A Unique Examination of the Buffer Overflow Condition

by

Terry Bruce Gillette

Bachelor of Science
Ocean Engineering
Florida Institute of Technology
1984

A thesis submitted to the
College of Engineering at
Florida Institute of Technology
in partial fulfillment of the requirement for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
May, 2002

We the undersigned committee hereby recommend
that the attached document be accepted as fulfilling in
part the requirements for the degree of
Master of Science of Computer Science.

"A Unique Examination of the Buffer Overflow Condition"

a thesis written by Terry Bruce Gillette

James A. Whittaker, Ph.D.
Professor and Director of the Center for Software Engineering Research
Committee Chairperson

Alan A. Jorgensen, Ph.D.
Senior Research Scientist, Computer Science
Committee Member

Fredric M. Ham, Ph.D.
Harris Professor, Electrical Engineering
Committee Member

William D. Shoaff, Ph.D.
Associate Professor and Department Head
Computer Sciences

Abstract

Title: A Unique Examination of the Buffer Overflow Condition
Author: Gillette, Terry Bruce. (M. S., Computer Science)
Committee Chair: James A. Whittaker, Ph.D.

Buffer overflows have been the most common form of security vulnerability for the last ten years. More over, buffer overflow vulnerabilities enable the type of exploits that dominate remote network penetration. As our reliance on commercial third party software is critical in the current computing environment one must consider the question of how these vulnerabilities are discovered in released proprietary software.

This thesis presents research focused on the fundamental issues surrounding the buffer overflow vulnerability. The objective is to analyze and understand the technical nature of this type of vulnerability and, on the basis of this, develop an efficient generic method that can improve the detection of this software flaw in released, proprietary software systems. The work is performed from the perspective of a security auditor searching for a single vulnerability in a released program, a different approach compared to the many previous studies that focus on both static source code analysis and run time fault injection. First, for systems that include commercial off-the-shelf software components, we perform a systematic review of buffer overflow exploit data and develop a classification hierarchy. The goal of this new taxonomy is to provide a tool to assist the auditor in developing the heuristic elements for exploratory testing. Second, we propose that a signature analysis of a disassembled binary executable can lead to the discovery of a buffer overflow vulnerability. In support of this argument we demonstrate a methodology that can be used on closed source proprietary software where only the executable binary image is available. In this case, the key selling point is not the potential rapid automated detection of a buffer overflow vulnerability but the proof of concept that security flaws can be detected by binary scanning techniques.

Table of Contents

Table of Contents (Continued)

Table of Contents (Continued)

Table of Contents (Continued)

List of Figures

List of Tables

List of Exhibits

Dedication

Staring at the back of someone's head for (6) months while the grass grows and the baby cries can be quite an ordeal especially when ones marriage included the promise of a partnership. For this reason, the following work is dedicated to my wife Pam without whose love, understanding and patience this would not have been possible.

Acknowledgments

When the thesis is finally written and sealed, the days of agonizing over half-baked ideas, struggling through writer's block, and worrying about when (and if) all this is going to end all seem rather distant. The many people who helped me in this long and sometimes frustrating process deserve special thanks, and they are many.

During my graduate studies I have received outstanding support by the Lockheed Martin Corporation. Without their sponsorship this advanced degree would have not been possible. In particular, management has advocated and endorsed the pursuit of further education. As my direct manager, I have had the great fortune to benefit from the mentoring Ben Dusenbery with whose guidance and encouragement made this work a reality.

Chapter 1

Introduction

Basic research is what I'm doing when I don't know what I am doing.

- *Wernher von Braun*

This thesis explores the buffer overflow vulnerability as it exists in today's modern information systems. We will then extend this knowledge by the design, implementation, and analysis of a novel approach to the discovery of potential buffer overflow vulnerabilities in closed source, proprietary software.

1.1 Motivation

Buffer overflows have been causing serious security problems for decades. In the last few years the underlying cause of the majority of computer system and network exploits and vulnerabilities have been the buffer overflow condition. As such, this represents or should represent a top security concern for all entities associated with information security. Broadly speaking the buffer overflow can affect any system where a static amount of space has been allocated for undefined dynamic input. The underlying architecture of modern computer systems makes all data handling processes susceptible to this condition. In the past the buffer overflow was treated as a software bug that would, at the very worst, manifest itself as a nuisance if it were to cause a running process to crash. With the arrival of time-sharing systems, buffer overflows became an intellectual curiosity as a means to seize control of a machine in a laboratory setting.

The advent of computer networks has given rise to new computational environments and computational models. Remote execution, distributed computing, and code mobility are no longer constrained to the research environment. These modern computational models bring great flexibility and new promises to our everyday world of information technology. However, accompanying the expanded potential comes a set of security implications that were not present when computation was carried out largely on local, stand-alone machines. The buffer overflow represents perhaps the most insidious example of an emergent security threat that scaled directly from a stand-alone problem to one of global significance.

Underlying architecture make all data handling processes susceptible to this condition. The key to the buffer overflow exploit is its ability to allow for the execution of arbitrary code. In the Age of Information and subsequent information warfare, the buffer overflow is analogous to the missile. Like its coun-

terpart it can be used to leverage a tactical or strategic advantage across any information system in place today. In this thesis we concentrate on the possibility of buffer overflow detection at run time using the unique approach of binary disassembly.

It is a fact that buffer overflows are caused by the poor implementation of high level programming languages during the software development process. Today, the vast majority of software executing on both commercial and government systems is untrusted, commercial off the shelf software. Unlike custom applications, which may in their specification include explicitly addressed security measures, documented third party source audits and rigorous testing, commercial software walks an often indistinct line between what is secure enough and what will be profitable. When one considers security-hardened software, it is usually in the context of the expense that would be incurred if it were to fail or if it were to be exploited. The development costs of these custom applications are usually in direct proportion to the potential liability that their failure or exploitation would cost. Consequently, this class of software is well out of reach of the average consumer, available only to the deep pockets that governments or big business can bring to the table. Even this does not guarantee error free, nonexploitable code. Widely published examples of very expensive software gone wrong include the Ariane IV disaster, the failure of a 1 billion US military space mission, and loss of life caused by a software controller for medical radiological equipment to name a few. This class of software represents the best money can buy and illustrates in dramatic fashion the serious challenges to complex software systems.

With the development of and widespread use of high level programming languages such as C and C++, dominant PC architectures, and dominant Operating Systems, the move has been away from heterogeneous custom software where a single customer funds development and is able to controls its destiny. Today's software is available to homogeneous, global audience and is driven by a market economy. Competitive market forces coupled with profitability margins serve to dictate when software is released to the masses. In addition, because of its commercial nature, this software is a proprietary product that is, in effect, secret from the very users who purchase it. The security risks inherent in using third party proprietary software are extremely important because today’s information systems are being built from increasing amounts of reused and prepackaged code. A huge portion of the global information infrastructure has been scaled up on this type of software.

The security analysis of complex software systems has always been and continues to be a serious challenge with many unanswered research issues. Unfortunately, third party software serves only to complicate matters. Code that is acquired from a vendor and delivered as an executable file with no source code available makes some traditional analyses impossible. To offset the costs of rigorous testing, vendors rely on the information hiding associated with an executable binary to provide a large portion of a programs security. The upshot is, that relying on today's third party software systems to ensure security is a

risky proposition. This is especially true when such systems are designed to work over a network with global extent. In fact, we have already witnessed the failure of this paradigm.

Current empirical evidence demonstrates that most external security violations are made possible by flaws in software and the buffer overflow ranks as the most significant [73]. The key suppliers of operating systems, firewalls, and web-based applications invest considerable effort to find these types of flaws. It is not just good enough that they find one or a few, they must find them all which is a demonstrated impossibility with large complex programs. Once released into the real world this software is subjected to an assault of global proportions as hackers both good and bad rush to exploit it. They are not limited by release schedules, timelines and deadlines. Their only limitation is their ability and more significantly, they do not have to find every flaw....they only have to find one.

1.2 Problem Statement and Our Approach

The central problem we study in this thesis can be stated in summary as:

> *We seek a technique to rapidly find an instance of a potential buffer overflow vulnerability in an executable binary. This technique will be scalable to commercial third party software systems.*

Static Source Code Analysis is proposed by University of Virginia computer science researchers as one such approach and focuses on the source structure, syntax and procedural design of the code [74]. This approach encompasses the use of search algorithms to find faulty coding constructs that could lead to a potential buffer overflow. The effectiveness and accuracy of the methods associated with source code analysis can be and have been well studied and are in fact well documented. Established techniques range from line-by-line hand auditing to automated procedures. The fact is, that despite their use in today's development environment, programming errors that lead to buffer overflows remain.

Run-Time Analysis or Black-Box Testing is another approach that is related to the input/output domain of a particular program. This well described class of techniques, usually performed at run-time, is highly dependent on the actual program input. Punishing automated testing tools [75] have been developed and are currently used in the software development process however buffer overflows continue to appear.

Far from being two discrete means of testing which operate on parallel paths, these two approaches are usually combined to form a third unified method to provide a more complete orthogonal coverage of the software. These methods, when broadly considered, represent the testing process of software in general. They are specifically used during the development cycle where access to the source code

is an essential element in developing the overall testing methodology. Ongoing research associated with these techniques is moving in the direction of capturing all errors in the development cycle.

As it only takes one undiscovered exploitable buffer overflow to compromise a released program we do not care about finding them all, we only concern ourselves with finding one. Thus the central question we pose in this thesis is: "'Is there a technique that one can use to find an incidence of a buffer overflow vulnerability, with accuracy and efficiency, when applied to a commercial third party software?" The research described here is an attempt to provide such a technique. Through a detailed examination of the buffer overflow phenomenon we develop a technique called binary scanning, and demonstrate its effectiveness for finding a single buffer overflow accurately and efficiently. Binary scanning is a novel method for examining a disassembled binary executable for a unique signature related to a buffer overflow vulnerability. Our approach will investigate high-level code constructs that result in a buffer overflow. These constructs will then be compiled into a binary then disassembled and examined for signatures in the assembly code. Our ultimate goal will be to develop an algorithm that can search for and discover these signatures. We note with interest that this general binary scanning approach yields its results in a manner similar to the way released software is exploited. That is, a single vulnerability is discovered as quickly as possible.

1.3 Brief Summary of Results and Contributions

We proposed binary scanning as a novel approach to locate potential buffer overflow vulnerabilities in proprietary software. Here we briefly summarize the results and contributions resulting from our research:

- Several binary scanning strategies have been identified and specific schemes have been developed. A substantial number of systematic empirical evaluations with the developed scanning algorithm have been performed.
- The binary scanning strategies do show a linear scalability. Our studies show that the algorithm that we have developed work from modest code constructs to small commercially released software. We propose then demonstrate that this scalability is linear right up through enterprise class software applications.
- The binary scanning strategies can outperform the other more common software testing techniques for finding a single instance of a buffer overflow vulnerability with certain unique characteristics. In the commercial software domains we studied, one can only assume that some level of testing was performed prior to release. Our results show that the binary scanning approach is a valid one.

1.4 Organization of the Thesis

In Chapter 2 we provide a topical overview of the buffer overflow exploit. To give balance to our investigations we explore the literature for the motivation behind exploits in general. To provide a temporal understanding, a chronology of the buffer overflow problem is presented.

In Chapter 3 we describe a taxonomy. The taxonomy provides a useful structure in which to gather a wide diversity of vulnerabilities into a hierarchy of categories. The strategies behind the classification scheme and conclusions resulting from the taxonomy are discussed.

To evaluate our proposed schemes and techniques, we performed a substantial review of the C and C++ programming language which is summarized in Chapter 4. Particular consideration was given to the standard library in the presentation of known dangerous functions. We extend this research by developing code constructs with known buffer overflow conditions to be used to baseline our vulnerability signatures and act as a control during testing. The control group program and methodology used in the design is described in Chapter 6.

In Chapter 5 we present how programs behave at compile time with a discussion of binary disassembly and at run-time in a Win32 environment with a discussion of the PE File format. In addition, the difficulties of reverse engineering and the legal implications of commercial product disassembly are summarized.

Using a baseline group of potential string primitives identified in Chapter 4, we will empirically derive possible compile-time signatures for use in a technique of binary scanning. Once a signature has been identified a binary scanning algorithm can be developed. Chapter 6 leads the reader through the configuration of our test apparatus, algorithm development. Chapter 6 concludes with several commercial proprietary binary files that are disassembled and scanned for a potential buffer overflow vulnerability.

In Chapter 7 we conclude with a summary of our work and provide an evaluation as to the validity of the approach. In addition, we formulate pointers to potential future work as an extension of this technique.

Chapter 2

A History of Buffer Overflow Vulnerabilities

All of physics is either impossible or trivial. It is impossible until you understand it, and then it becomes trivial.

- Ernest Rutherford

2.1 An Introduction to the Topic

As we enter the so-called "information age" of global networks, ubiquitous computing devices, and electronic commerce, many businesses, consumers, and other users are becoming increasingly concerned about computer security. Yet the current state of computer security is lamentably poor in practice. One survey found that nearly 2/3 of Internet hosts are vulnerable to unsophisticated, well known, easy-to-exploit attacks [1], and the number vulnerable to more clever or more recent attacks is presumably greater still. Recent FBI studies have shown that business losses are large and increasing [2]. Anecdotally, break-ins are rampant, and hackers discover new vulnerabilities almost every day. There are several causes for these trends.

First, today's systems rely heavily on applications built in an age when security did not receive the same attention it does today. The Internet, once populated almost exclusively by cooperating researchers, and local-area networks, populated exclusively by local co-workers, spawned many legacy applications that were originally designed for use only in a friendly environment. This legacy code is unlikely to go away anytime soon, yet the threats it must defend against have changed dramatically: global networks now expose us to a much broader array of adverse interests, including hackers, vandals, competitors, criminals and other untrustworthy entities. Nonetheless, we need someway to protect our data from a growing number of unfriendly and potentially malicious computer users even as we continue to rely on irreplaceable legacy applications. This leaves us in an exposed position with no obvious solution.

A second contributing factor is that building secure systems is fundamentally hard, and today's programming environments do not make the task any easier; in fact, they often make the task significantly harder.

- The operating system does not provide any way for applications to specify just the subset of privileges they actually need, and as a result when an application is compromised, the intruder typically obtains full access to the entire system.

- The Unix system-call interface does not provide any simple way to atomically query a system object for permission information before performing some operation on that object, and as a result exploitable race conditions are common.
- The language almost invariably, C does not assure memory safety, and as a result it may be easy to cause memory errors and force a security-critical application to crash or worse, execute arbitrary code.
- The standard C libraries provide string buffer management primitives that are unsafe, and as a result our programs fall prey to buffer overrun vulnerabilities.

In all cases, it is in principle possible to code the desired functionality in a secure way, but the secure way is often not the easiest, best-supported, most convenient, most portable, or standard way, and it is not always feasible to fix problematic aspects of the programming environment. Of course, when the standard library functions have security pitfalls, it becomes too easy for developers to inadvertently introduce security holes where these functions are used. In addition, modern programming practice includes the reuse of existing libraries. A symptom of this problem may be recognized in the frequency with which programs are found to contain the same mistakes again and again.

A third common cause of insecurity is that many of our security-critical applications are large and complicated. Complexity breeds subtle interactions, subtle bugs, and thus subtle security holes [3]. Moreover, sheer size can make the source code so unwieldy that it becomes very difficult to review the application for potential security errors, or even to understand exactly what it is doing. As a result, large, complex applications often go largely unscrutinized, despite their security-critical nature. One symptom of this phenomenon can be found in the number of vulnerabilities that have lain dormant in poorly reviewed source code for years before being discovered by the security community.

In essence, then, we have a software quality assurance problem with serious implications for computer security: How do we deal with the fact that our most trusted software, even our security software itself, often contains security vulnerabilities? Re-design and reimplementation of all security-critical code of questionable quality does not appear to be a viable option at the moment; it is simply too costly to be a general solution. As a consequence, we are stuck with an overwhelming amount of legacy and other code of questionable security. As a consumer of this software with no access to the source code we have no good way to check it for even the simplest, most common classes of coding errors.

At this point, some readers might suspect that the answer to these woes lies with the vendor to perform a careful, manual review of all security-critical code prior to release. Indeed, we would certainly agree that code inspection is essential to security. However, the root problem is that manual inspection is extremely time-consuming, and the amount of legacy code that would require auditing is enormous. As a result, code review is not applied as often or as thoroughly as it is needed. A second problem is that the sort of common, mindless mistakes that are easy for a programmer to make are also typically easy for a

reviewer to overlook, and thus we might hope for an even higher level of assurance than manual code review can provide by itself. These comments suggest that we might do well to look for tools to help automate the process, to reduce the burden on the reviewer, to raise the assurance level, and in general to reduce the cost of the security quality assurance process.

We have argued that assurance for security software is an important unsolved problem, and that simple programming errors account for a surprisingly large proportion of security failures. Critical to this entire discussion is the fact that a person looking to exploit a particular application needs to find only a single instance of a vulnerability. Perhaps the best illustration of this phenomenon may be found in the buffer overrun vulnerability, a variant of an array bounds violation error that forms one of the most prominent causes of insecurity in modern software systems [4]. In this chapter we discuss the buffer overflow paradigm and how it has manifested itself in the modern global software environment.

2.2 Motivation

Knowledge of the Buffer overflow condition is nothing new. In general, the theory surrounding buffer overflows has been part of computer science since the beginning of digital data processing. Our overwhelming use of the von Neumann architecture, where both data and programs are stored in the same memory space [78], makes the possibility of a buffer overflow a constant. This is especially true when using programming languages that do not perform bounds checking at compile time. In the past, on a single stand alone machine, the buffer overflow represented at worst an annoyance. With the advent of time-share systems beginning in the early 70's, the exploitation of the buffer overflow condition became a security concern [77] as it could, in theory, be used to escalate system privilege. As the time-sharing paradigm of the 1970's moved towards what we now know as computer networks the security implications of buffer overflows grew dramatically. Now we could not only comprise our own machine, we could remotely gain access on one belonging to someone else. We could use the buffer overflow condition to execute the code of our choosing. As the size and diversity of the Internet grew throughout the 1980's, speculation increased that a major security flaw would be exploited and as a result the Internet directly attacked. From 1986 to 1987 the size of the Internet grew almost 600 percent [76] and time was running out from the reality of a widespread exploit crashing the Internet.

On November 3, 1988 on what has come to be called the Black Thursday event [5], system administrators around the country came to work on that day only to find that their networks of computers were laboring under a huge load. If they were able to log in and generate a system status listing, they saw what appeared to be dozens or hundreds of “shell” (command interpreter) processes. If they tried to kill the processes, they found that new processes appeared faster than they could kill them. Rebooting the computer seemed to have no effect as within minutes after starting up again, these mysterious processes

overloaded the machine. A worm had invaded these systems. The worm had taken advantage of lapses in security on systems that were running *4.2 or 4.3 BSD UNIX* or derivatives like the SunOS. These security flaws allowed it to connect to machines across a network, bypass their login authentication, copy itself and then proceed to attack still more machines. The massive system load was generated by multitudes of worms trying to propagate the epidemic [5]. D. Bruce, MIT EECS Professor and Vice President for Information Systems estimated that *approximately 10%* of 60,000 Internet hosts were exploited [6].

One of the methods used to gain access to these systems was a buffer overflow exploit of the Unix service "finger". This hack involved co-opting the TCP finger service as a method to gain entry into a system and it represented the first well documented, widespread buffer overflow exploit. The *Finger* service reports information about a user on a host, usually including things like the user's first and last name, the location of their office, the number of their phone extension and so on. The Berkeley version of the finger server has been characterized as a really trivial program. It reads a request from the originating client, stores that information in a 512 byte buffer on the host machine, runs the local finger program with the request as an argument then ships the output back to the client. Unfortunately the finger server reads the remote request with `gets()`, a notoriously dangerous function. This is a standard C library routine that dates from the beginning of the language. The function has no parameter in which to perform bounds checking and therefore does not check for overflow of the server's 512 byte request buffer on the stack. The exploit is some VAX machine code that asks the system to execute the command interpreter `sh` supplied by the client as a 536 byte request to the finger server. This request is crafted to be specifically 24 bytes larger than the 512 byte buffer. This is just enough data to write over the server's stack frame for the main routine. When the main routine of the finger server exits, the program counter of the calling function should be restored from the stack, but the exploit wrote over this program counter with one that points to the VAX code in the request buffer. The program then jumps to the worm's code, part of the request, and runs the command interpreter, which the worm uses as a method to enter its bootstrap; a classic case of overwriting the stack frame pointer.

