT Users Manual 2.8.5” *The Snort Project* October 22, 2009

- [48] SQL Injection, Accessed: February 11, 2010 [Online] http://en.wikipedia.org/wiki/SQL_injection
- [49] Storm Worm, Accessed: February 10, 2010 [Online] http://en.wikipedia.org/wiki/Storm_Worm
- [50] S. Tatham, “Putty: A Free Telnet/SSH Client” Accessed: February 28, 2010 [Online] <http://www.putty.nl/>
- [51] Time Stamp Counter Accessed: February 30, 2010 [Online] http://en.wikipedia.org/wiki/Time_Stamp_Counter
- [52] D. Turner, T. Mack, M. K. Low, M. Fossil, J. Blackbird, D. McKinney, E. Johnson, S. Entwistle, and C. Wuesst, “Symantec Internet Security Threat Report: Trends for July-December 07”, *Symantec*, Volume XIII, April 2008.
- [53] A. Wiegstein, M. Schumacher, X. Jia, and F. Weidemann, “The Cross Site Scripting Threat”, *Virtual Forge*, Version 1.2, May 10, 2007.

Appendix

Appendix A

Screenshots of Reverse Engineering Tools

This section displays the screenshots of several of the reverse engineering tools described in this thesis.

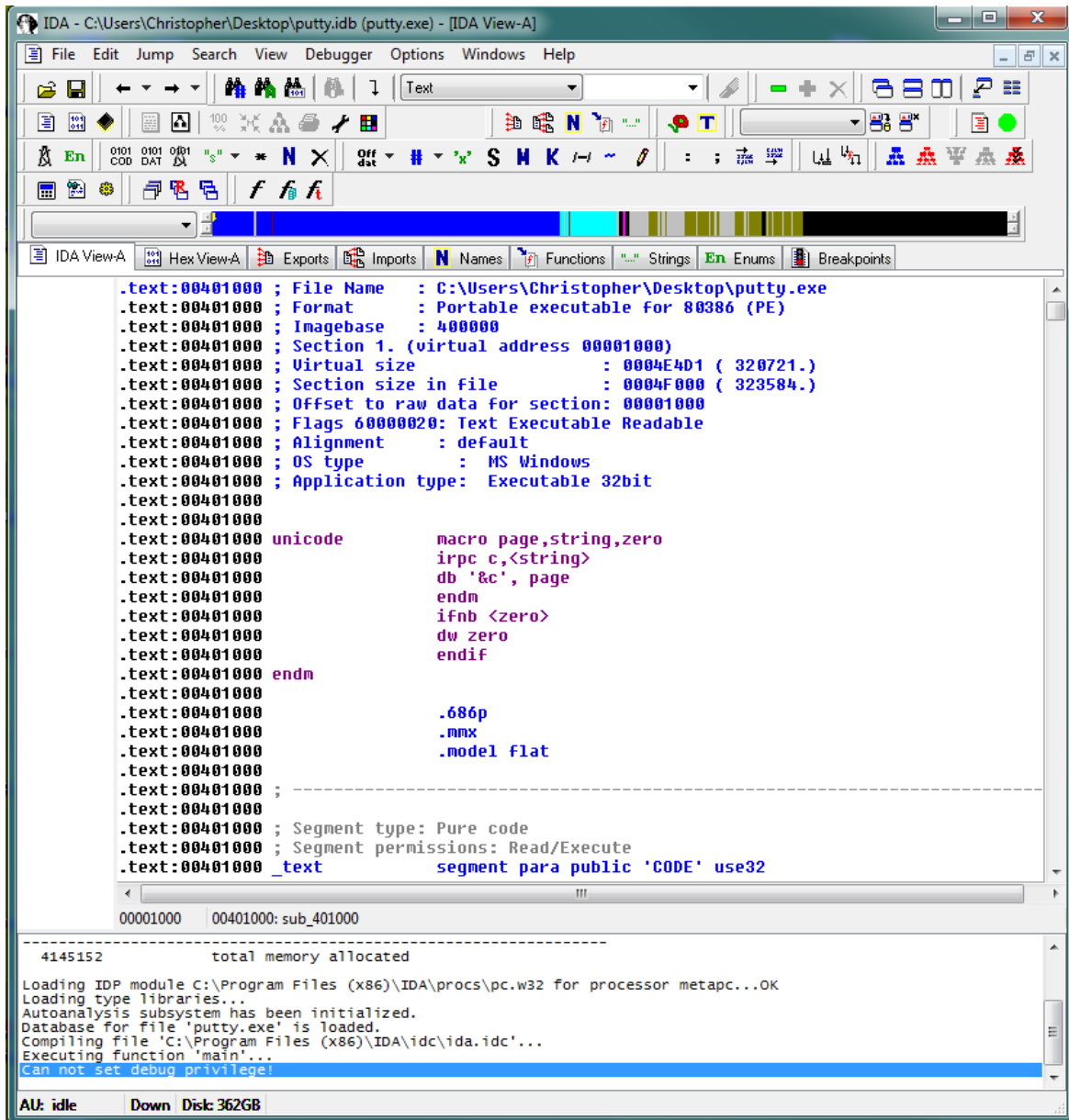


Figure A.1: Screenshot of the disassembler, IDA Pro