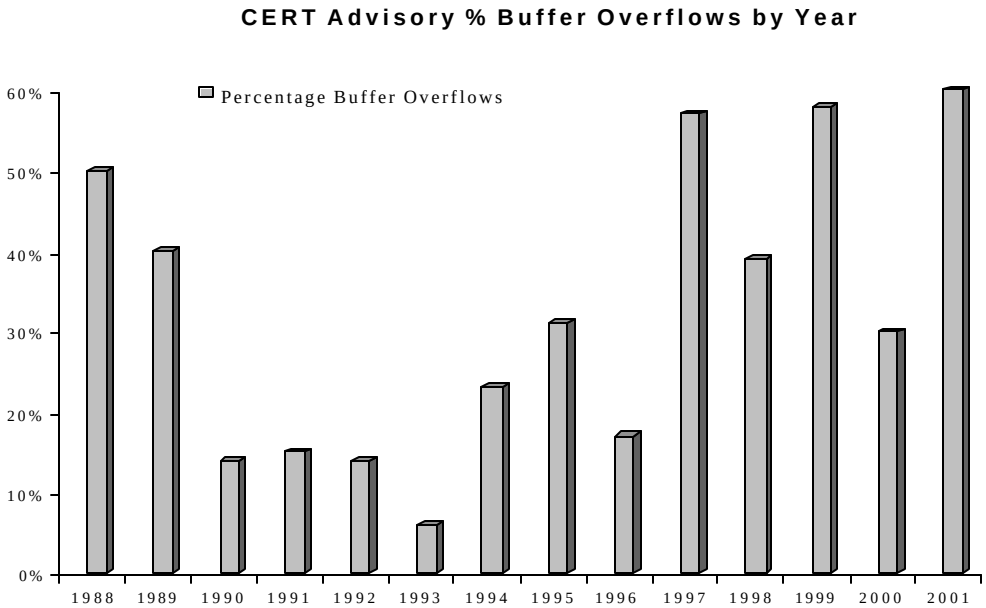
Shortly after the worm was analyzed and reported to use this feature of `gets()`, patches were released that replaced all instances of `gets()` in system code with code that maintained parameters against the length of the buffer. The danger inherent in `gets()` was so great that some libraries were modified by the complete removal of the `gets()` function. It is questionable why the function is mandated by the ANSI C standard and the answer must be backwards compatibility. This in itself speaks volumes of it's widespread usage. Although no documented reports associated with the finger server bug exist before the worm incident, in May 1988, students at UC Santa Cruz apparently penetrated security by exploiting a different finger server with a similar bug. The system administrator sent mail to Berkeley, but the seriousness of the problem was not appreciated as a major issue at the time.

Buffer overflow attacks remained relatively unheard of for many years following the Worm. One known example came in November of 1994, when one of the first commercial Web servers, running HP-UX, was successfully breached using a buffer overflow attack against the National Center for Supercomputing Applications (NCSA) 1.3 Web server [79]. As this Web server sat on the target's internal network and could be connected to through the firewall, the attackers had unfettered access to the victim's internal network.

The event that really fueled the frequency of attacks was the November 1996 publication of a paper entitled "Smashing the Stack for Fun and Profit", by Aleph One, in the on-line hacker magazine Phrack [7]. Aleph One's paper (itself based on a paper written by Mudge of the L0pht, an independent computer security think tank specializing in Windows NT) explains in detail how to write a buffer overflow exploit against a Unix system program. This moved the technical skills required from the graduate level down to anyone who could follow directions well. This heralded the birth of the script kiddie.

As a result in 1997 and 1998, buffer overflow exploits became extremely common, mainly targeting Unix systems, in particular the Open Source versions. While the Open Source organizations, like the various Linux distributors or FreeBSD, were quick to release patches, the number of exploits was astounding. To get a feel for the scope of this problem, an exact string search for “buffer overflow exploits” on Google.com returned 6,000 matches. Buffer overflow exploits continued right into 1999, 2000, 2001, and show no signs of going away anytime soon. If anything, the incidence of buffer overrun attacks has been increasing. See Figure 1 for data extracted from CERT advisories over the last decade.

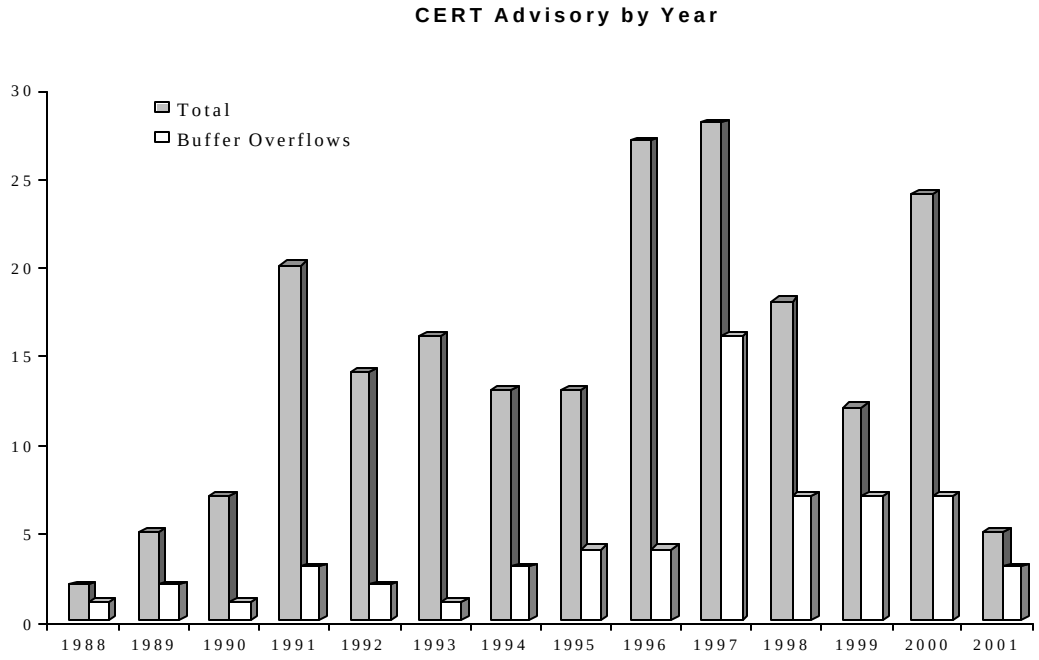
Figure 1: Frequency of buffer overrun vulnerabilities



Derived from a classification of CERT advisories. The chart shows, for each year, the total number of CERT-reported vulnerabilities and the number that can be blamed primarily on buffer overruns

Figure 2 shows that buffer overruns account for up to 50% of today's vulnerabilities, and this ratio seems to be increasing over time. A partial examination of other sources suggests that this estimate is probably not too far off: buffer overruns account for 27% (55 of 207) of the entries in one vulnerability database [8] and for 23% (43 of 189) in another database [9]. Finally, a detailed examination of three months of the bugtraq archives (January to March, 1998) shows that 29% (34 of 117) of the vulnerabilities reported are due to buffer overrun bugs [10].

Figure 2: Frequency of buffer overrun vulnerabilities by percentage



Derived from a classification of CERT advisories. The chart shows, for each year, percentage of CERT-reported vulnerabilities that were due to buffer overruns for each yea

The only big change in the original problem is that it now includes Microsoft products. One reason why Microsoft products were not initially attacked is that the techniques required are somewhat different and more importantly the code is closed source. This represents an especially alarming situation as the majority of enterprise wide information solutions are based on Microsoft products and associated third party applications and add-ons.

Windows 2000 has already patched several buffer overflow vulnerabilities in it's enterprise class of Operating System products with a least two being reported as a major security issues. The first, a buffer

overflow in Windows 2000 indexing service[1]. The Index Server is the built-in search engine in Windows 2000 that catalogues and indexes files and properties of the hard drive. Improper bounds checking on the input buffers on the DLL file (%system32\idq.dll) allows additional characters to be forced into the process space, overflowing the buffer and providing memory space for shell code insertion. As with all buffer overflows, the shell code simply requires to launch and bind a command shell to listen on a specific port and the attacker to connect to the port using netcat or telnet. All Windows products from NT4 forward are exposed to the vulnerability. This vulnerability gained worldwide attention when the "Code Red" worm exploited it. It was estimated that 359,000 hosts were exploited within the first 14 hours of its initial release [11].

The second, a buffer overflow in a FrontPage server extension[2]. The FrontPage extensions ship with IIS4 and IIS5, Office 2000 and Office XP, and extend the functionality of the IIS web server to support components used in the Visual Studio development suite. An optional feature of the FrontPage extensions is Visual Studio, Remote Application Deployment (RAD) component that contains an unchecked buffer vulnerability. The RAD feature allows developers to deploy custom COM components by allowing authenticated authors to upload COM components onto the server.

The unchecked buffer in the request processing routine allows a malformed command to insert shell code into the FrontPage process space, yielding access at either *IUSR_<hostname>* or system-level privileges. The buffer overflow occurs if fp30reg.dll receives a URL request that is longer than 258 bytes, exposed through a lack of length checking on the input string. By exploiting this vulnerability successfully, an attacker can execute code with the privileges of *IUSR_machinename* and under certain circumstances with the privileges of system. Visual Studio RAD component is not selected by default in the installation options, as is actively alerted as not suitable for production systems during installation if selected.

Clearly we have demonstrated that the buffer overflow problem is a major security issue. Despite a precise understanding for the last 15 years of the severity of the problem and of the programming practices that lead to this condition, it has not gone away. Now that we have developed a feel for the scope of situation we will turn our attention to those persons who develop and use these exploitation techniques and to investigate their potential motivations.

2.2.1 Benevolent Hacking: The White Hat

In the beginning it was a core group of singular minded individuals who programmed operating systems, examined core dumps, and basically spent their entire waking moments engrossed in the intrica-

1. Reference Case #10 in appendix C.
2. Reference Case #7 in appendix C.

cies of digital computation. It was this original group who earned the moniker “Hackers”. These were the pioneers of modern day computing. Hacking was originally characterized to convey the sense of `an appropriate application of ingenuity'. Whether the result was a quick-and-dirty patchwork job or a carefully crafted work of art, it was the cleverness that went into it that determined the hack. Somewhere along the lines the definition became confused and assumed the darker connotation that remains to this day. In order to distinguish those who create these clever hacks the terms “White Hat” and “Black Hat” are being used with the inference being obvious.

The white hat [80] has become the term for people who hack legitimately. These hackers devote their careers to discovering software vulnerabilities and then post these discoveries to Internet list servers or their own security related Internet homepages. Current motivation follows that by publicly posting vulnerabilities these hackers are forcing software companies to not only address these newly found problems but to also fix them. This is the paradigm of attacking a system to secure it. Other white-hat hackers discover operating system vulnerabilities, i.e. Linux, and email their results to kernel developers who then write and post software patches to mailing lists devoted to system vulnerabilities and Internet web sites like www.slashdot.org.

The White Hat's operate on the premise that so called old-fashioned security controls such as firewalls and intrusion-detection systems aren't enough. Their reasoning is that one must embrace the methods and mind-set of the enemy. In the past, the White Hats were loosely aligned in ad hoc groups or operated independently with the belief that the full disclosure of vulnerabilities was the quintessential means of securing computer systems. Their success was closely aligned with the rapid growth of the computer and network security industry. White-hat hackers have gone to become security consultants and are now associated with computer security consulting organizations, like Foundstone, @stake Research Labs and on-line security forums like bugtraq, www.securityfocus.com. Through their efforts, the software vendors learned that security issues were important. They also learned that they were very expensive to implement.

As computer security continues to become big business, a $21 billion industry by 2005 according to a International Data Corporation report, software vendors such as Microsoft seek to influence the past practice of full vulnerability disclosure through close association with these new companies. This is setting the stage for a new type of White Hat hacking as some view this as an intrusion by large corporations not interested in the time, effort and expense of true software security. The current debate centers on this new relationship between software vendor and security organization and the potential conflicts of interest. Some view this as being a less expensive, public relations alternative to developing solid secure code. One thing is sure to remain; third party software will continue to be under pressure from the moment of release by those who align themselves with the White Hat approach.

2.2.2 Malicious Hacking: The Black Hat

The Black Hat [81] has become the term for people who hack at the bounds of what is legal to those who are clearly breaking the law. The motivations common to those who would commit or attempt to commit computer-related crime are diverse, but hardly new. In general, criminals are driven by time-honored motivations, the most obvious of which are power, greed, revenge, lust, adventure, and even the desire to taste “forbidden fruit”. Computer crime makes no distinction to this general classification with computer criminals being no different. The ability to make an impact on large global systems from a remote location may, as an act of power, be gratifying in and of itself. The desire to inflict damage or loss on another may have roots in revenge as a motive, as when a disgruntled employee shuts down an employer's computer network, or to ideology, as when one defaces the web site of an organization or institution that one regards as against their beliefs or as abhorrent. Current activity on the so called ‘electronic frontier’ entails an element of adventure, the exploration of the unknown. This is especially true for the disaffected youth raised in the age of information. The very fact that some activities in cyberspace are illegal or likely to elicit official condemnation is enough to attract the rebellious, defiant, or the irresistibly curious. In some cases, given the degree of technical competence and skill required to commit many computer-related crimes, there is a unique motivational dimension worth mentioning here. This is, of course, the intellectual challenge of mastering a system with a high level of complexity.

Unauthorized hacking is a felony crime in the United States and many other countries. Clearly there exists an ongoing information war with the Black Hat seen as the enemy. The profile of the Black Hat runs from the 12 year old script kiddie with the time and persistence to get automated well documented exploit scripts to work to the government sanctioned hacker a well funded computer expert working for an intelligence organization with military objectives. For the Black Hat, the buffer overflow is of critical importance as it is one of the primary means of obtaining root access on a system. As the profile of the Black Hat is so wide we will categorize the malicious hacker, by stereotype, into three groups.

The Script Kiddie: The script kiddie practices hacking using scripts and programs written by others, often without an understanding of the exploit they are using. These are the hackers with limited technical expertise who, using easy-to-operate, pre-configured, and/or automated tools; conduct disruptive activities against networked systems. There are about 30,000 hacker-oriented sites on the Internet, bringing hacking and all the associated tools within the reach of anyone who may have an interest. As an example the Rootshell web site has a database of 690 exploit scripts [12]. Fyodor's Playhouse contains 383 attacks [13] and the Legacy hacking site has 556 exploits [14].

The malicious activity caused by the script kiddie is usually limited to Web Site defacement with the motivation being the social status that this type of conquest brings. Script kiddies can work alone or in

loosely knit ad hoc groups and often communicate using Internet Relay Chat (IRC) channels. For these practitioners their hacking is a social activity and Internet Relay Chat their communications link.

The script kiddies are not out for specific information or targeting a specific company. Their goal is to gain root the easiest way possible. This is accomplished by focusing on a small number of well-documented exploits, and then using automated scanning techniques, search the entire Internet for that vulnerability. Some of the more advanced users may use more specialized tools to leave behind sophisticated backdoors. Most have no idea what they are doing and only know how enter rudimentary input the command prompt. The common strategy is to randomly search for a specific weakness, and then exploit that weakness. It is this random selection of targets along with their large numbers that make the script kiddie such a dangerous threat.

Most of the tools available to the script kiddie are easy to use, widely distributed and well-documented allowing use by anyone. The script kiddie methodology is a simple one. Scan huge tracts of Internet address space for a specific weakness, when that weakness is found, it is exploited. The most coveted remote exploits that are in use by the script kiddie are the ones associated with the buffer overflow.

Mafia Hackers: Russian hackers first captured the world's imagination in 1994, when a young mathematician, Vladimir Levin, hacked into the computers of Citibank and transferred $12 million to the bank accounts of his friends around the world. Levin was arrested, but his case inspired other hackers, for example, Ilya Hoffman, a talented viola student at the Moscow Conservatory, who was detained in 1998 on charges of stealing $97,000 over the Internet. Another group of Russians stole more than $630,000 by hacking into Internet retailers and grabbing credit card numbers [82]. Numbers this large certainly attract the attention of organized crime and hacking associated with this element has been traditionally centered in Eastern Europe. Recently the Russian mafia has been implicated in hacks involving the theft of up to one million sets of credit card details.

The FBI, in response to an expanding number of instances in which criminals have targeted major components of information and economic infrastructure systems, has established the National Infrastructure Protection Center (NIPC). Based on FBI investigations, classified sources and other information, the NIPC has observed that there has recently been a dramatic increase in organized hacker activity specifically targeting U.S. systems associated with e-commerce and other Internet-hosted sites. The majority of the intrusions have occurred on networks using the Microsoft Windows NT operating system. In these cases the hackers are exploiting known buffer overflow vulnerabilities to gain unauthorized access and download propriety information [15].

Government Sanctioned Hacking: The U.S. Government Accounting Office estimates that 120 countries or groups have or are developing information warfare systems. There are many reported Pentagon intrusions to have surely come from abroad. The United States acknowledged their involvement in

1998 when it was announced that the CIA was devising a computer application that could attack the infrastructure of other countries. One can only imagine the importance of the buffer overflow as an exploit tool to gain remote access in the development of this classified program.

A series of sophisticated attempts to break into Pentagon computers has continued for more than three years according to a member of the National Security Agency's advisory board. Officials at the Pentagon and NSA have called the intrusions “massive” and said they caused significant disruptions. In the Middle East government supported cyber attacks are becoming more prevalent and more methodical. There is documentation that supports the existence of a coordinated campaign on the pro-Palestinian side to identify vulnerable Israeli sites and gain root access using buffer overflow exploits [16].

With our current good guy versus bad guy, exploit and patch model of computer security unpublished exploits are highly coveted in the hacker subculture and may be considered state secrets within governments. The power of a single remote exploit that is unknown to vendors allows a single person to potentially break into thousands of machines, often times with no recognizable trace of how it was done. Many Intrusion Detection Systems (IDS) will not recognize the fingerprints of these new exploits. The hackers with knowledge of these exploits typically do not deface web pages. This knowledge is held for high dollar gain or as a tool for true information warfare. The last thing a hacker with this type of information wants is to bring attention to themselves or their methods. In some cases unpublished exploits can circulate in the underground for up to a year before being disclosed to the masses. The damage caused by a dozen hackers have such an exploit while actively using it over a one year period is often never reported to the extent that it makes the public media.

With individuals and corporations increasingly relying on software with demonstrated security flaws and the Internet to manage everything from their finances to their personal health records, incidents of malicious hacking continue to increase. More than 7,000 computer security violations were reported in the first three months of 2001, more than in all of 1998, according to the CERT Coordination Center, a security research group at Carnegie-Mellon University in Pittsburgh [62].

Just how do these hackers find the buffer overflow vulnerabilities that lead to system exploits? As the majority of the exploited software is closed soured, third party applications, source code auditing is not an option. Are the buffer overflow conditions found simply by brute force or is there a more methodical step-by-step method that can be used. It is this premise that our thesis will attempt to address as we develop a methodical approach to finding these types of vulnerabilities in closed source applications.

2.3 Chronology of Buffer Overflow Exploits

Hacking has been around for more than a century. While many would assume that hacking is a rather recent phenomenon, the practice has a rather long and varied history. As early as the 1870s, an

activity that we would call hacking today was occurring on the new United States phone system. Since the turn of the century, the meaning of the term hacker has evolved. A subculture has grown up around the practitioners and this has seeped into popular culture over the last two decades. As information technology continues to play an increasingly important role in our society so will the hacker continue to make his presence felt.

We present a general chronology of computer hacking and where appropriate, particular attention is paid to the buffer overflow exploit. How did hacking begin and how was it integrated into the overall development of modern digital computing?

Without the real programmers, computers, computing, networks, the Internet and the buffer overflow would not exist. The history of digital hacking begins with John W. Mauchly and J. Presper Eckert. These two men collaborated on what was to become an icon in computer science. This was one of the first truly digital computers, the ENIAC or Electronic Numerical Integrator and Computer. ENIAC was built in 1946 and was designed with over 17,000 vacuum tubes, 30,000 resisters and covered over 1500 square feet of floor space. Despite it's size it was only capable of 1000 calculations per second. Compare that to the computers of today that are capable of tens of millions of calculations per second. The ENIAC was the first digital machine to be able to perform an "if-then" statement or a "branch conditional statement" [83]. In 1949 Presper and Mauchly also launched the BINAC or Binary Automatic Computer, a computer that stored data using magnetic tape.

The next major milestone in the history of hacking is the UNIVAC computer. The UNIVAC was first commercial computing project commissioned for the US Census Bureau and was another of Mauchly and Presper's efforts. This was the first so-called solid state computer and could handle both alphabetic and numerical information. This development was important in that it represented a significant move toward the miniaturization of the computer. With the UNIVAC, the transistor replaced the vacuum tube and the computer shrunk down to the size of a room.

One of the first organized groups of hackers to be formed emerged from the Massachusetts Institute of Technology in 1961. They were members of MIT's Tech Model Railroad and possessed an obsession with the PDP-1 computer. The Tech Model Railroad Club programmed the PDP-1 to control their complex model railroad track and switches. From these humble beginnings MIT would achieve critical acclaim with a world renowned Artificial Intelligence department. No chronology could be considered as complete without including a brief history of the Unix operating system. Unix has been recognized as the quintessential operating system.

Unix was developed in 1969 at AT&T Bell Labs as part of an effort to build an internal operating system to be integrated into their telephone business. The birth of Unix was based on the need of Dennis Ritchie, Ken Thompson and others to produce a programming environment that could support multiple

users. Within this environment they identified a need for a tree structured file system and easy access to devices from within the OS plus a user level command interpreter. From these specialized needs Unix was conceived. It is important to note that Unix also gave rise to another major milestone in hacker history, the C programming language. With the advent of C, the original Unix kernel was rewritten to make it machine independent, or portable, thus making Unix the OS the choice for academic research. Particularly important from the aspect of this thesis are the function libraries within the C language. When used incorrectly, several standard C functions are directly related to the existence of the buffer overflow vulnerability. The C programming language is by far the worst offender leading to this security issue.

Table 1:
A Chronology of Buffer Overflow Exploits[1]

Year(s)	Event
1940's-1970's	The era of the huge mainframes (Catman).
1946	J. Presper Eckert and John W. Mauchly created one of the first digital, general purpose computers: the Electronic Numerical Integrator and Computer (ENIAC) (Williams).
1949	J. Presper Eckert and John W. Mauchly created the first computer that stored its data on magnetic tape: the Binary Automatic Computer (BINAC) (Williams).
1950's	J. Presper Eckert and John W. Mauchly created the UNIVAC computer, which was the first computer that could handle alphanumeric information. It was also smaller than the previous computers--the first step to making computers smaller (Williams). Some of the first hackers of this time were Peter Deutsch, Bill Gosper, Richard Greenblatt, Tom Knight, and Jerry Sussman. (Brunvand). This was the "Golden Age" of hacking. This was when hackers made some of the largest and most important discoveries on computers. The hackers of this time are respected by the hackers of today because the computers they worked on were so cumbersome and they only had a few tools to help them learn about these machines (Brunvand).
1960's	Hacker culture spread to the general culture as the computers did. Centers of hacker culture had by now spread from MIT to Carnegie Mellon University, and Stanford University. Some of the famous hackers of this time were Ed Fredkin, Brian Reid, Jim Gosling, Brian Kernighan, Dennis Ritchie, and Richard Stallman (Brunvand).
1961	TMRC used the PDP-1, the "first commercially successful computer on the market," [MIT got these computers] to program their model train tracks and switches. Thus began the hackers and the AI Lab (Williams).

Table 1:
A Chronology of Buffer Overflow Exploits [1] (Continued)

Year(s)	Event
1961	MIT began using the computer PDP-1, which were the "first commercially successful computer on the market" (Williams). The Tech Model Railroad Club (TMRC) at MIT, who had moved from programming the complicated wiring and switches of their model trains to programming computers, "adopted the machine as their favorite tech-toy and invented programming tools, slang, and an entire surrounding culture that is still recognizably with us today" (Catman). This group was the first to adopt the term "hacker" (Catman).
1940's-1970's	Era of the mainframes (Catman).
1946	J. Presper Eckert and John W. Mauchly created one of the first digital, general purpose computers: the Electronic Numerical Integrator and Computer (ENIAC) (Williams).
1949	J. Presper Eckert and John W. Mauchly created the first computer that stored its data on magnetic tape: the Binary Automatic Computer (BINAC) (Williams).
1950's	J. Presper Eckert and John W. Mauchly created the UNIVAC computer, which was the first computer that could handle alphanumeric information. It was also smaller than the previous computers--the first step to making computers smaller (Williams). Some of the first hackers of this time were Peter Deutsch, Bill Gosper, Richard Greenblatt, Tom Knight, and Jerry Sussman. (Brunvand). This was the "Golden Age" of hacking. This was when hackers made some of the largest and most important discoveries on computers. The hackers of this time are respected by the hackers of today because the computers they worked on were so cumbersome and they only had a few tools to help them learn about these machines (Brunvand).
1960's	Hacker culture spread to the general culture as the computers did. Centers of hacker culture had by now spread from MIT to Carnegie Mellon University, and Stanford University. Some of the famous hackers of this time were Ed Fredkin, Brian Reid, Jim Gosling, Brian Kernighan, Dennis Ritchie, and Richard Stallman (Brunvand).
1961	TMRC used the PDP-1, the "first commercially successful computer on the market," [MIT got these computers] to program their model train tracks and switches. Thus began the hackers and the AI Lab (Williams).
1961	MIT began using the computer PDP-1, which were the "first commercially successful computer on the market" (Williams). The Tech Model Railroad Club (TMRC) at MIT, who had moved from programming the complicated wiring and switches of their model trains to programming computers, "adopted the machine as their favorite tech-toy and invented programming tools, slang, and an entire surrounding culture that is still recognizably with us today" (Catman). This group was the first to adopt the term "hacker" (Catman).

Table 1:
A Chronology of Buffer Overflow Exploits [1] (Continued)

Year(s)	Event
1962-1969	The Department of Defense's Advanced Research Project Agency (ARPA) creates the network ARPANET. They intended for researchers to use. The first universities to use the ARPANET were Stanford Research Institute, UCLA, UC Santa Barbara, and the University of Utah (PBS).
1969	The creation of the ARPANET. This was the first computer network linking "universities, defense contractors, and research laboratories"Hacker Ken Thompson created the operating system UNIX. Hacker Dennis Ritchie created the programming language C. Both of these creations became popular across most computers, which allowed hackers to use on set of tools to hack into many different machines (Catman).
1972	The InterNetworking Working Group is founded to govern the standards of the developing network. Vinton Cerf is the chairman and is known as a "Father of the Internet" (PBS)
1973	"ARPANET goes international" (PBS).
1974-1981	ARPANET moves away from its research and military beginnings and becomes commercialized (PBS).
1974	"Bolt, Beranek, and Newman opens Telnet, the first commercial version of the ARPANET" (PBS).
1975	First portable computer was marketed (Catman).
1977	Steve Jobs and Steve Wozniak created Apple Computers (Neupart & Munkedal).
1979	The first USENET groups are created by Tom Truscott and Jim Ellis; now people from all over can join discussion groups (PBS).
1980	USENT bulletin board began broadcasting information. USENET was a network of UNIX machines that could talk to each other (Catman).
1981	IBM creates its own personal computer (Neupart & Munkedal).
1982-1987	The ARPANET is recognized as an internet. The language of computers on the Internet, TCP/IP is created (PBS).
1982	UNIX hackers from Berkeley began Sun Microsystems; they put UNIX on less expensive workstations (Catman).
1983	The hacker film War Games was released (Catman).
1984	The hacker magazine 2600: The Hacker Quarterly was created (Neupart & Munkedal).
1984	AT&T made a version of UNIX (Catman).
Mid-1980's	Personal computers make accessing the Internet cheap and easy (PBS).

Table 1:
A Chronology of Buffer Overflow Exploits [1] (Continued)

Year(s)	Event
1986	The "Computer Fraud and Abuse Act, and the Electronic Communications Privacy Act" (Neupart & Munkedal) was passed.
1988-1990	Hackers on the Internet become a concern (PBS).
1988	The computer worm created by the hacker Robert Morris crashes 6,000 computers on the Internet (Neupart & Munkedal).
1988	The Computer Emergency Response Team (CERT) is created to address network security (PBS).
1989	The Cuckoos Egg is written after Clifford Stoll, a system administrator, catches hackers who had broken in to his system (PBS).
1990	ARPANET is closed (PBS).
1990	The Secret Service cracks down on hackers during Operation Sun Devil (Neupart & Munkedal).
1991	The gopher is created, which is "the first point-and-click way of navigating the files of the Internet" (PBS).
1991	The Michelangelo virus was scheduled to crash computers, but nothing happened (Livingston).
1993	The first graphics-based Web browser is created (PBS).
1994	Russian Vladimir Levin creates a hacker group that hacks into Citibank (Neupart & Munkedal).
1995	Kevin Mitnick incarcerated on charges of "wire fraud and illegal possession of computer files stolen from such companies as Motorola and Sun Microsystems" (Christensen).
1995	The movies Hackers and The Net are released (Neupart & Munkedal).
1996	"Hackers alter the websites of the U.S. Justice Department . . . , the CIA . . . , and the Air Force" (Livingston).
1996	"Approximately 40 million people are connected to the Internet" (PBS).
1996	“Smashing the Stack for Fun and Profit” by Aleph One UNIX Buffer overflows
1997	"A 15-year-old Croation youth penetrated computers at a U.S. Air Force base in Guam" (Christensen).
1998	The New York Times website was defaced to show the anger for the imprisonment of Kevin Mitnick (Livingston).
1998	Two hackers in China were sentenced to death for hacking into a bank and stealing money (Livingston).

Table 1:
A Chronology of Buffer Overflow Exploits [1] (Continued)

Year(s)	Event
1998	"U.S. Attorney General Janet Reno announces National Infrastructure Protection Center" (Neupart & Munkedal).
1998	The Pentagon was hacked by an Israeli teenager (Neupart & Munkedal).
1998	The hacker group L0pht speaks to the Senate about network security issues (Neupart & Munkedal).
1998	Win32 Exploits by Barnaby Jack
1999	Kelly Air Force Base was attacked by hackers, but the hackers were detected and stopped (Livingston).
1999	The United States Information Agency was hacked (Livingston).
1999	President Clinton announces he has set aside $1.46 billion for a plan to improve government computer security (Livingston).
1999	"Unidentified hackers seized control of a British military communication satellite and demanded money in return for control of the satellite (Christensen).
2000	Russian hacker "stole credit card numbers from an Internet music retailer and posted them on a website after an attempt to extort money from the company failed" ("Rebuffed Internet Extortionist Posts Stolen Credit Card Data").
2000	The following sites were attacked by hackers using denial of service: Yahoo, eBay, CNN.com, Amazon.com, Buy.com, ZDNet, E*Trade, and Datek (Levy).
2000	Kevin Mitnick is released from prison.
2001	Plug and play shell code for Win32 available over the Internet
2001	Code Red and Nimda worms exploit Win32 buffer overflows

1. The following is a listing of cited works used in developing this table.

Brunvand, Erik. “The Herioc Hacker: Legends of the Computer Age” Department of Computer Science October 15, 1996. February 22, 2000. http://www.cs.utah.edu/~elb/folklore/afs-paper/afs-paper.htm

Catman “Hacker History.” March 11,2000. http://www.thefuturesite.com/catman/history

Christensen, John. “The Trials of Kevin Mitnick.” *CNN.com* March 18,1999. February 9, 2000. http://www.cnn.com/SPECIALS/1999/mitnick.background

Levy, Stephen, and Brad Stone. “Hunting the Hackers” *Newsweek.com* February 21, 2000. March 11, 2000. http://www.newsweek.com/nw-srv/printed/us/st/a16375-2000feb13.htm

Livingston, Brian. “Project Against Trojan Horses” CNN.com January 17, 2000. March 11, 2000. http://cnn.com/2000/TECH/computing/01/17/trojan.horse.idg/index.htm

Neupart & Munkedal. “Hacker History-Soceity and Hackers in an Historical Perspective.” March 11, 2000. http://www.n-m.com/english/security/hackhistory.htm

P.B.S. “PBS Life on the Internet.” March 11, 2000. http://www.pbs.org/internet/timeline

Williams, Jim. “Hacker History-The Real Programmers” *About.com* April 26, 1999. February 28, 2000. http://netsecurity.about.com/compute/netsecurity/library/weekly/aa042699.htm?rnk=r&terms=%22The+Real+Programmers%22

Chapter 3

Buffer Overflows-A Taxonomy

The most exciting phrase to hear in science, the one that herolds new discoveries, is not "Eureka!" (I found it!) but "Thats funny"...

- Isaac Asimov

3.1 Introduction

We have demonstrated the omnipresent nature of Computer vulnerabilities. In every operating system fielded, in every important network application released, programming errors which result in the buffer overflow vulnerability have allowed unauthorized users to enter systems, or authorized users to take unauthorized actions. In recent years we have seen the development of sophisticated vulnerability databases and vulnerability exploitation tools by the so-called "computer underground". Detailed descriptions of how to find vulnerable states have appeared in various periodicals such as PHRACK and 2600, and on the USENET [17]. A large subset of these discussions essentially show how to probe a system for clues that indicate the system is running software known to be vulnerable to a buffer overflow exploit, or that it is being administered in such a way as to allow an attacker to run a buffer overflow attack.

Some of these tools are capable of automating the exploitation of vulnerabilities that were thought to require considerable expertise, including the buffer overflow. These tools, ready-made and of considerable complexity [18], are freely and widely available, and pose a significant threat that cannot be ignored. The celebrated Kevin Mitnick is an early example of a vandal who used such tools and databases to penetrate hundreds of computers before being caught [19]. Our study clearly demonstrates that with the widespread use of computers today, and increased computer knowledge in the hands of people whose objective is to obtain access to unauthorized systems and resources, it is no longer possible or desirable to implement security through obscurity [20].

Efforts to eliminate buffer overflow security flaws have failed miserably; indeed, sometimes attempts to patch such a vulnerability have increased the danger. Further, designers and implementers rarely learn from the mistakes of others. We see the recurrence of the buffer overflow as a case in point as it is an easily preventable security hole that can, for the most part, be eliminated through proper programming practice. Part of the complication is, that in the past, the buffer overflow problem was rarely documented in a format to allow for the creation of a database of characteristics related to the exploit

mechanism itself. A search of the literature demonstrated this with two broad categories of associated information. The first category involved numerous examples of what could be termed 'underground how to manuals' associated with a particular buffer overflow exploit [7][45][47][49][50][58]. The second concerned the hundreds of security advisories that alert users that a particular vulnerability exists along with details on the affected software release and the patch information. A well ordered systematic global description of the buffer overflow vulnerability does not exist.

In this research we are concerned with finding these types of vulnerabilities in released third party software and we want to find them quickly. When a product is released it is subject to potential attack from a global community. This is in effect equivalent to the exploratory testing of the software on a massive scale. As a security auditor, a single entity, we must ask how can we gain an advantage over the global community in testing the product. The testing performed by the global community can be characterized as a random and chaotic process, certainly with no defined plan. Each element within the community explores the product according to their own particular goals and is limited by individual expertise. One way to gain significant advantage is to leverage this haphazard testing process in our favor. We can do this by developing a methodology that consists of specific tasks, objectives and documentation, that make finding both those software systems and the individual domains within those systems, that may contain an exploitable buffer overflow, a systematic process. In this approach we first identify, at a top level, the potential areas of a system that enable this type of vulnerability. This allows us to identify candidate target software. Once the target has been selected we use a similar approach at a more detailed level to perform directed interactive testing or more commonly, exploratory testing.

In operational terms, exploratory testing is an interactive process of concurrent product exploration, test design, and test execution. The outcome of an exploratory testing session is a set of notes about the product, failures found, and a concise record of how the product was tested. When practiced in a rigorous methodical fashion, it yields consistently valuable and auditable results. Key to our approach is an abstraction of the buffer overflow exploit problem in terms that can be applied by the security auditor to develop the heuristic elements which define and are integrated into the actual exploratory test plan.

To effectively test and ensure that computer systems are secure against malicious attacks we need to analyze and understand the characteristics of faults that can subvert security mechanisms. A classification scheme can aid in the understanding of the types of buffer overflows that cause security breaches by categorizing and grouping faults that share common external characteristics. Knowing how systems have failed can help us build systems that resist failure. Petroski [21] makes this point eloquently in the context of engineering design, and although software failures may be less visible than those of the bridges he describes, they can be equally damaging as we discussed in chapter two.

In this chapter we formulate a classification strategy then collect and organize[1] a number of actual buffer overflow exploits that have caused failures. We will establish a framework for a database format, so that designers, programmers, analysts and auditors may do their work with a more precise knowledge of what has gone before. The buffer overflows we examined were able to cause conditions or circumstances that resulted in denial of service, unauthorized disclosure, unauthorized destruction of data, or unauthorized modification of data and included most modern operating systems and ancillary software. There is perhaps a legitimate concern that this kind of information could assist those who would attack computer systems. Partly for this reason, we have limited the cases described here to those that already have been publicly documented elsewhere and are relatively old. We do not suggest that we have assembled a representative random sample of all known buffer overflows, but we have tried to include a wide variety.

Using our collection we then classify the buffer overflow vulnerability along axiomatic lines and present our findings as a taxonomy. The unique contribution of this work is an analysis of the problem in a new way which will allow for an improvement of security in existing systems, and will provide a framework for exploratory software testing that highlights those areas prone to exploitable buffer overflow security flaws. This contrasts the work to [22], which argued that a preventative approach using formal methods to design secure systems is appropriate. We emphatically agree; however, as nonsecure systems continue to be used, our work is presented with the hope it will guide maintainers and software implementers to improve the security of these flawed systems and software. We offer the taxonomy for the use of those who are presently responsible for auditing released software and identifying exploitable flaws. We feel that buffer overflow vulnerability data, organized this way and abstracted, could be used to identify the heuristic elements critical for successful interactive product testing and resultant identification of exploitable security flaws.

In addition, our taxonomy attempts to organize information about buffer overflows so that, as new vulnerabilities are reported, readers will gain a fuller understanding of which parts of systems and which parts of the system operational cycle are more susceptible to vulnerabilities than others. This information should be useful not only to those faced with the difficult task of assessing the security of a system already built, but also to software designers. To accurately assess the security of a computer system, an analyst must find its flaws. To do this, the analyst must back up and understand the system at a global level and recognize that buffer overflow vulnerabilities that threaten computer security exist in unique areas of the system. The issue of how to find the underlying vulnerabilities in the first place is of paramount importance in any exploratory test plan.

---

1. See Appendix C for a collection of (20) case studies

This chapter presents a brief discussion of the characteristics that are desired in any effective taxonomy. This is followed by a brief review of current taxonomies in the computer and network security field. These taxonomies in present use include lists of categories, lists of terms, empirical lists, results categories, and matrices. As large global classifications of vulnerabilities they all suffer from a lack the focus on our particular problem. A proposed, highly specific, taxonomy for the buffer overflow exploit is then presented. Our taxonomy was developed from the criticisms of the taxonomies that have been published and from using a process or operational viewpoint of ways, means, and ends. When one uses this viewpoint, an attacker on computers or networks can be seen as attempting to reach or "link" to ultimate objectives. This link is the buffer overflow exploit and is established through a defined operational sequence of access, tools, and results that allows these attackers to connect to their objectives.

3.2 An Effective Taxonomy

Classification of information is as much an art form as it is a science especially when classifying computer security vulnerabilities. Systems that immediately come to mind are the library classification systems such as the Library of Congress Classification (LCC) or the Dewey Decimal Classification (DDC). These were developed to arrange printed matter in topical or disciplinary categories (i.e. to position books related to the same or similar subjects next to each other). Our subject is much more abstract and as a result we must go much further in developing a classification strategy for the buffer overflow vulnerability.

A taxonomy can be broadly described as a system of classification allowing one to uniquely identify something. The best-known example, the science of systematics, classifies animals and plants into groups showing the relationship between each. Further, the classification is unique, so two of the same animal will always be classified with the same groups. That is, if one considers the hierarchy to be a tree structure with each branch uniquely numbered, each species of animal or plant is uniquely identified by an exhaustive and unambiguous 6-tuple (kingdoms, phylums, classes, orders, family, genus). This biological hierarchy is repeatable regardless of who is doing the classifying, widely accepted and provides useful insight on each particular instance of classification. Any taxonomy we wish to use to in an attempt to describe buffer overflow vulnerabilities should provide the same benefits. The primary goal of our taxonomy is to enable a security auditor to focus on those areas where the potential for an exploitable condition may exist in order to develop test heuristics. Other specific objectives include the specification of a historical record of buffer overflow exploits in a form that system designers and implementers can use to anticipate flaws in their systems. In other words we look for a way to describe the buffer overflow in a form useful for database characterization as well as an improved method of showing common characteristics in related buffer overflow exploits for prevention and elimination.

A taxonomy similar to the biological classification of plants and animals will accomplish these. Such a taxonomy allows one to classify each vulnerability as a unique ordered tuple. This is essential, as it will allow the security auditor to narrow the search domain when attempting to detect new vulnerabilities. Perhaps more importantly, it allows us to determine how many instances of externally similar exploits are known, which in turn suggests where efforts to reduce or eliminate the flaws should be focused. It also allows us to characterize conditions under which the flaw arises, suggesting ways to detect new instances of the flaw.

3.2.1 Characteristics of a Satisfactory Taxonomy

We use our buffer overflow exploit analysis to devise a classification, or set of classifications, that enable the analyst to abstract the information desired from a set of system properties. This information may be a set of services, used as an intrusion mechanism to transport the buffer overflow “code”; a set of environment conditions necessary for an attacker to exploit the vulnerability; a set of characteristics common to a particular end result; or other data. The specific data used to classify a vulnerability is very important and highly dependent upon the specific goals of the classification. This explains why multiple classification schemes are extant. Each serves the needs of the community or communities to which its classifier belongs.

Our problem of interest is to discover buffer overflow vulnerabilities before attackers can exploit them therefore our classification will focus on the external system level domain where these flaws tend to exist. As we are using the biological classification model our taxonomy should follow and have classification categories with the following characteristics:

- *Mutually exclusive* - The categories do not overlap with classification in one category excluding classification in all others. In other words similar vulnerabilities are classified similarly. For example, all buffer overflow exploits using TCP packets should be grouped together. However, we do not require that they be distinct from other attacks. For example, a vulnerability involving a TCP packet may be used by many different services. Hence it should also be grouped with the particular destination service. As a result, exploits may fall into multiple classes. Because a buffer overflow can rarely be characterized in exactly one way, a realistic classification scheme must take the multiple characteristics common to each attack into account. This allows some structural redundancy in that different buffer overflow exploits may lie in the same class; but we expect (and indeed desire) this overlap.
- *Exhaustive* - When one considers the universe set of that being classified, when taken together, the categories are inclusive across all possibilities.
- *Accepted* - Classifications should be logical and intuitive so that they become generally approved for widespread use.

- *Unambiguous* - Regardless of who is classifying the rule set is clear and precise so that classification is not uncertain. This implies that our classification should be primitive. Determining whether an exploit falls into a class requires a 'yes' or 'no' answer. This means each subclass has exactly one property. For example, the question “does the vulnerability manifest itself in the UNIX operating system or Windows NT” is ambiguous; the answer could be either, or both. For our scheme, this question would be two distinct questions: “is the vulnerability specific to UNIX” and “is the vulnerability specific to Windows NT”. Both can be answered 'yes' or 'no' and there is no ambiguity to the answers.

- *Repeatable* - Regardless of who is classifying, repeated applications of the rule set result in the same classification. This means that our classification terms should be well-defined. For example: What is the reason for a buffer overflow attack? One can argue that the classification “reason for” is simply an alternate manifestation of the classification “intent”. However the classifier “reason for” is much more subjective and may include an attempt by the classifier to define the personal motives of the attacker. Where as the classification “intent” is limited to the specific objectives of the buffer overflow attack. For this reason the term “reason for” would not be considered as a valid classification term.