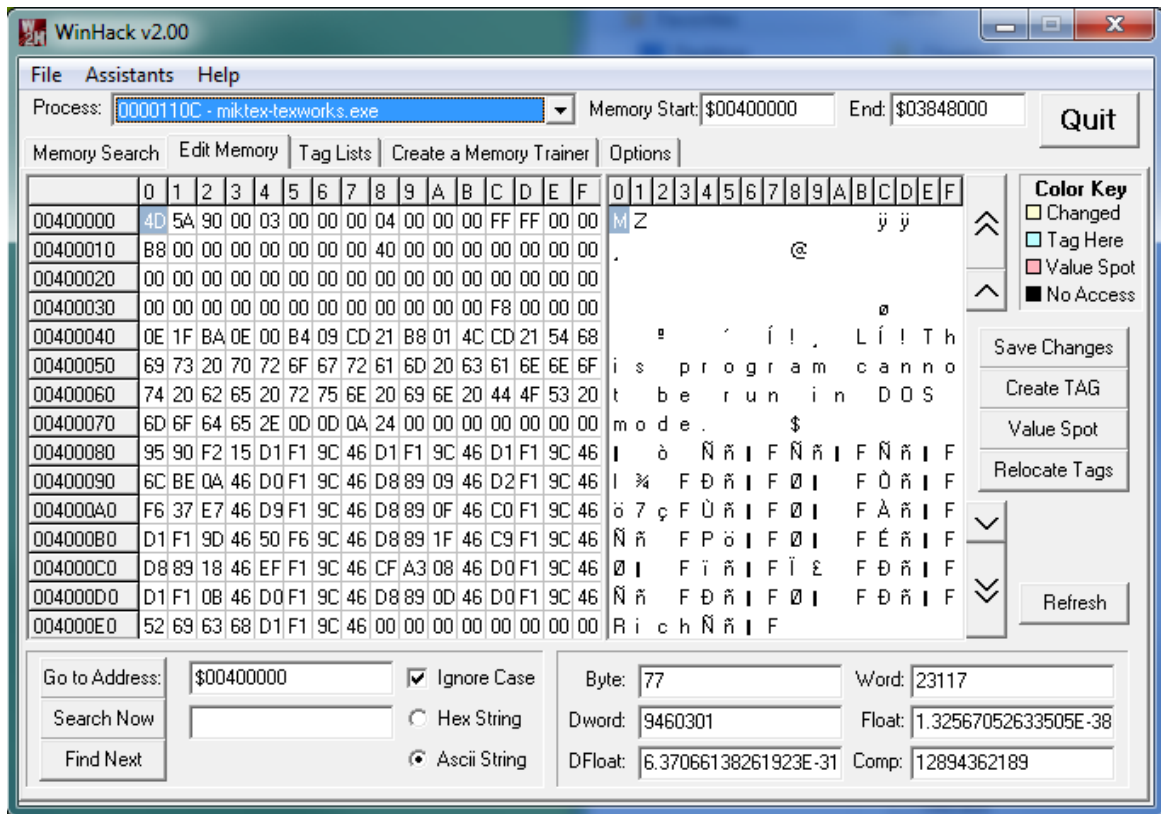


Figure A.2: Screenshot of the memory editor, Winhack 2.0

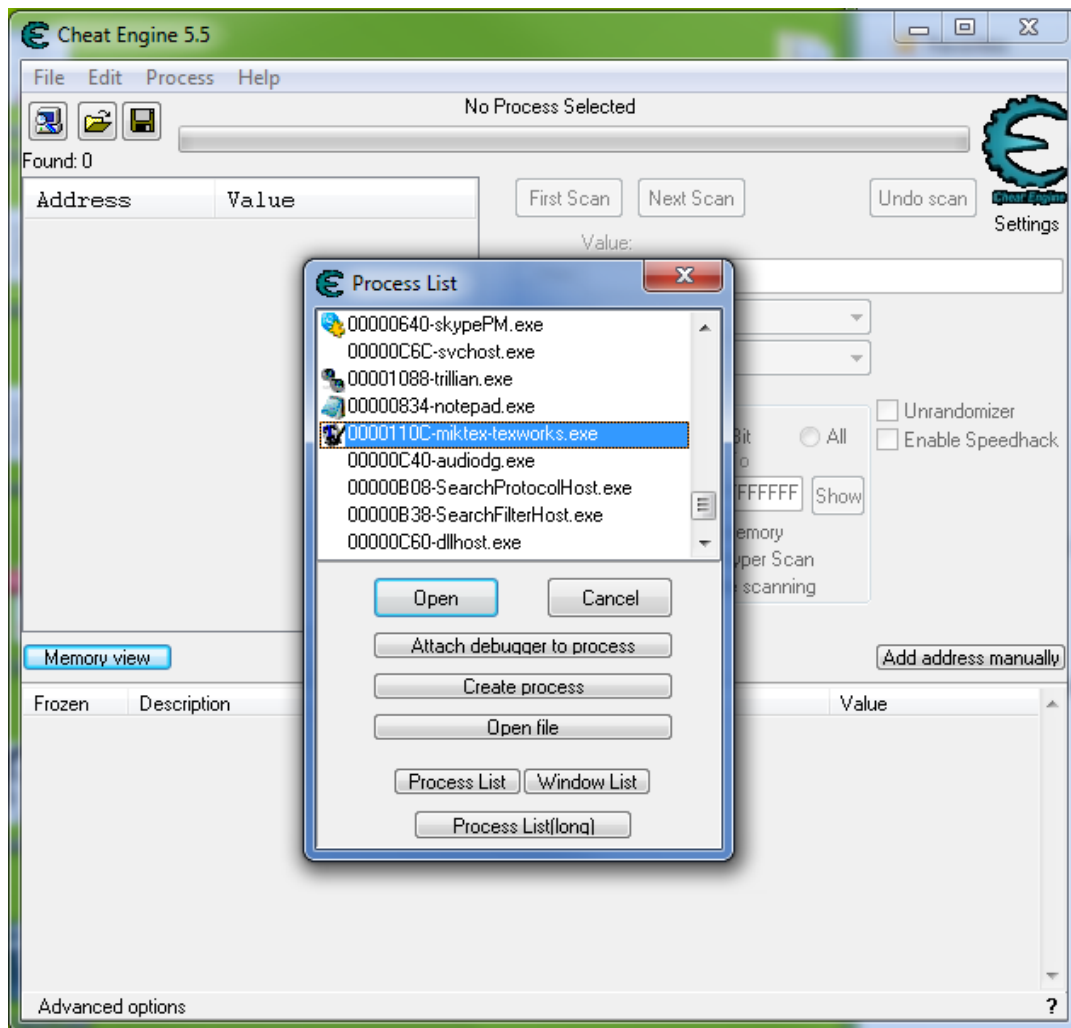


Figure A.3: Screenshot of memory editor and disassembler, Cheat Engine 5.5

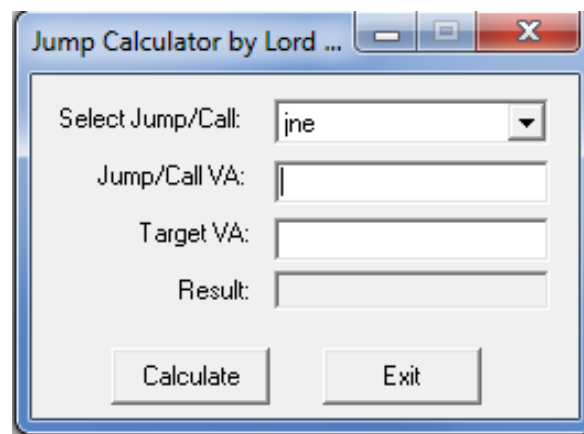


Figure A.4: Screenshot of the jump calculator, JumpCalc

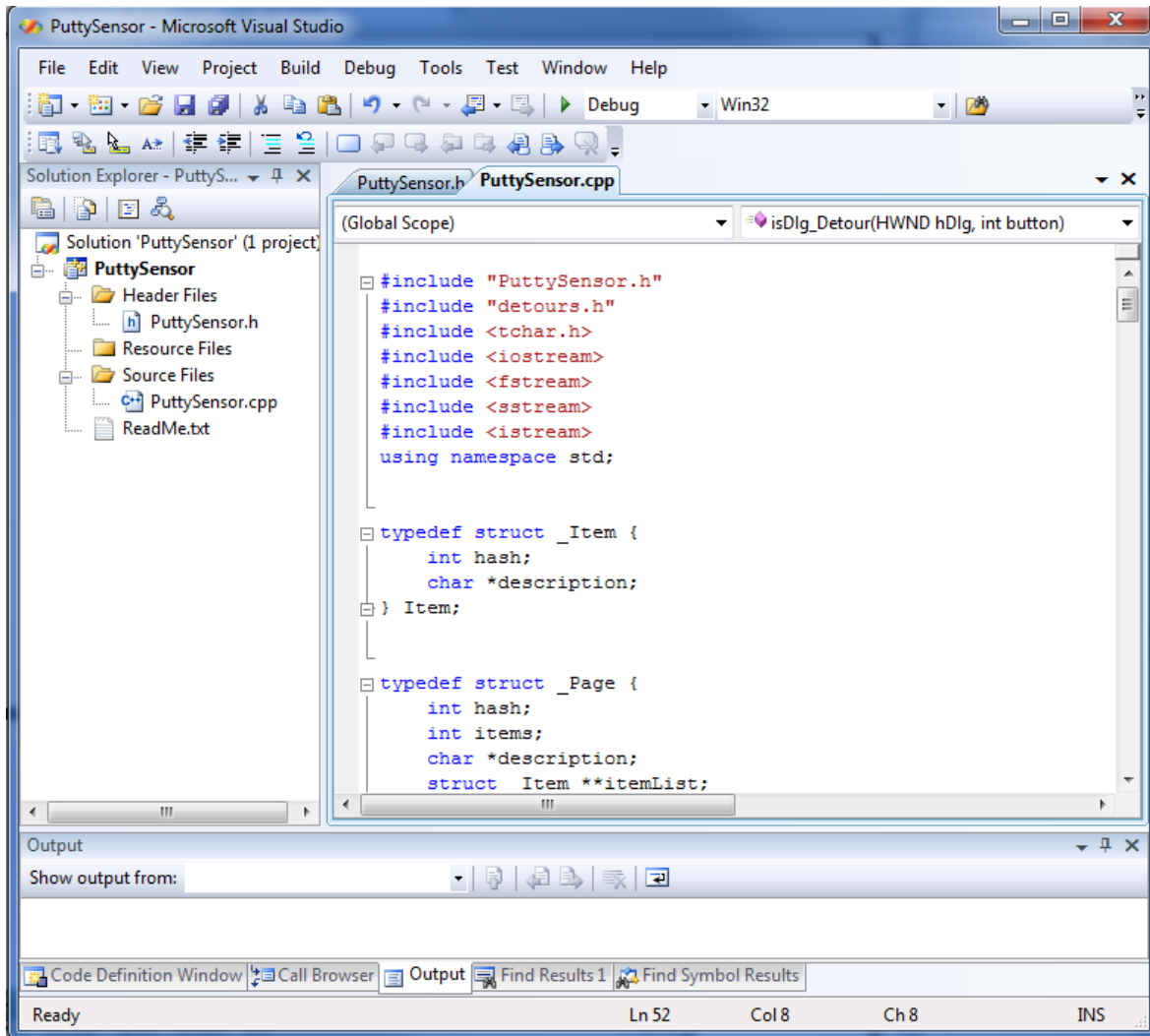


Figure A.5: Screenshot of the compiler, Visual Studio 2008

Vita

Christopher Thomas Rathgeb was born in Lancaster, CA on 13 March 1985. He graduated from Farragut High School in 2002 where he went on to the University of Tennessee in Knoxville, TN. He recieved his Bachelor of Science in Computer Engineering in August 2007. Shortly after graduating with his Bachelor's degree he started his own software company, Zippz Software, LLC. His company focuses on game solutions for personal computers and mobile devices. In August 2008, he returned to UTK where he is currently pursuing his Masters of Science degree in Computer Engineering. He intends to continue his education towards a Ph.D. Christopher's interests include video games, reverse engineering, mobile devices, and computer security.