These distinctions represent useful tools that can be used to evaluate possible taxonomies. It should be expected, however, that a satisfactory taxonomy would be limited in some of these discriminators. It is important not to loose sight of the fact that a taxonomy is only an approximation of reality, one that is used to leverage a greater understanding in a field of study. It is important to note that this is only an approximation, and as such, it may fall short in some categories. This may be particularly the case when the characteristics of the data being classified are widely divergent, imprecise and uncertain, as was the data collected for this study. In fact it can be demonstrated that this is a characteristic of the buffer overflow in general. Nevertheless, we believe that our classification approach is valid and is an important and necessary process for the systematic study of the buffer overflow.

3.3 Previous Efforts

Faults in operating system software and application software associated with bounds checking can lead to the security breaches associated with the buffer overflow exploit. A systematic knowledge of this class of faults, their general characteristics, and how they enter the system is important to ensure secure operation and to preserve the integrity of stored information. Previous work includes other taxonomies [23], [24], [25] that have recently been developed for organizing data about software defects and anomalies of all kinds. These are primarily oriented toward collecting data during the software development process for the purpose of improving it. Other classifications are more oriented towards a broad ordering of security vulnerabilities in general. Past work, while not suitable for our particular effort, allows us insight to the approach required to correctly abstract difficult software flaws.

We are primarily concerned with a unique security flaw that is detected only after the software has been released for operational use and includes both operating systems and applications. Our taxonomy, while not incompatible with these efforts, reflects this perspective. Our goals are more limited than

those of these earlier efforts in that we seek primarily to provide an understandable record of a singular type of security flaw that has occurred. They are also more ambitious, in that we seek to categorize a vulnerability that is difficult to characterize and define. We will attempt to classify not only the details of the exploit, but also the mechanism of the attack, the place it entered the system, and the attackers intent.

3.3.1 Protection Analysis (PA) Project

The Protection Analysis (PA) Project conducted research on protection errors in operating systems during the mid-1970s. The group published a series of papers, each of which described a specific type of protection error and presented techniques for finding those errors. The proposed detection techniques were based on pattern-directed evaluation methods, and used formalized patterns to search for corresponding errors [27]. The results of the study were intended for use by personnel working in the evaluation or enhancement of the security of operating systems [28]. The objective of this study was to enable anyone with little or no knowledge about computer security to discover security errors in the system by using the pattern-directed approach. The final report of the PA project proposed four representative categories of faults [26]. These were designed to group faults based on their syntactic structure as follows:

- Domain errors, including errors of exposed representation, incomplete destruction of data, incomplete destruction of content, and incomplete destruction of context
- Validation errors, including boundary condition errors and failure to validate operands
- Naming errors, including aliasing and incomplete revocation of access to a deallocated object.
- Serialization errors, including multiple reference errors and interrupted atomic operations

The PA project's classification is too broad and nonspecific to be useful for our purposes, however the group's research was an important foundation in helping us understand the formalized process of fault classification.

3.3.2 The Research in Secured Operating Systems (RISOS) Project

The RISOS project was a study of computer security and privacy conducted in the mid-1970s [29]. The project was aimed at understanding security problems in existing operating systems and to suggest ways to enhance their security. The systems whose security features were studied included IBM's OS/MVT, UNIVAC's 1100 Series operating system, and the TENEX system for the PDP-10. The final report of the project discussed several issues related to data security in general. It suggested administrative actions that could prevent unauthorized access to the system and methods to prevent disclosure of information. The main contribution of the study was a classification of integrity flaws found in the operating systems studied. The fault categories proposed by researchers of RISOS [29] are the following:

- Incomplete parameter validation
- Inconsistent parameter validation
- Implicit sharing of privileged/confidential data
- Asynchronous-validation/Inadequate-serialization
- Asynchronous-validation/Inadequate-serialization
- Inadequate identification/authentication/authorization
- Violable prohibition/limit
- Exploitable logic errors

The fault categories proposed in the RISOS project are general enough to classify faults from several operating systems, but the generality of the fault categories prevents the type of fine-grain specific classification that we require for our unique security flaw.

3.3.3 The Landwehr Taxonomy

Landwehr et al. [30] published a collection of security flaws in different operating systems and classified each flaw according to its genesis, or the time it was introduced into the system, or the section of code where each flaw was introduced. This study was motivated by the observation that the history of software failures is largely undocumented. The research also compared the frequency of security incidents against the taxonomies with the goal of helping software programmers and system administrators "...to focus their efforts to remove and eventually prevent the introduction of security flaws..." [30]. The authors remark that: "...knowing how systems have failed can help us build systems that resist failure...." The objective of Landwehr taxonomy was to describe how security flaws are introduced, when they are introduced, and where the security flaws can be found. An outline of the three categories in the taxonomy is presented below:

1) By Genesis:

   a. Non-malicious

      validation errors
      domain errors
      serialization/aliasing errors
      errors that result from inadequate identification/authentication
      boundary condition errors
      logic errors

   b. Malicious

      viruses
      worms
      trojan horse

time bombs
trap doors

2) By time of introduction:
    a. Development
    b. Maintenance
    c. Operation

3) By location:
    a. Operating system routines
    b. Support software
    c. User programs

The Landwehr security flaw taxonomy by genesis extended the previous research of the PA and RISOS groups with the introduction of a new category of flaws, intentional flaws. Intentional flaws are flaws that are introduced deliberately into a program so that they can be exploited at a later time. Trap-doors, Trojan horses, time bombs, and covert channels are examples of intentional flaws. Inadvertent flaws in the Landwehr taxonomy were similar to the flaw taxonomies found in the PA and RISOS projects. The Landwehr security flaw taxonomy by time of introduction characterized security flaws by when they were introduced into a system. The Landwehr study was the first to describe when security flaws were introduced during the software development life cycle (SDLC). The Landwehr security flaw taxonomy by location characterized security flaws by where the security flaw occurred. This taxonomy differentiated security flaws as either hardware or software and subdivided the software category into operating system, support, and application flaws.

The Landwehr taxonomies extended security flaw research by providing multiple classification hierarchies for characterizing security flaws. The realization that security flaws cannot be simply described by a single attribute was an important contribution and one that we will choose to follow in our analysis. Of the three taxonomies discussed so far, only the by location taxonomy is germane to our classification of the buffer overflow flaw.

3.3.4 The Marick Survey

Brian Marick [31] published a survey of software fault studies from the software engineering literature. Most of the studies reported faults that were discovered in production quality software. Although the results of the study are insightful, the classification scheme provided is more appropriate to the survey

format that Marick used. For this reason we find it not suitable for the organization and classification of buffer overflow exploit data.

3.3.5 The Bishop Taxonomy

Bishop studied flaws in the UNIX operating system and proposed a flaw taxonomy for the UNIX operating system [32]. Rather than describe security flaws using a single set of categories, Bishop proposed that security flaws should be described using a single taxonomy that is composed of several collections of categories or axes. The proposed axes were:

- Nature of the flaw;
- Time of introduction;
- Exploitation domain; effect domain;
- Minimum number of components;
- Source of the identification of the vulnerability

Although this study extended security flaw taxonomy research by including a number of criteria that we previously had not considered, we find it too narrow and specific for our purposes.

3.3.6 Aslam's Taxonomy

Aslam's study [33], [34], as extended by Krsul [35], approached classification slightly differently, through software fault analysis. Aslam proposed to classify the faults found in the UNIX operating system in a manner complementary to Bishop [32]. The objective of this taxonomy was to unambiguously classify security faults and provide a theoretical basis for the data organization of a vulnerability database. Selection criteria were provided for each subclass so that all fault categories are specific and distinct. The Aslam taxonomy contained the following major categories:

- Coding Faults: These are flaws introduced during software development. Coding faults were further subdivided into:
- Condition validation errors and
- Synchronization errors.
- Emergent Faults: Flaws that result from improper installation of software, unexpected integration incompatibilities, and when a programmer fails to completely understand the limitations of the run-time modules. Emergent faults were subdivided into:
- Configuration errors
- Environmental errors.

The Aslam taxonomy was used as the theoretical basis for the Crosbie et.al. vulnerability database that was used in the Intrusion Detection In Our Time (IDIOT) IDS [36].

The classification suffers from flaws similar to those of the PA and RISOS studies and upon close analysis breaks down when considering different levels of abstraction. For these reasons we find it inappropriate for the classification of our singular security flaw.

3.3.7 The Lindquist Taxonomy

Lindquist and Jonsson proposed two taxonomies that differed from previous work in that characterized security flaws as attacks. The classification was based on the technique used and the result of the attack. The objectives of Lindquist and Jonsson research were threefold: (1) to establish a framework for the systematic study of computer attacks; (2) to establish a structure for reporting computer incidents to incident response team; and, (3) to provide a mechanism for assessing the severity of an attack [37].

The Lindquist Intrusion Technique Taxonomy is based on previous research by Neumann and Parker [38] and divided intrusive techniques into three principal categories:

1) Bypassing Intended Controls: This category includes attempts to attack passwords, spoof privileged programs, and attack programs utilizing weak authentication.
2) Active Misuse of Resources: This category includes active attacks such as buffer overflows as well as exploitation of world writeable system objects.
3) Passive Misuse of Resources: This category includes all probing attacks that attempt to identify weaknesses in the scanned system [37].

The Lindquist Intrusion Result Taxonomy is based on the Confidentially, Integrity, and Availability (CIA) model. It divided intrusion results into three categories:

1) Exposure: These are attacks against system confidentially and are subdivided into disclosure of confidential information and service to unauthorized entities.
2) Denial of Service: These are attacks against system availability and are subdivided into selective, unselective, and transmitted attacks. Transmitted attacks are attacks that affect the service delivered by other systems to their users, not the service delivered by our system to other systems.
3) Erroneous Output: These are attacks against system integrity and are subdivided into selective, unselective, and transmitted attacks [37].

The Lindquist Intrusion Result Taxonomy use of the widely respected CIA model provides a good theoretical foundation for the classification of system attacks. Intrusion results are an important component of the buffer overflow vulnerability as each exploit is tailored to the attack. As such, the Lindquist Intrusion Result Taxonomy will be used as a model for our analysis in developing a proposed buffer overflow exploit taxonomy.

3.3.8 The Fisch Damage Control and Assessment Taxonomy

A review of literature reveals only one dedicated intrusion response taxonomy - the Fisch DC&A taxonomy [39]. The Fisch DC&A taxonomy classified the intrusion response according to:

- When the intrusion was detected (during the attack or after the attack);
- The response goal (active damage control, passive damage control, damage assessment, or damage recovery).

While the categories covered by the Fisch taxonomy should be components of any intrusion based taxonomy, additional components are necessary to more accurately classify our unique exploit as an attack.

3.3.9 Summary of Previous Methods

Our research into security taxonomies has allowed us to view the body of doctrine in order for us to develop our own approach for the classification of the buffer overflow exploit as an attack. With perhaps the exception of the Lindquist Taxonomy, previous security taxonomy research has been too focused, not focused enough, or focused on intrusion response. Another problem with existing classification techniques is that they all rely on independent discriminators by using a single category for each class of flaw. The Lindquist Taxonomy we found to be an approach that we could model although it remains too broad for our purposes and converges on the independent classifier approach. In addition, it did not consider the type of attack, type of service being attacked, sensitivity of the information being attacked, or the environmental constraints required for a successful exploit. The next section addresses these open research issues within the context of developing a new taxonomy of buffer overflow exploits.

3.4 A Taxonomy of Buffer Overflow Exploits

The classification of security flaws, vulnerabilities, and intrusions has received much attention in the past as we have demonstrated in the proceeding section. However, this previous work does not directly carry over into the area of classifying buffer overflow exploits as attacks. Hierarchies are the most common structure for organizing large collections of data. Current classification methods used by those who track buffer overflow exploits [10], [40], [41], [42] can be described as being a flattened class space. That is one class for every leaf in the hierarchy. If any structure exists at all in the current classification of buffer overflow exploit data it is by date of occurrence and by the effected operating system or application.

Discovering the natural structure that underlies a field of inquiry is a challenging and interesting problem. First, we must examine the buffer overflow within the context of a security issue. Most buffer overflow attack scripts are not general purpose and require a very specific target. They generally require

application X version Y running on operating system Z. This breadth of targets and approaches makes it difficult for to accurately define a classification taxonomy based on independent discriminators. To generalize, we call the set of buffer overflow exploits attacks. It is clear that buffer overflow exploits have the potential to be extremely dangerous, however researchers are doing little work to understand the nature of these attacks. What are the characteristics? How have attacks changed over time? What level of sophistication is required to use documented exploits? In what areas of existing systems are these attacks occurring? More importantly within the context of this thesis, can we develop a heuristic foundation for exploratory testing?

We propose to answer these questions by creating a taxonomy of buffer overflow attacks as classified by a set of dependent classifiers. To do this we will abstract the buffer overflow in a hierarchal manner beginning at the most elementary level and working upward. At the most basic state the buffer overflow represents a programming flaw of common occurrence. At the next level we can look at the buffer overflow as a potential security vulnerability succeeded by the buffer overflow as an attack. At the top level of the hierarchy, the buffer overflow can represent a security incident. Before discussing vulnerability, attack and incident classification schemes, we define our terminology:

- Vulnerability: A misconfigured or faulty element of a computer system that can be exploited for unauthorized use of the computer.
- Attack: An attack is simply a single use of a buffer overflow exploit in an attempt to gain unauthorized access, or an attempt to gain unauthorized use, regardless of success. The attack is characterized by a script or set of instructions which format the buffer overflow exploit that an intruder puts into action to accomplish the unauthorized enterprise.
- Incident: Any event when an attacker, by methods or actions, uses or attempts to use an attack exploit against a target. An incident can be characterized by grouping like attacks that can be distinguished from other incidents because of the degree of similarity of objectives, techniques, and timing and the distinctive signature of the attackers.

As buffer overflow incidents are made up of attacks, we propose that it is appropriate to develop a taxonomy based within the context of the attack itself. This can then be extended to include the broader classification of incidents. A taxonomy of attacks is, however, more useful than the higher level global classification of incidents. When one considers buffer overflow incidents, the usefulness of an attack taxonomy remains constant across the entire universe set of incidents and therefore includes all attacks. We believe that such an attack taxonomy is useful both in the evaluation of existing systems as well as in the development of new systems. By comparing well defined categories of attack mechanisms against the details of the target system of interest, one can begin to establish a system for classification. From our review of existing vulnerability classification research we concluded that existing taxonomies do not directly apply to our attack classification scheme. This is because attacks have features that do not exist in security flaws or vulnerabilities:

- Goals,
- Specifications for the attacking host,
- Transmission methods by which the attacker reaches the target, and
- Requirements the attacker must meet in order to launch the attack.

In addition, unlike vulnerability classification schemes, attack classification schemes are not necessarily concerned with identifying the specific exploited flaw. This is particularly useful as we are limited to a single known flaw, the buffer overflow. When one considers the class of successful buffer overflow exploits several homogeneous features exist:

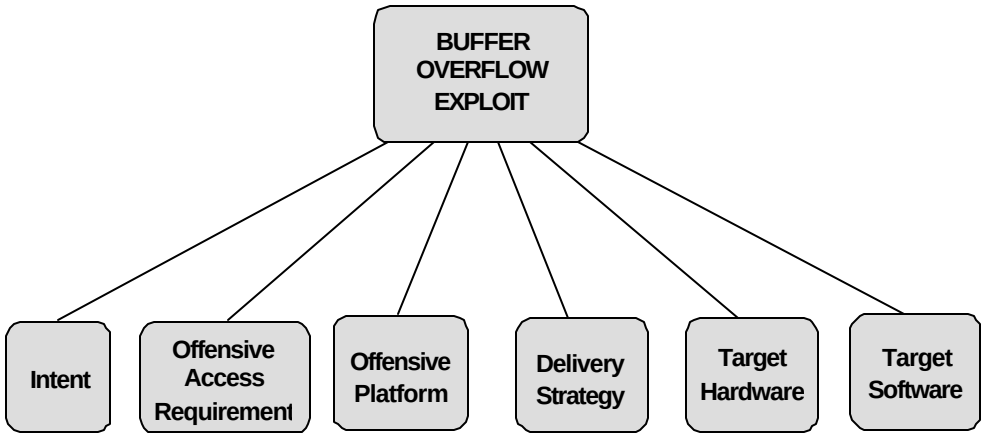
- The buffer overflow exploits more than one vulnerability
- The buffer overflow may uses more than one transmission mode
- The buffer overflow results in different goals

The existing vulnerability classification systems strive to use independent classifiers. This is dissimilar from our attack based classification scheme as we rely on dependent classifiers. Dependent classifiers allow one to choose multiple categories within a class while independent classifiers force one to choose a single category for each class. Using the buffer overflow as our definition of an attack, there exists a buffer overflow that no attack classification scheme that uses independent classifiers can uniquely classify. The reason is that our definition of an attack does not specify how many vulnerabilities the attack can exploit, how many goals the attack may achieve, or how many transmission methods it can use. No reasonable scheme of independent classifiers will uniquely classify an attack that is the union of all existing attacks.

The class of buffer overflow exploits contains many scripts that have multiple goals, multiple delivery methods, work against multiple targets, from multiple platforms, and that exercise multiple vulnerabilities. As a result of the wide diversity of attack scripts, a taxonomy that uses independent classifiers is difficult. Our approach is to use a classification scheme that is fine grained enough to yield interesting results but broad enough to allow us to quickly characterize attacks. We classify each attack within the following (6) major categories each of which is a classification hierarchy in itself:

1) Intent
2) Target Hardware
3) Offensive Platforms
4) Transmission Protocol
5) Offensive Requirements
6) Target Software

Figure 3: The (6) major classification categories

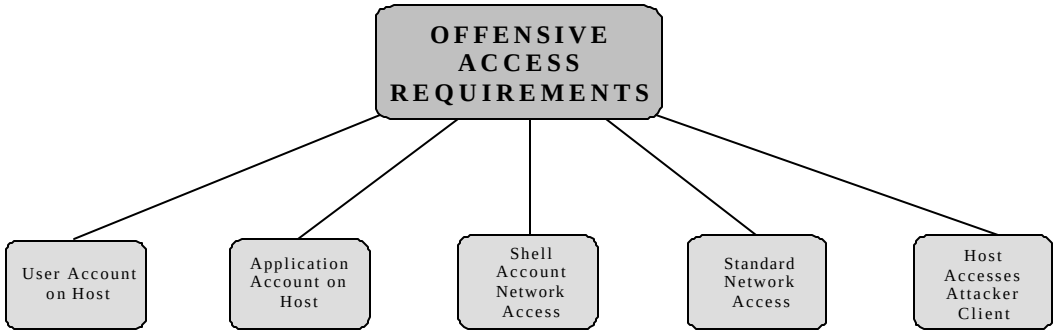


These six major categories establish the overall dimensions of the classification state space (i.e. a given buffer overflow exploit attack is either in or out of a given category). Classes that are higher in a hierarchy can mean “other”, “all”, or a subset of the lower classes. Given the dimensions of the space and the exploit information, any person should be able to recreate the overall state space. Within each major class there exists a hierarchy of subclasses. Some of the dimensions of this subspace will be classifications that will be self-sustained, consistent, objective, and capable of distinguishing important features that can be used to find patterns of and dependencies that might help us better understand the nature of buffer overflow exploits.

3.4.1 Offensive Access Requirements

A buffer overflow attack can be classified according to the access requirements needed to exploit the target system. These form the most part are self explanatory with the exception of the category “Host Accesses Attacker Client”. This represents an emergent area of so-called "*pushed*" exploits where the target of the attack must access the attacker’s information in order for the attacker to launch the attack. This is most common with web sites that attack users that visit.

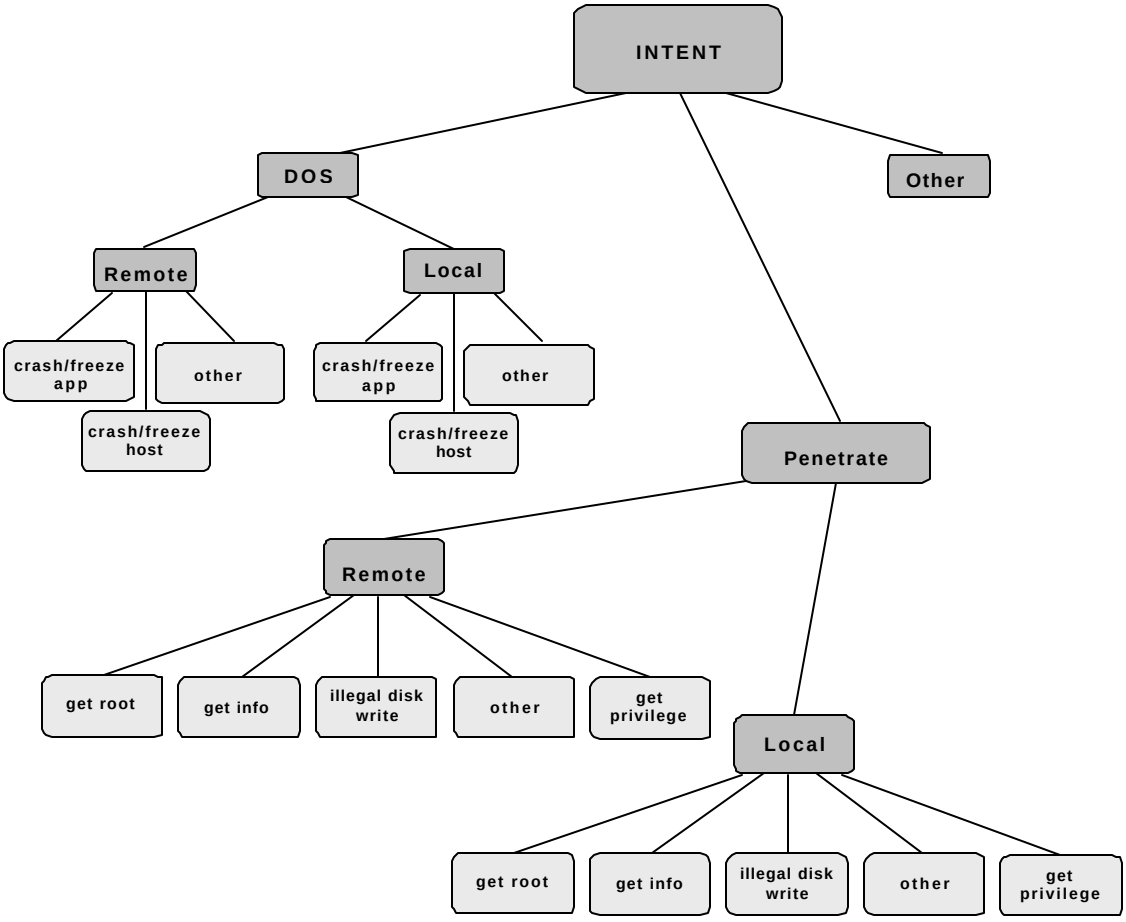
Figure 4: Hierarchy of Offensive Requirements



3.4.2 Intent

This broad category does not attempt to discriminate each possible malicious objective of a particular attach rather, it focuses on a system-based classification. At a top level the buffer overflow exploit can be utilized to cause a denial of service (DOS)[1] or more commonly to penetrate the system to some degree. The hierarchy of intent is shown below.

Figure 5: Hierarchy of Intent



This hierarchy considers all daemons to be applications or services unique to a particular OS. Since we also considered the network protocol stack to be separate from the OS, this means that usually only local attacks can abuse the OS. This was part of our reasoning for the local and remote attack intent classifications.

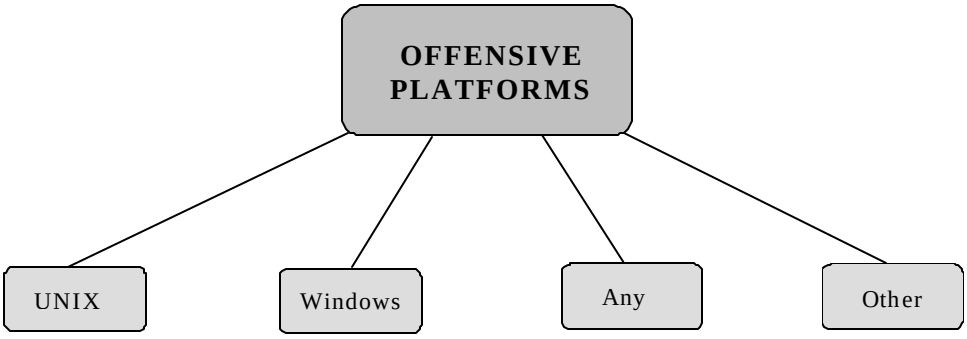
3.4.3 Offensive Platform

For the most part, buffer overflow exploits are machine independent when one considers the offensive platform requirements. In some cases however, there may be dependencies associated with the

1. See Appendix 3 Exploit 1A as an example

services or operating system the target platform is running that requires a unique offensive platform. In practice, it is common knowledge that the offensive platform of choice is typically running a variant of Linux.

Figure 6: Offensive Platform Requirements



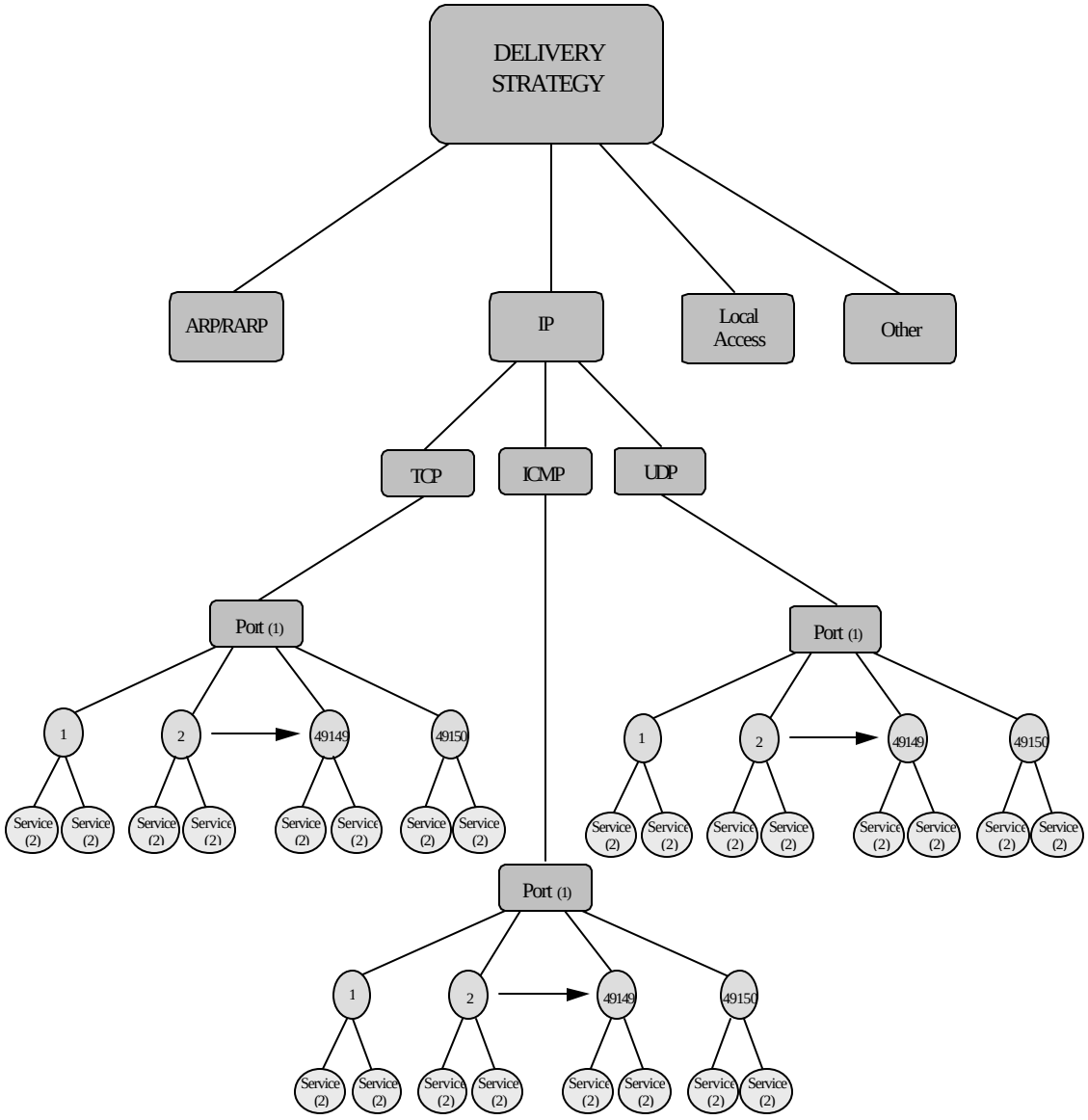
3.4.4 Delivery Strategy

This represents the connection between the attackers and their objectives. The buffer overflow is characterized by an input of excess data into a process. In order to reach the desired process an attacker must inject a payload of malicious code and to do this the attacker must take advantage of some sort of delivery mechanism. For the remote buffer overflow exploit, this is a key part of the attack mechanism. In most cases a remote attack centers on a particular host service or application and can be distinguished by:

- A process that is listening on a open communications port
- The process accepts client side supplied input
- Relies on standard communications protocols

The hierarchy associated with transmission of the buffer overflow exploit represents a meta-category, as it is the key discriminator used in this taxonomy. It allows differentiation between the hundreds of known buffer overflow exploits and is an important classification tool for the development of exploratory test heuristics. It contains the majority of information associated with services, applications, port numbers and communications protocols.

Figure 7: Delivery Strategy

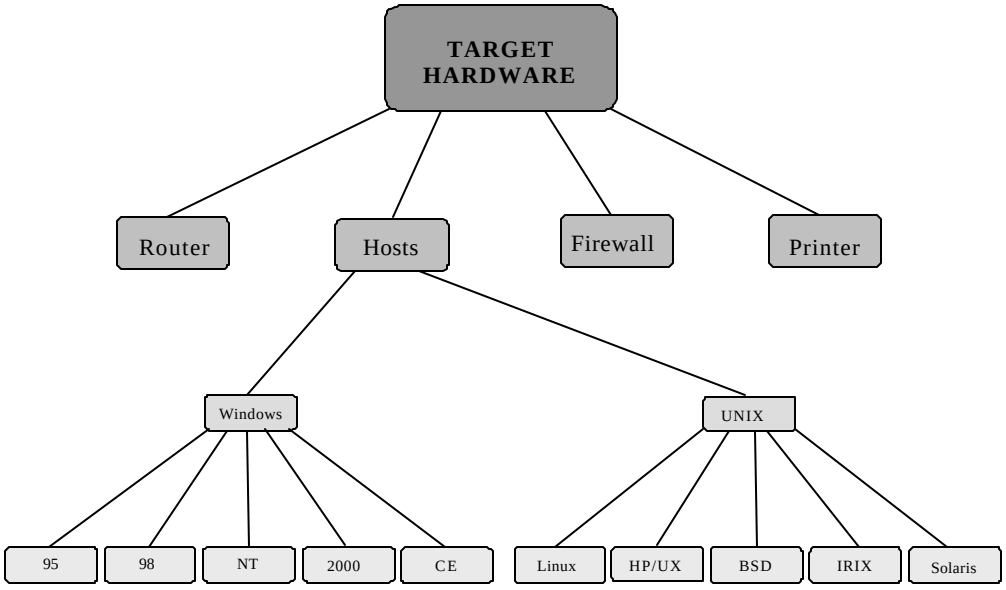


(1) See Appendix B for listing of common ports
(2) See Appendix A for listing common Multipurpose Internet Mail Extensions (MIME) types

3.4.5 Target Hardware

Under target hardware, we mark the type the machine that the attack abuses. As we defined, classes that are higher in a hierarchy can mean “other”, “all”, or a subset of the lower classes. What this means for example, is that checking the target type Unix could mean that the attack effects all Unix hosts, a subset of the Unix operating systems listed, or a completely different Unix operating system not listed.

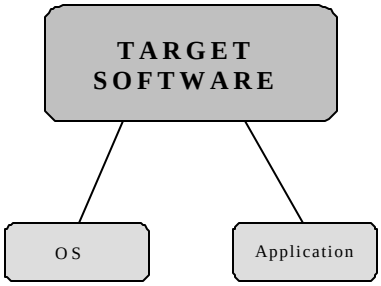
Figure 8: Target Hardware



3.4.6 Target Software

Under attack software, we identify the software that the attack abuses. Often, it is difficult to determine whether a daemon process is running as part of an application or part of the OS. Our solution is to attempt classification along the lines of application (that may use an OS daemon) and OS software.

Figure 9: Target Software



3.5 Case Studies

The following case studies exemplify the types of security flaws resulting from buffer overflows. Without making claims as to the completeness or representativeness of this set of examples, we believe they will help the programmer to know where the areas in a completed program are prone to buffer overflow exploit. More importantly, we believe that the data when classified in this manner will assist the security auditor to develop targeted exploratory test heuristics.

All of the cases documented here reflect actual flaws in released software. As we are concerned primarily with proprietary closed source Operating Systems and applications we have selected our data set as representative of those exploits common to the Windows family of platforms. For each case, a source (usually with a reference to a publication) is cited, the software/hardware system in which the flaw occurred is identified, the flaw and its effects are briefly described, and the flaw is categorized according to the taxonomy. Where it has been difficult to determine with certainty the exact category of an exploit feature, the most probable category (in the judgment of the author) has been chosen, and the uncertainty is indicated by the annotation '?'. In some cases, a buffer overflow exploit is not fully categorized.

Our taxonomy allows us to group cases according to the systems on which they occurred however, since we are focused on the Windows platform our data set would reflect this preference. It is important to note that Unix systems exhibit an approximate equal number of exploited buffer overflow vulnerabilities when compared to Windows systems, especially in recent years. Since readers may not be familiar with the technical details of all of the elements included in the taxonomy, brief introductory discussions of relevant details are provided as appropriate.

3.6 Summary

In this chapter we presented a buffer overflow exploit classification scheme that helps in the unambiguous classification of buffer overflow attacks that is suitable for data organization and processing. A representative database of exploits using this classification was implemented and is being used to aid in our fundamental understanding of the buffer overflow exploit throughout the remainder of this thesis. This taxonomy was not meant to be a complete one and is certainly open for modification. We believe that our scheme can be easily expanded because the criteria used for the taxonomy does not rely on software implementation details and is designed to encompass the general external characteristics of the buffer overflow exploit. Also, our existing categories can be extended to include any new exploits that cannot be classified into the existing categories, should any be found.

We used a small database that also needs to be extended with more exploits. The database currently has 20 significant buffer overflows across Windows systems only. We believe that there exists data to extend the collection to over 400 cataloged buffer overflows and would include other systems such as UNIX as well as routers and switches. Once this is complete, a more complex evaluation of the database can be performed for some of our original research goals: building heuristic test elements, guide software design and testing, and monitor the evolving characteristics of the buffer overflow exploit.

Chapter 4

The Buffer Overflow Exploit-Technical Discussion

For every complex problem, there is a solution
that is simple, neat, and wrong.

- Henry L. Menken

4.1 General Description

A buffer overflow occurs in a program anytime the program writes more information in an array, the buffer, than the space allocated in memory for it. This causes the adjacent area, the areas above the direction of buffer growth, to be overwritten. When this occurs all previously stored values are corrupted. Buffer overflows are defined as programming errors that are typically introduced into a program as a result of the programmer failing to enforce boundary conditions on the data being copied into the buffer. Unfortunately, as we shall soon demonstrate, buffer overflow programming flaws are quite common as a direct result of certain widely used and dangerous C library functions, those that handle strings in particular. Once a buffer overflow vulnerability has been coded into a program testing may not uncover it, so that the vulnerability may exist in the program undiscovered, hidden and silent for years. The potential then becomes one of the program being the target of a sudden attack in which the vulnerability is exploited to gain unauthorized access to a system.

A buffer overflow may occur by accident during the execution of a program. With this type of circumstance, the chances are very unlikely that it will lead to a security compromise of the system. Most often the information that is clobbered, in areas adjacent to the buffer, will only cause the program to crash or produce results that are obviously incorrect. On the other hand, in a buffer overflow attack, it is the objective of the attacker is to use the vulnerability to corrupt information in a carefully controlled way in order to execute malicious code designed by the attacker. If this succeeds, the attacker effectively hijacked the control of the system. Once control is transferred to the malicious code, it carries out the instructions of the attacker, usually the granting of complete unauthorized system access.

In particular, many attacks have been successful against Windows NT and Windows 2000 systems [45] [56] [58] [47] [59]. We remark, that although this thesis is concerned with proprietary software and we have limited our discussion to Windows systems, buffer overflows are applicable to most Operating Systems. Axelsson [55] compared the security of Windows NT and UNIX systems against known types of attack, and found them to be roughly equally vulnerable.

A buffer overflow attack may be local or remote. In a local attack the attacker already has access to the system and may be interested in escalating his/her access privilege. A remote attack is delivered through a network port, and may achieve both simultaneously by gaining unauthorized access and maximum access privilege.

Summarizing, we see that a buffer overflow attack usually consists of three parts:

1) The planting of the attack code into the target program;
2) The actual copying into the buffer which overflows it and corrupts adjacent data structures;
3) The hijacking of control to execute the attack code;

We now examine in more detail the technical mechanism of buffer overflow attacks.

4.1.1 The Hardware/Software Interface

To understand the characteristics of most buffer overflows, we first must understand the way memory is structured and organized within the machine when a typical program runs. On many systems, virtual address space is dedicated to each process and that space is somehow mapped to real memory. In this discussion we will describe memory organization and layout and explain the relationship between a function and memory space. We will outline the processes that are, in theory, allowed to address big chunks of continuous memory. We will show how parts of this memory can be potentially abused.

4.1.1.1 What defines a program?

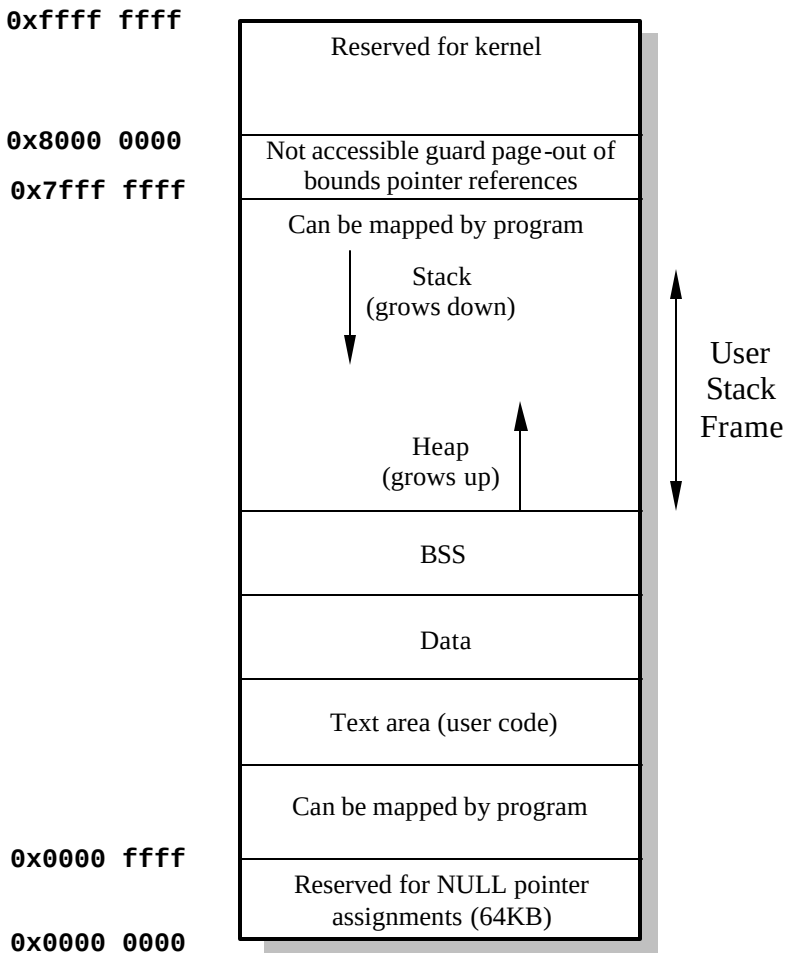
In general terms we can view a program as an instruction set, expressed in machine code (regardless of the language used to write it) and it is this program that we commonly call a *binary* or an *executable*. To arrive at this binary file, the high level source language that includes all variables, constants and instructions is processed by the compiler. In effect then, this binary file is a compile time object. This section presents the memory layout within the different parts of the binary.

4.1.1.2 Memory Organization

In order to understand what goes on while executing a binary, we need to have a look at the organization of virtual memory. It relies on different well defined areas for segmenting tasks between user and kernel process space. A Windows (in this case NT) process embodies many things such as, amongst others, a running program, one or more threads of execution, the process' virtual address space and the dynamic link libraries (DLLs) the binary uses. The process has 4 GB of virtual address space to use. Half of this is, from address 0x00000000 to 0x7FFFFFFF, private address space where the program, its DLLs and stack (or stacks in the case of a multithreaded program) are found. The other half, address 0x80000000 to 0xFFFFFFFF is the system address space where such things as NTOSKRNL.EXE, the

(kernel program) and the (hardware abstraction layer) HAL are loaded (ref. figure 10). When a program is run, NT creates a new process. It loads the program's instructions and the DLLs the program uses into a private address space (the text area). This area is read-only and it is shared between every process associated with running a given binary. Attempting to write into this area causes a *segmentation violation error*. The first thread is started and a stack is initialized.

Figure 10: Windows NT Memory Layout



Memory Layout

This represents the default behavior. It can be changed as of to assign 3 GB as private address space and 1 GB as system address space. This is to boost the performance of programs, such as databases, the require large amounts of memory.

Before attempting any further explanation, let's recall a few things about variables in C. The global variables are used throughout the entire program while the local variables are only used within the function where they are declared. The static variables have a defined size depending on their type, when

they are declared. Variable types can be *char*, *int*, *double*, or memory addresses in the case of *pointers.* On a machine utilizing the Intel architecture, the *pointer* is a word and represents a 32bit integer address within memory. With the use of pointers, the size of the area pointed to is obviously unknown at the time of compilation. To explicitly allocate a memory area, a dynamic variable is used. This variable is really a pointer pointing to that allocated address space. It is also important to note that global/local, static/ dynamic variables can be combined without complications.

With this understanding, let's go back to the memory organization for a given process. The *data* area stores the initialized global static data (the value is provided at compile time), while the *bss* segment holds the uninitialized global data. The machine is able to reserve memory space at compile time since the size of the data is defined according to the objects they hold.

This memory space, reserved at compile time for program execution, that contains the grouping of both local and dynamic variables, is known as the user stack frame. We know that our high level languages allow us to invoke functions in a recursive manner. As a result, the number of instances of a local variable is not known in advance. The concept of the stack allows for this functionality by pushing the values required by each instance of the function onto the stack.

The stack is located on top of the highest addresses within the user address space or user frame, grows in a downward direction, and works according to a LIFO model (Last In, First Out). The bottom of the user frame area is reserved for allocation of dynamic variables. This region is called heap, it grows in a upward direction and contains the memory areas addressed by pointers and the dynamic variables. When a variable is declared, the associated pointer (a 32bit variable) is either in BSS or in the stack and does not point to any valid address. When a process allocates dynamic memory (i.e. using malloc) the address of the first byte of that memory (also 32bit number) is put into the pointer.

4.1.1.3 The Stack and the Heap

Each time a function is called, a new environment must be created within memory space for local variables and the function's parameters. We use the term ‘environment’ to define all the elements appearing while a function is executing. That is, all arguments, local variables, as well as the return address within the execution stack. The *ESP* (extended stack pointer) register holds the top stack address, which is at the bottom as in our representation the stack grows downward. The *ESP* stack pointer, because of the last in first out implementation, points to the last element added to the stack. It is important to note that the *ESP* is architecture dependent and may sometimes point to the first free space in the stack. The *ESP* can be changed in a number of ways both indirectly and directly. When something is pushed onto the stack the *ESP* increases accordingly. When something is *POP*ed off of the stack the *ESP* shrinks. The *PUSH* and

POP operations modify the *ESP* indirectly. The *ESP* can be manipulated directly, with an assembly code instruction of "*SUB ESP, 04h*" which pushes the stack out by four bytes or one word.

We could express the address of a local variable within the stack as an offset relative to *ESP*. However, because items are being continuously added or removed to and from the stack, the offset of each variable would then need readjustment, a very inefficient proposition. The use of a second register allows to improve on our efficiency. To do this we use register *EBP* (extended base pointer) to hold the start address of the environment of the current function. Therefore, it's enough to express the offset related to the value in this register. It stays constant while the function is executed. Now we have a easy method to find the parameters or the local variables within a function.

The basic unit used within the stack is a word. On i386 CPU architectures it is (32) bits long or (4) bytes. This value is different across other architectures. As an example, on Alpha CPUs a word is (64) bits. The stack only manages words, and what this means is that every allocated variable uses the same word size. The stack is usually manipulated with just 2 CPU instructions:

1) *PUSH* value: This instruction puts the value at the top of the stack. As stack growth is downward, the push reduces *ESP* by a word to allow us to get the address of the next word available in the stack. The stored value is given as an argument within that word;

2) *POP* destination: This instruction puts the item from the top of the stack into the 'destination'. In other words, it puts the value held at the address pointed to by *ESP* in the destination and increases the *ESP* register. In effect, nothing is really removed from the stack. Just the value of the pointer to the top of the stack changes.

4.1.1.4 The Registers[1]

The registers are a designated series of storage areas that hold exactly one word (4 bytes), while the memory space itself is made of a series of words. Each time the machine places a new value within a register, the old value is lost. Registers are designed to allow direct communication between memory and CPU. As a point of interest, the first 'e' appearing in the registers name designates a register for use within a 32bit architecture and means 'extended'. This feature indicates the evolution between old 16bit and present 32bit architectures. The registers can be divided into 4 categories:

1) general registers: *EAX, EBX, ECX* and *EDX* are all used to manipulate data;

2) segment registers: the 16bit registers *CS, DS, ESX* and *SS*, all hold the first part of a memory address;

3) offset registers: these indicate an offset related to segment registers:

    a. *EIP* (Extended Instruction Pointer): indicates the address of the next instruction to

1. This discussion centers around the x86 intel architecture, other systems (alpha, sparc, etc) have registers with different names but similar functionality.

be executed;

b. *EBP* (Extended Base Pointer): indicates the beginning of the local environment for a function;

c. *ESI* (Extended Source Index): holds the data source offset in an operation using a memory block;

d. *EDI* (Extended Destination Index): holds the destination data offset in an operation using a memory block;

e. *ESP* (Extended Stack Pointer): the top of the stack;

4) special registers: they are only used by the CPU and will not be covered in this thesis.

To summarize our process memory, first we have the code or text segment with all data in this segment represented by assembler instructions that are executed by the processor. The execution of this code is non-linear, it can skip code, jump, and call functions depending on certain conditions. For this reason we have a pointer called *EIP*, or instruction pointer to keep track of where in the execution path we are. The address to where *EIP* points to always contains the address of the code that will be executed next. Second we have the data segment, a space for variables and dynamic buffers. Last we have the stack segment, which is used to pass data (arguments) to functions and as a space for variables local to those functions. The bottom or start of the stack usually resides at the very end of the virtual memory of a page, and grows in a downward direction. The assembler command *PUSH* will add a word to the top of the stack, and *POP* will remove one word from the top of the stack and put it in a register. To allow for direct access to stack memory, there is the stack pointer *ESP* that points at the top of the stack or lowest memory address within the stack frame.

Now that we have developed a familiarity with the organization of memory and it's association with our binary at run time we will turn our attentions to how a executable behaves from start to finish within the context of the i86 architecture.

4.1.2 Binary Execution at Run Time

In this section we will present the behavior of a program at run time from call to finish. A program is typically made up of functions ranging from the simple to the complex. At run time these functions are called then executed according to the flow of the program. Each time a function is called stack space is automatically allocated. The stack holds all information required within the context of the current function call for all function calls including the function call to *main()*. We can view this as a container

of information unique to each function call that is a continuous block of storage. We call this container a activation record or, alternatively, a stack frame. There are many things can go into an activation record. These contents, laid down at compile time, are generally both architecture-dependent and compiler-dependent. Some of the common items placed in stack frames, as we have seen, include values for the non-static local variables of the function, the arguments passed to a function (actual parameters), saved register information, and the address to which the program should jump when the function returns. We have shown that many of these items are kept in machine registers instead of on the stack, mainly for reasons of added efficiency (a compiler-dependent factor).

The purpose of this section is to detail the behavior of the stack and the registers during function execution. The buffer overflow exploit tries to interrupt the normal behavior of the function at run time. To understand this attack, it's useful to know what the normal behavior is. Executing a function is divided into three distinct steps:

1) the prologue: when entering a function two requirements must be accomplished. First the state of the stack must be saved before entering the function. Second the amount of memory required for running the function must be reserved.
2) the function call: when a function is called, its parameters are pushed onto the stack and the instruction pointer (IP) is saved to allow the remainder of the program to resume execution from the correct place after the function has completed it’s execution;
3) the function return: restores the organization of memory to the state immediately prior to calling the function. In this section we'll demonstrate using the following example:

```
void foo(int I, int j)
{
   char str[5];
   int k = 3;
   return;
}
int main(int argc, char **argv)
{
   int i = 1;
   foo (1, 2);
   i = 0;
   printf(“i = %d\n” , I);
}
```

4.1.2 Example 1

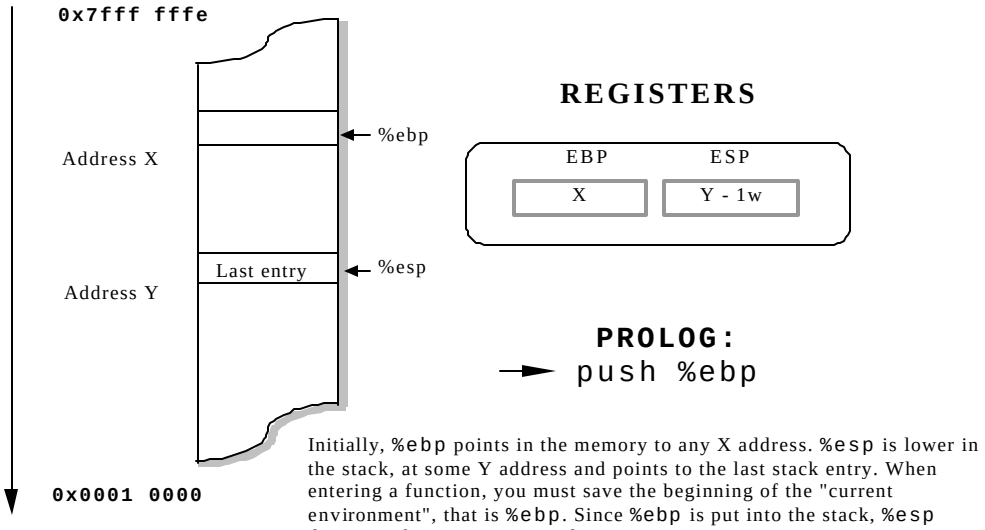
4.1.2.1 The Prolog

When looking at the assembly instructions, a function always starts with the instructions:

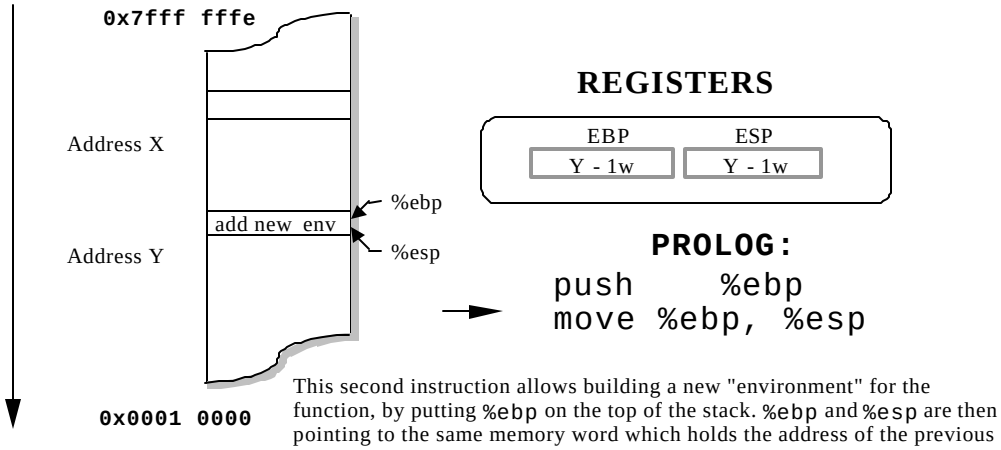
- *PUSH EBP*
- *MOV EBP, ESP*
- *PUSH ESP, 0Ch* //where 0Ch is program dependent

The combination of these three instructions together make what is called the prolog. The following figures detail our example program, specifically the way the *foo()* function works by detailing the stack mechanics associated with the handling of the local variables (*char str [5]=abcde*; and *k=3*). In addition we demonstrate the operations of the *EBP* and *ESP* registers:

Figure 11: Elementary Stack Behavior at Run Time - *The Prolog*

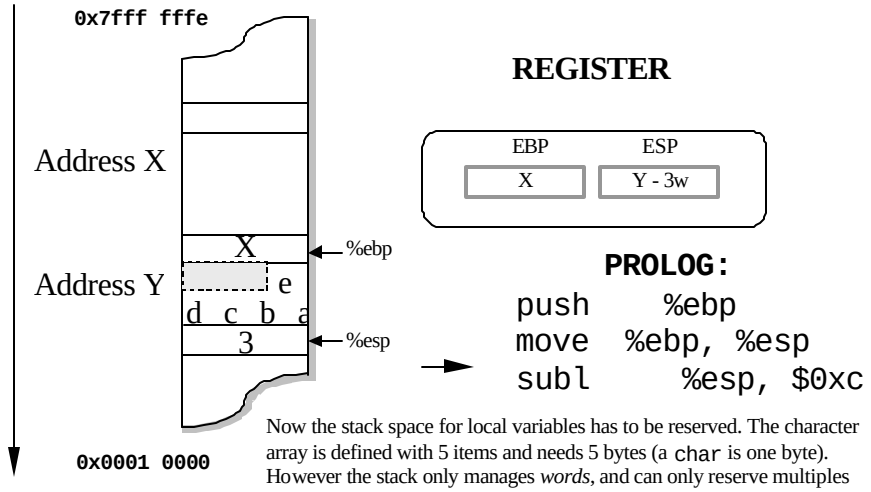


Initially, `%ebp` points in the memory to any X address. `%esp` is lower in the stack, at some Y address and points to the last stack entry. When entering a function, you must save the beginning of the "current environment", that is `%ebp`. Since `%ebp` is put into the stack, `%esp` decreases by a memory word.



This second instruction allows building a new "environment" for the function, by putting `%ebp` on the top of the stack. `%ebp` and `%esp` are then pointing to the same memory word which holds the address of the previous environment.

Figure 11: Elementary Stack Behavior at Run Time - The Prolog (Continued)



Now the stack space for local variables has to be reserved. The character array is defined with 5 items and needs 5 bytes (a `char` is one byte). However the stack only manages *words*, and can only reserve multiples of a *word* (1 *word*, 2 *words*, 3 *words*, ...). To store 5 bytes in the case of a 4 bytes *word*, you must use 8 bytes (that is 2 *words*). The grayed part could be used, even if it is not really part of the string. The `k` integer uses 4 bytes. This space is reserved by decreasing the value of `%esp` by `0xc` (12 in hexadecimal). The local variables use 8+4=12 bytes (i.e. 3 words).

Although the mechanism itself is important, what we really want to remember here is the position of the local variables. We notice that the local variables have a negative offset when related to ***EBP***. This is illustrated by the ***i=0*** instruction in the ***main()*** function. The assembly code detailed below demonstrates the use of indirect addressing to access the variable (***i***):

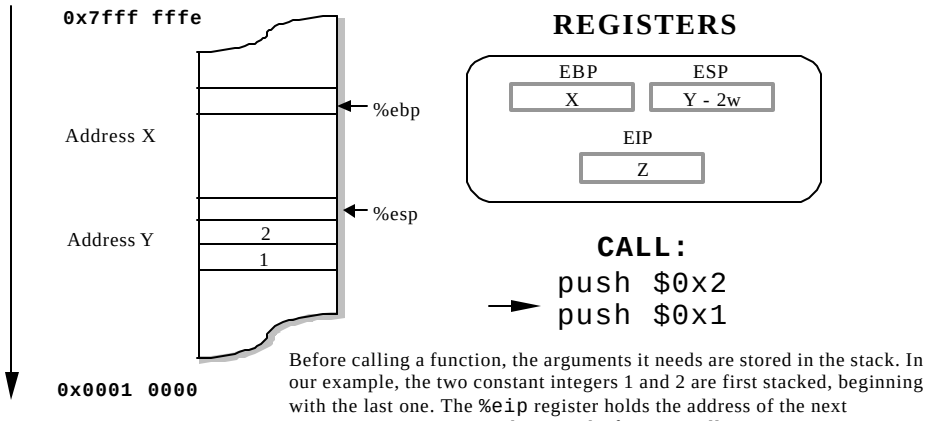
```
00401060:    MOV     dword ptr [epb-4],  0
```

What this line of assembly instructions means is move the source value 0, into the destination variable found at 'minus 4 bytes' relatively to the ***EBP*** register. The ***main()*** function contains only one local variable, variable (*i*), because it is the only one as well as the first one, its address is 4 bytes (i.e. integer size) 'below' the ***EBP*** register.

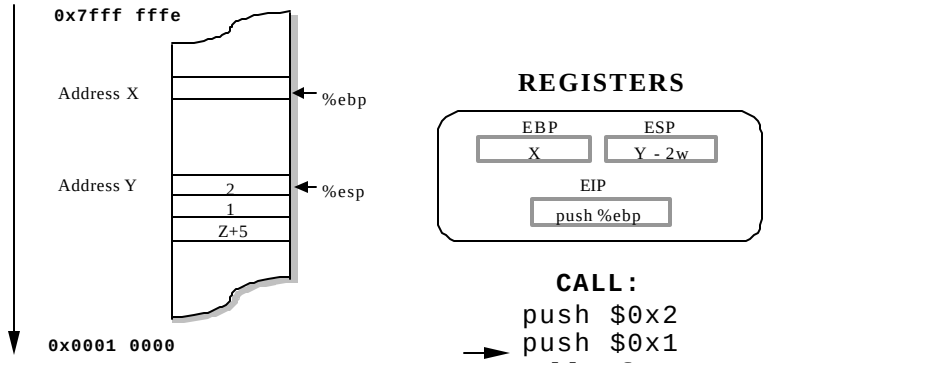
4.1.2.2 The Call

Similar to the way the prolog of a function prepares its environment, the function call allows the function to receive its arguments. In addition, once the function is terminated, program execution is allowed to resume at the exact place in the program which originally called the function. Using our example, let's take the ***foo(1, 2)*** function call in ***main()***.

Figure 12: Elementary Stack Behavior at Run Time - *The Call*



Before calling a function, the arguments it needs are stored in the stack. In our example, the two constant integers 1 and 2 are first stacked, beginning with the last one. The `%eip` register holds the address of the next instruction to execute, in this case the function call.



When executing the call instruction, %eip takes the address value of the following instruction found 5 bytes after (call is a 5 byte instruction - every instruction doesn't use the same space depending on the CPU). The call then saves the address contained in %eip to be able to go back to the execution after running the function. This "backup" is done from an implicit instruction putting the register in the stack :

```
push %eip
```

The value given as an argument to call corresponds to the address of the first prolog instruction from the foo() function. This address is then copied to %eip, thus it becomes the next instruction to execute.

Once we are within the body of the function, the arguments as well as the return address have a positive offset when related to ***EBP***, since the next instruction pushes this register onto the top of the stack. The ***j=0*** instruction in the ***foo()*** function illustrates this. The Assembly code, detailed below, once again uses indirect addressing to access the ***j*** variable.

```
0040103E:    MOV     dword ptr [epb+0Ch],  0
```

The 0Ch represents the +12 integer in hexadecimal (***0xc***). The notation used means put the source value 0 in the destination memory location found at "+12 bytes" relative to the ***EBP*** register. The variable (*j*) is the second argument to the function and it is located at 12 bytes 'on top' of the ***EBP*** register.

We arrive at 12 bytes in the following manner, 4 bytes for the instruction pointer backup, 4 bytes for the first argument and 4 bytes for the second argument. We will illustrate this in the next section; the return.

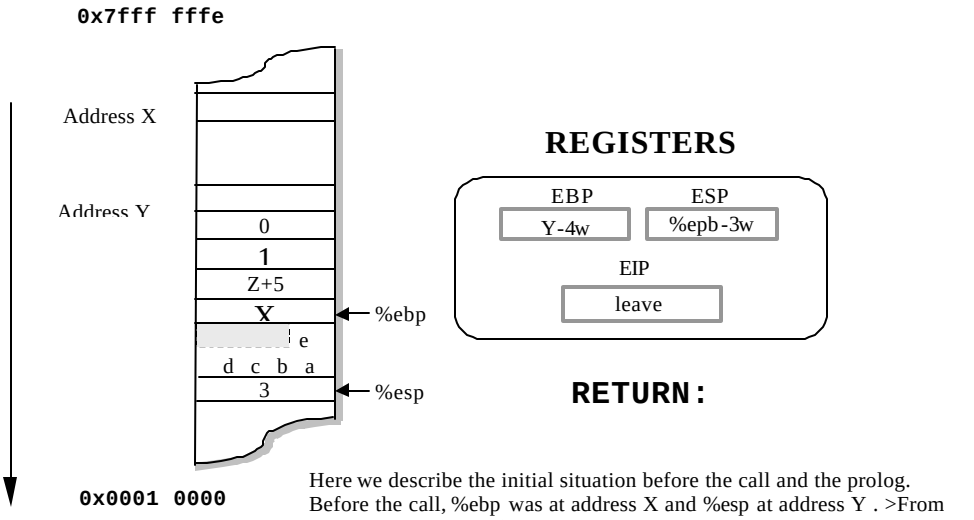
4.1.2.3 The Return

Leaving a function is accomplished using just two steps. First, our environment that was created for the function, must be cleaned up. This means putting ***EBP*** and ***EIP*** back in their original state as they were immediately before the call. Once this is accomplished, the stack must be checked in order to get the information related to the function we are just coming out of. The first step is done while we are still within the function. The instructions to accomplish this are:

- leave (which is really made up of the instructions: ***MOV ESP, EBP*** and ***POP EBP***)
- ret

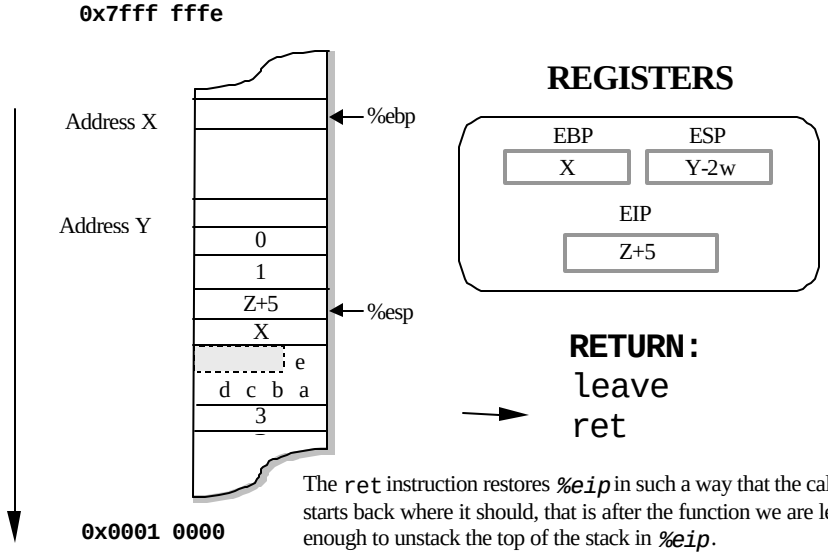
The second step is accomplished within the function where the subject call took place. This step consists of cleaning up the stack from the arguments of the called function. We demonstrate this by extending the previous example of the ***foo()*** function.

Figure 13: Elementary Stack Behavior at Run Time - *The Return*



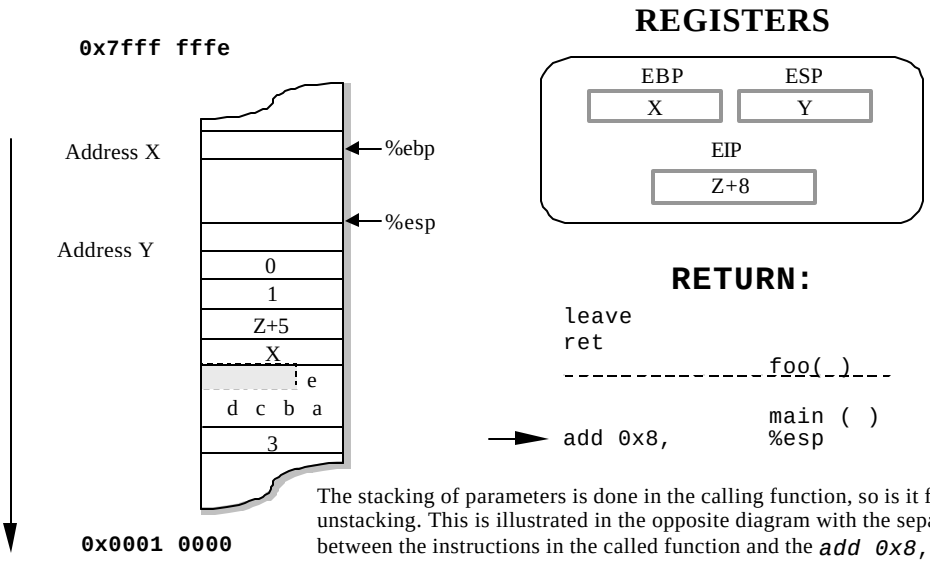
Here we describe the initial situation before the call and the prolog. Before the call, %ebp was at address X and %esp at address Y . >From there we stacked the function arguments, saved %eip and %ebp and reserved some space for our local variables. The next executed instruction will be leave.

Figure 13: Elementary Stack Behavior at Run Time - The Return (Continued)



The `ret` instruction restores ***%eip*** in such a way that the calling function execution starts back where it should, that is after the function we are leaving. For this, it's enough to unstack the top of the stack in ***%eip***.

We are not yet back to the initial situation since the function arguments are still stacked. Removing them will be the next instruction, represented with its `Z+5` address in ***%eip*** (notice the instruction addressing is increasing as opposed to what's happening on the stack).



The stacking of parameters is done in the calling function, so is it for unstacking. This is illustrated in the opposite diagram with the separator between the instructions in the called function and the ***add 0x8, %esp*** in the calling function. This instruction takes ***%esp*** back to the top of the stack, as many bytes as the ***foo()*** function parameters used. The ***%ebp*** and ***%esp*** registers are now in the situation they were before the call. On the other hand, the ***%eip*** instruction register moved up.

4.1.2.4 The Disassembly

We load up our first example (chap4_ex1.c) in the Microsoft Developer Studio 97 and compile using Visual C++ version 5. This build results in 79 KB executable binary file. The following disassembly of the binary has been highlighted to indicate the locations of the prolog, call and return utilized in our previous discussion of stack mechanics.

```
-- C:\Program Files\DevStudio\MyProjects\chap4\chap4_ex1.c
--
1:    /* test1.c */
2:
3:    void foo(int i, int j)
4:    {
00401020   push        ebp
00401021   mov         ebp,  esp      PROLOG for function foo
00401023   sub         esp,  0Ch
5:        char str[5] = "abcde";
00401026   mov         eax,  [00410A30]
0040102B   mov         dword ptr [ebp-8],eax
0040102E   mov         cl,  byte ptr ds:[00410A34h]
00401034   mov         byte ptr [ebp-4],  cl
6:        int k = 3;
00401037   mov         dword ptr [ebp-0Ch],  3
7:        j = 0;
0040103E   mov         dword ptr [ebp+0Ch],  0
8:        return;
9:    }
00401045   mov         esp,  ebp
00401047   pop         ebp            RETURN from function foo
00401048   ret
10:
11:   int main(int argc, char **argv)
12:   {
00401049   push        ebp
0040104A   mov         ebp,  esp      PROLOG for function main
0040104C   push        ecx
13:       int i = 1;
0040104D   mov         dword ptr [ebp-4],  1
14:       foo(1, 2);
00401054   push        2
00401056   push        1              CALL function foo
00401058   call        00401000
0040105D   add         esp,  8        RETURN from function foo
15:       i = 0;
00401060   mov         dword ptr [ebp-4],  0
   16:      printf("i=%d\n",i);
00401067   mov         eax,  dword ptr [ebp-4]
0040106A   push        eax
0040106B   push        410A38h        CALL printf
00401070   call        004010A0
00401075   add         esp,  8        RETURN from printf
17:   }
00401078   mov         esp,  ebp
0040107A   pop         ebp            RETURN from main function
0040107B   ret
---------------------- No source file  --------------------
```

4.1.3 Assessing Stack Overflow Vulnerabilities

4.1.3.1 The Activation Record

The main problem with buffer overflows, from an exploit point of view, is finding the security-critical region to overwrite in a manner consistent with the desired attack. With stack overflows we can demonstrate that there is always something security-critical to overwrite on the stack and that is the return address. In this section we will review some mechanics in assessing an overflowing stack to further our understanding of the buffer overflow condition. To demonstrate this we will create a stack-allocated buffer then overflow it in a manner that will allow us to overwrite the return address located in the stack frame. To implement this sample plan, we first have to figure out which buffers, in a program we can overflow as well as the characteristics of the overflow itself. As we have demonstrated, in general there are two types of stack-allocated data which exist. These two types of data include non-static local variables and parameters to functions.

We ask the question, “can we overflow both types of data?”. The answer is dependent on the stack location of the data. We can only overflow those data items with a lower memory address than the return address. With this in mind, our first order of business will to be to select some function then ‘map’ the stack. By mapping the stack we will be able to find out where the parameters and local variables are located relative to the return address we're interested in. In this method we will use both program output and binary disassembly to delineate our stack. We will start with the following simple C program:

```
void test(int i)
{
   char buf[12];
}
int main()
{
   test(12);
}
```

4.1.3 Example 1

The test function we will be using has one local parameter and one statically allocated buffer. In order to allow us to look at the memory addresses where these two variables are located (relative to each other), we'll modify our code slightly:

```
void test(int i)
{
   char buf[12];
   printf("&i = %p\n", &i);
   printf("&buf[0] = %p\n", buf);
}
int main()
{
   test(12);
}
```

4.1.3 Example 2

We compile then execute our modified code with the following results observed in the output:

```
&i = 0x0064FDF4
&buf[0] = 0x0064FDE0
```

Now we are able to look in the general vicinity of these data, and determine if we see anything that looks like a potential return address. We will start by looking eight bytes above *buf*, and stop looking past the end of integer (*i*)eight bytes. In order to accomplish this, we will again modify our code as follows:

```
/* Assign char *j as a global variable,
so we don't add anything to the stack
*/

char *j;
void test(int i)
{
   char buf[12];
   printf("&i = %p\n", &i);
   printf("&buf[0] = %p\n", buf);
   for(j=buf-8;j<((char *)&i)+8;j++)
      printf("%p: 0x%x\n", j,
*(unsigned char *)j);
}
int main()
{
   test(12);
}
```

4.1.3 Example 3

Notice that in order to get eight bytes beyond the start of the variable *i* we had to cast the variable's address to a *char**. The reason for this is because when C adds eight to an address, it is really add-

ing eight times the size of the data type it believes is stored at the memory address. What this means is, that by adding eight to an integer pointer we will increase the memory address by 32 bytes instead of the desired eight bytes. For this reason we use the *char** variable type. Here is a typical output from our new program:

```
C:\WINDOWS\Desktop>test2

&i = 0064FDF4
&buf[0] = 0064FDE0

0064FDD8: 0x3c    0064FDEA: 0x0
0064FDD9: 0xfd    0064FDEB: 0x0
0064FDDA: 0x64    0064FDEC: 0xf8
0064FDDB: 0x0     0064FDED: 0xfd
0064FDDC: 0x0     0064FDEE: 0x64
0064FDDD: 0x0     0064FDEF: 0x0
0064FDDE: 0x0     0064FDF0: 0x9d
0064FDDF: 0x0     0064FDF1: 0x10
0064FDE0: 0x40    0064FDF2: 0x40
0064FDE1: 0xe1    0064FDF3: 0x0
0064FDE2: 0x40    0064FDF4: 0xc
0064FDE3: 0x0     0064FDF5: 0x0
0064FDE4: 0x84    0064FDF6: 0x0
0064FDE5: 0x0     0064FDF7: 0x0
0064FDE6: 0x0     0064FDF8: 0x38
0064FDE7: 0x0     0064FDF9: 0xfe
0064FDE8: 0x3     0064FDFA: 0x64
0064FDE9: 0x0     0064FDFB: 0x0
```

The question that we really need to focus our attention on is, “does anything in the output look like a return address?”. Remember, a memory address is one word or four bytes, and our output is represented by the single byte *char *,* as a result we are only looking at things one byte at a time. How can we figure out the range where the return address will be located? We want to start by looking at the things that we know. One thing we know for sure is that the program will return to the *main()* function. Accordingly if we can get the address of the *main()* function and then look for a pattern of four

```
00401091   pop         ebp
00401092   ret
12:   int main()
13:   {
00401093   push        ebp
00401094   mov         ebp,esp
14:      test(12);
00401096   push        0Ch
00401098   call        00401005
0040109D   add         esp,4
15:   }
004010A0   pop         ebp
004010A1   ret
```

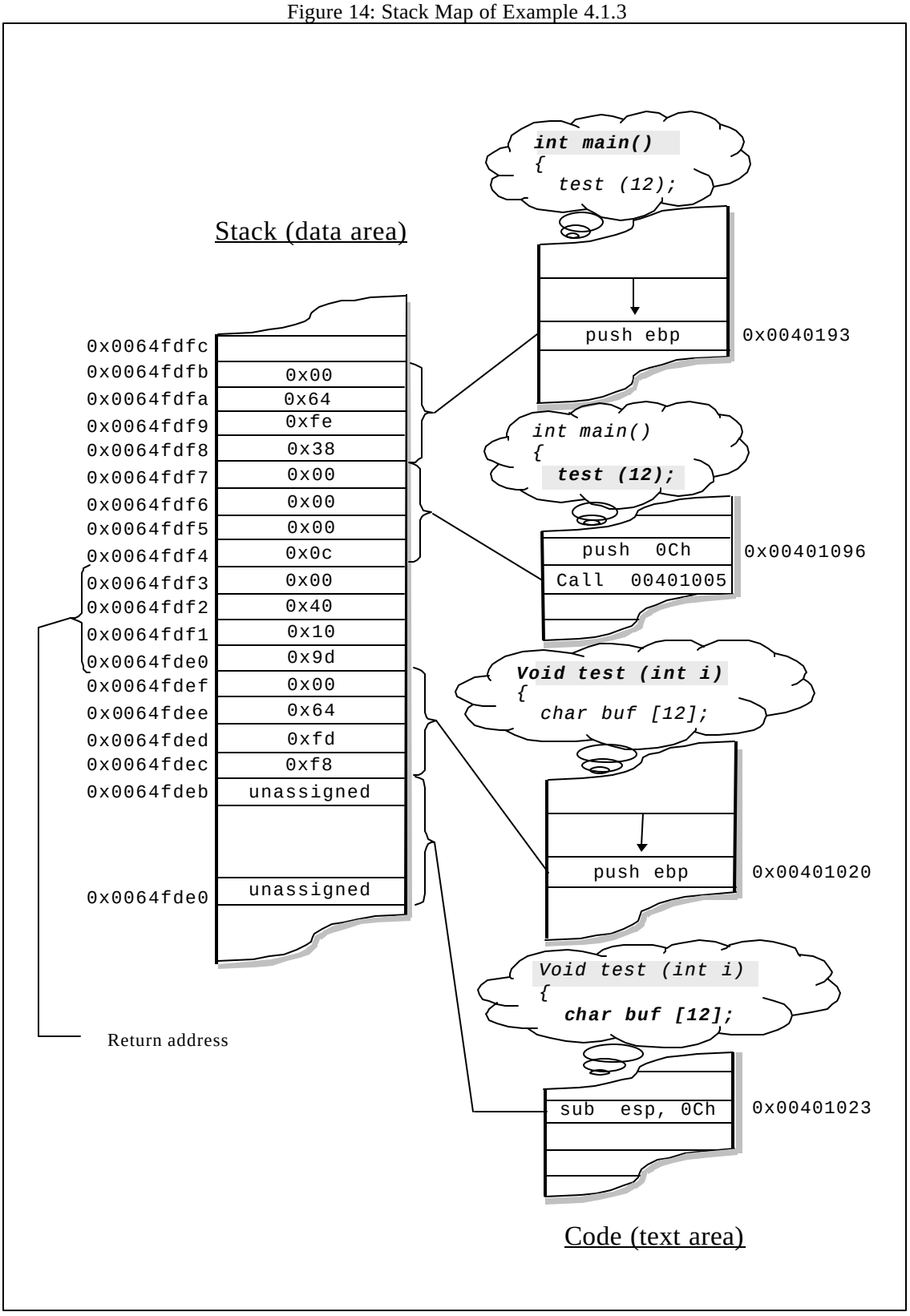
consecutive bytes that are quite close to this address we will locate our return address. To find the entry address for the `main()` function we will disassemble our binary:

From the disassembly we note that he entry to the function `main` is found at `0x004010A5`. Therefore in our output, we would expect to see three consecutive bytes, where the first is `0x00` the second `0x40`, and the third `0x10`. We expect this because we can demonstrate that the code from the start of `main()` to where the test returns is just a few bytes long. It is important to note that the ix86 architecture stores some multibyte primitive types in a way that seems strange. Data storage in memory is somewhat unusual in that it is stored last byte first and first byte last. Whenever we use data, as output for example, they are treated the right way. For instance if we print out one byte at a time, the individual bytes print ‘right side up’ however, when we look at four bytes that are consecutive in memory, they're in reverse order. With the reverse order of consecutive bytes in memory in mind, let's look for the main function's pattern of four bytes. We'll begin by locating those sections where the two most significant bytes are 0x0 and 0x40. This is because the most significant bytes are the last two in the set of four. In our output we find the following candidate:

```
0064FDF0: 0x9d
0064FDF1: 0x10
0064FDF2: 0x40
0064FDF3: 0x0
```

This memory fits our requirements perfectly. When we reassemble these four bytes, we get `0x0040109d`, which is 8 bytes past the start of `main()`.

Figure 14: Stack Map of Example 4.1.3



So now let's map out the entire stack, starting at the beginning of the *main()* function. The first PUSH onto the stack is *0x0064fe38* which becomes the base for this stack frame:

0x0064fdfb – 0x0064fdf8 Bottom of Stack (Frame Base for *main ()*)

The second PUSH onto the stack is our integer *int i* which is passed the value of (12) in our main function. This (32) bit word is represented by the hexadecimal value of *0x000000c*

0x0064fdf7 – 0x0064fdf4 Parameters

The next four bytes, starting at *0x0064fdf0*, constitute the return address. The next PUSH onto the stack (*0x0064fdf8*) is the base pointer for the *void test ()* function.

0x0064fdef – 0x0064fdec Previous Frame Pointer

The last (12) words on our stack represent our *char buf [ ]* and have been unassigned.

0x0064fdeb – 0x0064fde0 Buffer [ ]

We now have a good picture of what our stack frame looks like. The stack grows downwards toward memory address 0x00000000. This stack frame contains, listed in order, the function parameters, the return address of the calling function, the previous frame pointer, and finally our stack variable buffer [ ]. As the stack space for our local variables moves towards higher memory locations we can see that if we overflow these variables we will overwrite the return address for the function that we are in. In addition, as our buffer growth is towards higher memory locations and our stack growth downward, it becomes possible to overwrite the return address in the stack frame below us.

4.1.3.2 The Stack Smashing Buffer Overflow Exploit

In it’s most general form, this security attack achieves two primary goals:

1) The first is the injection of attack code into memory. This is typically a small sequence of machine op code instructions that spawns a shell, into a running process.
2) Change the path of execution of the running process to execute the attack code.

It is important to note that these two goals share a mutual dependence. It can be demonstrated that injecting attack code without the means to execute it is not necessarily a security vulnerability. If the buffer overflows and the overflow is long enough the return address will be corrupted, (as well as everything else in between). If the return address is overwritten by the buffer overflow so as to point to the attack code, this will be executed when the function returns and represents a change in the execution path of the running process. Thus, in this type of attack, the return address on the stack is used to hijack the control of the program.

Overwriting the return address, as explained above, gives the attacker the means of hijacking the control of the program as well as providing a mechanism to inject and store unique attack code. Most commonly the attack code is stored in what was the original buffer. Thus, the information which is copied into the buffer will contain both the binary machine language attack code as well as the address of this code which will overwrite the return address. This is by far, the most popular form of buffer overflow exploitation and is sometimes referred to as “smashing the stack”. As we will show below, the reason for this popularity is because by overflowing stack buffers in this manner one can achieve both goals of attack code injection and execution path change simultaneously.

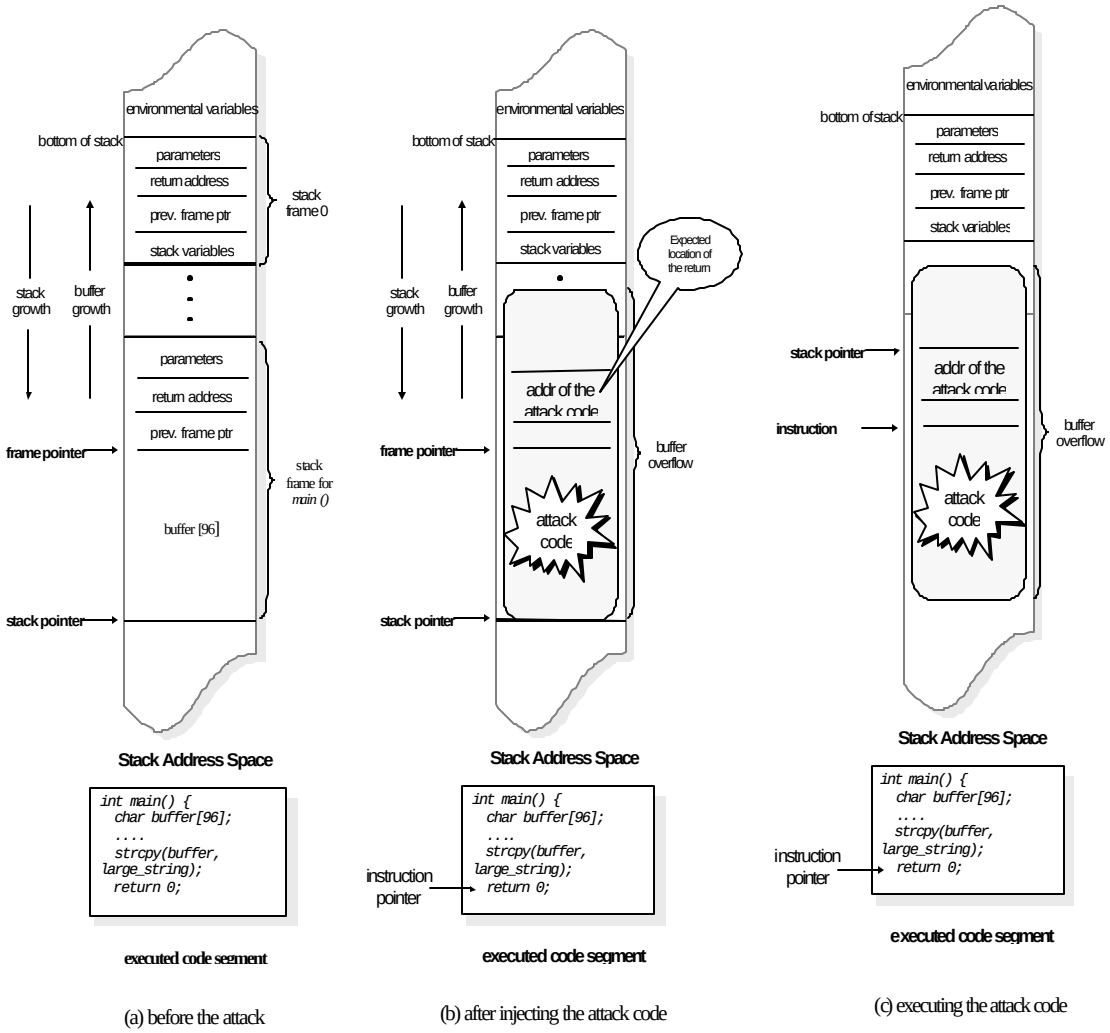
We briefly mention another type of buffer overflow attack which has been exploited and is known as the heap smashing attack. This is an attack associated with buffers that reside on the heap (a similar attack involves buffers residing in data space). Heap smashing attacks are much more difficult to exploit, for the simple reason that it is difficult to change the execution path of a dynamic running process by overflowing heap buffers. Although the potential as a security vulnerability exists, because of the difficulty involved, heap smashing attacks are far less prevalent.

A complete C program to demonstrate the so called stack smashing attack is presented below and was used by Leavy [7] in his landmark article which went on to demonstrate the exploit in great detail.

```
char attackcode[] =

"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\
xb0\x0b"

"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\
x40\xcd"
   "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];
int i;
long *long_ptr;

int main() {
  char buffer[96];

  long_ptr = (long *)large_string;
  for (i=0; i<32; i++)
    *(long_ptr+i) = (int)buffer;
  for (i=0; i<(int)strlen(attackcode); i++)
    large_string[i] = attackcode[i];
  strcpy(buffer, large_string);
  return 0;
}
```

Figure 15 detailed below illustrates the memory address space of a process undergoing this type of attack. The process stack after executing the initialization code and entering the *main()* function is illustrated in Figure 15(a) The process stack has been frozen in time at a point before executing any of the instructions. We pay particular attention to the structure of the top stack frame. This is the stack frame for the *main()* function. Common to all stack frames, this stack frame contains, in order, the function parameters, the return address of the calling function, the previous frame pointer, and finally our stack variable buffer.

Figure 15: A *Stack Smashing* Buffer Overflow Exploit[1]



Looking at Levy's program in the above example, the sequence of instructions for spawning a shell is stored in a string variable called `char attackcode []`. This attackcode is the op code equivalent to executing `exec(``/bin/sh'')` on a UNIX system. Similar Op code can be crafted for Windows systems with more difficulty however, primarily due to the unique call format involved. Within the `main()` function, the two for loops prepare the attack code by writing two sequences of bytes to `large_string`. Starting on line 16, the first for loop writes the (future) starting address of the attack

1. Figure from ARASH BARATLOO, NAVJOT SINGH; *"Transparent Run-Time Defense Against Stack Smashing Attacks"*; http://www.research.avayalabs.com/project/libsafe/doc/usenix00/paper.html

code. Starting on line~18 the second for loop copies the attack code (excluding the terminating null character). On line 20 the stack is completely smashed by the `strcpy()` function.

Figure 15(b) illustrates the stack space after executing the `strcpy()` call. It is important to note how the unsafe use of `strcpy()` simultaneously achieves both requirements of the stack smashing attack. First it injects the attack code by writing it on the stack space of the running process. Second by overwriting the return address with the address of where the attack code is located, it effectively instruments the stack to change the path of execution. The attack is completed once the return statement on line 21 is executed. At this point the instruction pointer "jumps" and the machine starts executing the attack code. We illustrate this step Figure 15(c).

In a real security attack, the attack code would normally come from user input. In the worst case scenario the attack would be originated remotely and transmitted over a network connection. A successful attack on a running process would give the attacker an interactive shell. This shell would be at the same access privilege as the process that was smashed. On a UNIX based system a successful exploit of a root process would result in an interactive shell with a user-ID of root commonly referred to as a root shell. Although this paper focuses on buffer overflow vulnerabilities involving Windows operating systems, we choose a UNIX type example to illustrate this type of attack. This is due to the relative complexity of the Win32 Applications Programming Interface (API) in comparison with the more simple Unix system calls. As a result of this complexity relatively few people have had a sufficient understanding of the intricacies of the Windows API at an assembly level in the past to exploit a buffer overflow in such a controlled fashion. Thus, while we can demonstrate that it is trivial to exploit a buffer overflow so as to make a Windows program or service crash (perhaps being effective as a Denial of Service attack, DOS), it is not a trivial case to exploit a buffer overflow in order to attain access and/or increased privileges on a Windows system. Thus, few examples exist which can easily demonstrate the general case of buffer overflow exploits within Windows software.

That said, buffer overflows on Windows systems are becoming widely exploited. We know that, buffer overflows existed in Unix-like operating systems for many years before they were well understood, documented and exploited. There currently exists a generalized framework for identifying and exploiting buffer overflows in Windows operating systems [45]. For a time it was believed that many buffer overflow vulnerabilities in Windows were 'purely theoretical'[1] - We have seen that with time, a little skill and some creativity that more and more people have made the theoretical practical.

1. This is a reference to a claim Microsoft made about a vulnerability in some of its software The l0pht then went and produced working exploit code.

4.1.3.3 Other Variants of the Buffer Overflow Exploit

All of these methods seek to alter the program's control flow so that the program will jump to the attack code. The basic method is to overflow a buffer that has weak or non-existent bounds checking on its input with a goal of corrupting the state of an adjacent part of the program's state, e.g. adjacent pointers, etc. By overflowing the buffer, the attacker can overwrite the adjacent program state with a near-arbitrary[1] sequence of bytes, resulting in an arbitrary bypass of C's type system[2] and the victim program's logic.

What we are interested in here is the kind of program state that the attacker's buffer overflow seeks to corrupt. In principle, the corrupted state can be any kind of state. For instance, the original Morris Worm [5], used a buffer overflow against the `fingerd` program to corrupt the name of a file that `fingerd` would execute. In practice, most buffer overflows found in 'the wild' seek to corrupt code pointers: program state that points at code. The distinguishing factors among buffer overflow attacks is the kind of state corrupted, and where in the memory layout the state is located.

Activation Records: As we have demonstrated, each time a function is called, it lays down an activation record on the stack [54] that includes, among other things, the return address that the program should jump to when the function exits. Attacks that corrupt activation record return addresses overflow automatic variables as detailed in figure 15. By corrupting the return address in the activation record, the attacker causes the program to jump to attack code when the victim function returns and dereferences the return address. This form of buffer overflow, we know from the previous section, is called a "stack smashing attack" [7] [46] [47] [48] [49] and constitute a majority of current buffer overflow attacks.

Function Pointers: '`void (* foo)()`' declares the variable foo which is of type 'pointer to function returning void'. Function pointers can be allocated anywhere (stack, heap, static data area) and so the attacker need only find an overflowable buffer adjacent to a function pointer in any of these areas and overflow it to change the function pointer. Some time later, when the program makes a call through this function pointer, it will instead jump to the attacker's desired location. An example of this kind of attack appeared in an attack against the superprobe program for Linux.

Longjump buffers: C includes a simple checkpoint/roll-back system called setjmp/longjmp. The idiom is to say "`setjmp(buffer)`" to checkpoint, and say "`longjmp(buffer)`" to go back to the check-point. However, if the attacker can corrupt the state of the buffer, then "`longjmp(buffer)`" will jump to the attacker's code instead. Like function pointers, longjump buffers can be allocated anywhere, so the attacker need only find an adjacent overflowable buffer. An example of this form of attack appeared

1. There are some bytes that are hard to inject, such as control characters and null bytes that have special meaning to I/O libraries, and thus may be filtered before they reach the program's memory.
2. Certainly an indication of the weakness of C's type system.

against Perl 5.003. The attack first corrupted a longjump buffer used to recover when buffer overflows are detected, and then induces the recovery mode, causing the Perl interpreter to jump to the attack code.

What are the different ways of combining code injection and control flow corruption techniques? The simplest and most common form of buffer overflow attack combines an injection technique with an activation record corruption in a single string. The attacker locates an overflowable automatic variable, feeds the program a large string that simultaneously overflows the buffer to change the activation record, and contains the injected attack code. This is the template for an attack outlined by Levy [7]. Because the C idiom of allocating a small local buffer to get user or parameter input is so common, there are a lot of instances of code vulnerable to this form of attack.

The injection and the corruption do not have to happen in one action. In the case of the frame pointer overwrite [50], the attacker can inject code into one buffer without overflowing it, and overflow a different buffer to corrupt a code pointer. This is typically done if the overflowable buffer does have bounds checking on it, but gets it wrong, so the buffer is only overflowable up to a certain number of bytes. The attacker does not have room to place code in the vulnerable buffer, so the code is simply inserted into a different buffer of sufficient size.

If the attacker is trying to use already-resident code instead of injecting it, they typically need to parameterize the code. For instance, there are code fragments in libc (linked to virtually every C program) that do '`exec(something)`' where 'something' is a parameter. The attacker then uses buffer overflows to corrupt the argument, and another buffer overflow to corrupt a code pointer to point into libc at the appropriate code fragment.

4.1.3.4 Attack Code

Although this thesis is concerned with the buffer overflow flaw in general and is specifically moving in the direction of finding these flaws at run time we would be careless to neglect the characteristics of the exploitcode used in attacks. The so called "arbitrary" code used in most buffer overflow exploits is known as shell code. Shell code is raw code in opcode format that will spawn a shell. Opcode is presented as strings of characters that represent format, register identifiers and machine instructions. On UNIX type systems the normal and most common type of shell code is a straight `/bin/sh execve()` call. This code calls `execve()` to execute /bin/sh which obviously spawns a shell. A key characteristic of shell code is it's complete lack of portability between systems. There are many papers on writing shell code all geared to exploiting buffer overflows primarily in UNIX systems [52], [53]. The real art of producing working shell code is crafting it in a way that avoids any binary zeroes in the code. For the most part, the buffers that we will be overflowing are `char []` buffers. As such, any null bytes located in the shell code will be considered as the end of input and the copy will be terminated.

The Windows API environment complicates the crafting of shell code and was responsible for the lag time in Windows system exploit rates. With the recent release of “plug and play” shell code [51] for the Windows system it is no surprise to see the acceleration of reported exploits increasing.

4.2 Discussion of the C and C++ Programming Language

The C programming language was devised in the early 1970s as a system implementation language for what was to become the Unix operating system. The first high level language implemented under the early UNIX systems was B. The B language, like it’s predecessor BCPL, was a typeless language. Being untyped meant that all data was considered as machine words and while being extremely simple it lead to many complications. As a result, a new typed language was developed which evolved into C [44].

BCPL, B, and C all fit firmly within the traditional procedural family and include other typical languages such as Fortran and Algol 60. The ‘C’ type languages are particularly oriented towards system programming as this was their heritage. They are small and compactly described, and are amenable to translation by simple compilers. They can be characterized as being `close to the machine' in that the abstractions they encompass are readily grounded in the native data types and operations supplied by conventional computer architectures. Another feature is the fact that they rely on standard library routines for input-output and other interactions with the operating system. Abstractions within the C language lie at a sufficiently high enough level that, with proper use, portability between machines can be achieved. For these reasons C has become one of the dominant languages of today.

4.2.1 The Standard Library

Within the C programming language there exists an entire class of ‘pre-packed’ software that is supplied for use by every C compiler. Collectively, this class of software is known as the C Standard Library. This library consists of categories of functions used for programming tasks we will want to perform and perform often. We access these library functions within our programs by including library header files. The preprocessor directive ‘`#include`’ performs this task. A preprocessor include directive causes the preprocessor to replace the directive with the contents of the specified file. Using the library header information, the preprocessor searches the for the place that contains the source files which are then included with the program at compile time. These header files are used to group functions together which perform similar or related tasks. As an example, the header file ***string.h*** provides access to a range of functions dealing with strings (i.e. arrays of characters). Similarly, the header file ***stdio.h*** provides access to a whole range of functions to do with inputting and outputting data from programs. An example of a programming task that we usually want to perform on a repetitive basis is getting text data into our

programs for subsequent processing, a name or password for instance. Using this example, we can use the ***gets()*** function which is supplied in the standard input/output library and accessed via the ***stdio.h*** header file:

> ***gets(s):*** reads the next input line of text into character arrays. It takes, as a single parameter, the start address of an user defined area of memory which one hopes is suitable to hold the input. The complete input line is read in and stored in the memory area as a null-terminated string. Our reasoning for using *gets( )* as an example will become painfully obvious as we proceed.

Another example of a common programming task is outputting text data from our programs. For this, we have used the ***printf()*** function, also accessed via the ***stdio.h*** header file:

> ***printf(char *format, ...):*** prints formatted output on the output device. The function takes a variable number of arguments. Format is a required argument which contains the text to be printed as well as any required conversion specifications. The remaining arguments specify the data to be converted to textual output for display.

The C Standard Library contains 18 standard headers that include hundreds of defined function routines. Most buffer overflow problems in C can be traced directly back to the standard C library. Perhaps the worst class of functions are the ones associated with the string operations that perform no argument checking (***strcpy, strcat, sprintf, gets***).

4.2.2 Unsafe String Primitives

Buffer overflows are so common because C is inherently unsafe. Array and pointer references are not automatically bounds-checked, so it is up to the programmer to do the checks themselves. More importantly, many of the string operations supported by the standard C library are unsafe. The programmer is responsible for checking that these operations cannot overflow buffers, and programmers often get those checks wrong or omit them altogether. As a result, we are left with many legacy applications that use the unsafe string primitives in a unsafe manner. This problem is made worse by the reuse of existing code libraries. In addition, programs written today still use the unsafe operations because they are familiar.

4.2.2.1 The ***gets()*** Function

This function reads a line of user-typed text from the standard input. It does not stop reading text until it sees an EOF character or a newline character. This represents the classic example of an unsafe string primitive. The ***gets()*** function performs no bounds checking at all. It is always possible to overflow any buffer using the ***gets()*** function.

4.2.2.2 The *str*()* Functions

The *str*()* functions include *strcpy()* and *strcat()*. The *strcpy()* function copies a source string into a destination buffer. No specific number of characters will be copied and it is this feature that leads to problems. The number of characters copied is directly dependent on how many characters are in the source string. If our source string happens to come from user input, and we don't explicitly restrict its size, we could potentially overflow the destination buffer. The *strcat()* function is very similar to *strcpy()*, except it concatenates one string onto the end of a buffer and again if we don't explicitly restrict its size, we could potentially overflow the destination buffer. Both of these functions have so called safe alternatives with *strncpy()* and *strncat()*. Unfortunately, programs that use just the “safe” subset of the C string API are not necessarily safe, because the “safe” string primitives have their own pitfalls. The *strncpy()* function may leave the target buffer unterminated. Using *strncpy()* has performance implications because it zero fills all the available space in the target buffer after the ‘\0’ terminator. For example, a *strncpy()* of a 13-byte buffer into a 2048-byte buffer overwrites the entire memory space. Both *strncpy()* and *strncat()* encourage off-by-one bugs. for example *strncat(dst, src, sizeof dst-strlen(dst)-1*) is the correct syntax while omitting the -1 results in an off-by-one error.

4.2.2.3 The Format Family Functions

A number of format functions are defined in the ANSI C definition which we will call the “format string”. A format function represents a special kind of ANSI C function. These functions take a variable number of arguments, one of which is from the so called format string. When a function from the format family evaluates the format string, it evaluates the extra parameters given to the function. The additional parameter is used as a conversion type function, and is used to represent primitive C data types in a string representation that is human readable. This family of functions are used in nearly every C program, to output information, print error messages or process strings. In general, a format string is an ASCII string that contains text and format parameters. As an example:

printf (“You have entered the following incorrect data: %s\n”, dataBuf);

The text to be printed is “You have entered the following incorrect data:”, followed by a format parameter ‘%s’, that is replaced with the user character data in dataBuf. Therefore the output looks like: “You have entered the following incorrect data: <*user data*>. Some format parameters include:

Table 2: ANSI C Format Parameters

PARAMETER	OUTPUT	PASSED AS
%d	decimal (int)	value
%n	number of bytes written so far, (* int)	reference
%s	string ((const) (unsigned) char *)	reference
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value

There are some basic format string functions (*`printf()`* and *`sprintf()`*) on which more complex functions are based. It is important to note that some of these are not part of the standard library but are widely available. The functions *`fprintf(), printf(), sprintf(), snprintf(), vfprintf(), vprintf(), vsprintf(), vsnprintf(),`* are all versatile functions used to convert the simple datatypes that exist within the C language to a string representation.

- fprintf - prints to a FILE stream
- printf - prints to the ‘stdout’ stream
- sprintf - prints into a string
- snprintf - prints into a string with length checking
- vfprintf - print to a FILE stream from a va_arg structure
- vprintf - prints to ‘stdout’ from a va_arg structure
- vsprintf - prints to a string from a va_arg structure
- vsnprintf - prints to a string with length checking from a va_arg structure

By using the format parameters shown above, the programmer can specify the format of the data being represented and process the resulting string output to stderr, stdout, syslog, etc. The format string controls both the behavior of the function as well as the specification of the type of parameters that should be printed. These parameters are saved on the stack (pushed) and are saved either directly (by value), or indirectly (by reference).

4.2.2.4 Stack Behavior During a Format String Function Call

The behavior of the format function is controlled by the format string that it calls as an argument. The function retrieves those data parameters requested by the format string from the stack. As an example:

```
printf ("%d has no address, number %d has: %08x\n", i, a, &a);
```

From within the printf function the stack looks like:

Figure 16: Stack Frame at a `printf()` Call

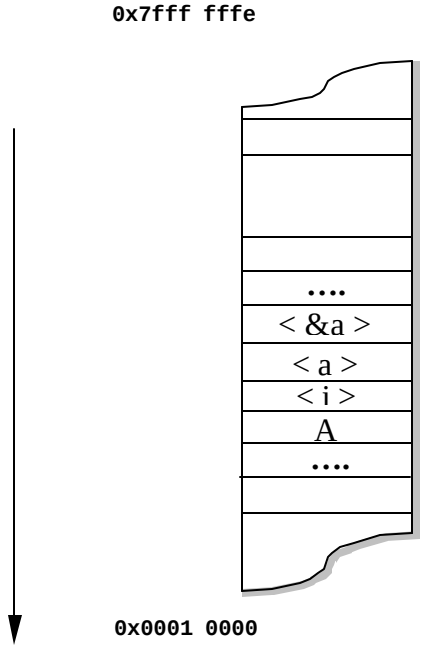


Table 3: Format String Stack Values

A	address of format string
i	value of the variable i
a	value of the variable a
&a	address of the variable i

The format function now parses the format string '`A`', by reading a character a time. If it is not '`%`', the character is copied to our output. In case it is, the character behind the '`%`' specifies the type of data parameter that should be evaluated. In addition, the string "%%" has a special meaning and is used to

print the escape character '*%*' itself. Every other parameter relates to data, which is located on the stack,. formatting it then storing it into a buffer.

These functions are often used to mimic the behavior of `strcpy()` in a fairly straight forward way. For this reason, it is just as easy to create a buffer overflow error to a program using `sprintf()` and `vsprintf()` as with `strcpy()` for instance. Since `sprintf()` can expand an arbitrary string using the '`%s`' format character, any call to `sprintf()` or `vsprintf()` which expands dynamic data into a fixed-size buffer has to be considered suspicious. As we will see the `sprinf()` function is often used in error messages where the error message is in the form of a string literal with user data being read by the format character. The user data is appended to the string literal and all placed into a destination buffer.

In addition, with the inclusion of the format parameter in this family of functions we create the potential for an entire new class of recently discovered vulnerabilities known as "Format String" vulnerabilities. In this class of vulnerabilities, the format parameter in combination with the user supplied input are implemented as two different types of information channels, the control channel and the data channel. The control channel is actively parsed while the data channel information is just copied. These two different types of information are merged into one using special escape characters or sequences to determine which channel is currently in a active state. The problem occurs when the format string is partially or completely undersupplied through incorrect programming practice. In the general case, when this occurs the data string passed for a straight copy is scanned for format characters. As we can control the behavior of our input and now insert the format characters we can potentially change memory address space by indirection using these function parameters. This presents us with an entire new method of exploitation. For this reason, when incorrectly used, the "format family" represents a class of truly dangerous functions. The exact mechanisms associated with the various format string exploits are outside the scope of the research however the reader is encouraged to read the excellent paper on the subject by Halvar Flake [61].

4.2.2.5 The *`scanf()` Family

The scanf family of functions is also poorly designed. In this case, destination buffers can be overflowed by dynamic data. As `*scanf()` parses data of dynamic origin into fixed buffers by using the '`%s`' format character, any `*scanf()` call which targets a fixed-size buffer with a '`%s`' format character is suspect and may point to a potential buffer overflow.

4.2.2.6 Other Functions

Other potential dangerous functions include `streadd()` and `stercpy()`. While not every compiler has support for these calls, programmers who have these functions available within their code

library should be cautious when using them. They have the same inherently dangerous features previously discussed. These functions translate a string that might possess unreadable characters into a representation that is printable. Another less common function is `strtrns()`, since many compilers do not support it. The function `strtrns()` takes, as its arguments, three strings and a destination buffer into which the resulting string is stored. The first string is basically copied into the destination buffer. A character gets copied from the first string to the destination buffer, unless that character appears within the second string. If this occurs, then a character at the same index in the third string gets substituted instead.

In summary, we have reviewed a subset of common C library functions that are susceptible to buffer overflow problems. There are certainly many more problematic functions and this is not intended to be a complete survey of every function within every common library. A partial listing of problematic calls is provided in the following table:

Table 4: C Library Functions Associated with Buffer Overflows

Function	Severity	Solution
gets	Extreme Risk	Use fgets(buf, size, stdin). This is almost always a big problem!
strcpy	High Risk	Use strncpy instead.
strcat	High Risk	Use strncat instead.
sprintf	High Risk	Use snprintf instead, or use specifiers to provide length precision.
scanf	High Risk	Use specifiers to provide length precision, or do your own parsing.
sscanf	High Risk	Use specifiers to provide length precision, or do your own parsing.
fscanf	High Risk	Use specifiers to provide length precision, or do your own parsing.
vfscanf	High Risk	Use specifiers to provide length precision, or do your own parsing.
vsprintf	High Risk	Use vsnprintf instead, or use specifiers to provide length precision
vscanf	High Risk	Use specifiers to provide length precision, or do your own parsing.
vsscanf	High Risk	Use specifiers to provide length precision, or do your own parsing.
streadd	High Risk	Verify you allocate 4 times the size of the source parameter as the size of the destination.

Table 4: C Library Functions Associated with Buffer Overflows

strecpy	High Risk	Verify you allocate 4 times the size of the source parameter as the size of the destination.
strtrns	Moderate Risk	Manually check to see that the destination is at least the same size as the source string.
realpath	High Risk (or less, depending on the implementation)	Allocate your buffer to be of size MAXPATHLEN. Also, manually check arguments to ensure the input argument is no larger than MAXPATHLEN.
syslog	High Risk (or less, depending on the implementation)	Before passing strings to this function, truncate all string inputs at a reasonable size.
getopt	High Risk (or less, depending on the implementation)	Before passing strings to this function, truncate all string inputs at a reasonable size.
getopt_long	High Risk (or less, depending on the implementation)	Before passing strings to this function, truncate all string inputs at a reasonable size.
getpass	High Risk (or less, depending on the implementation)	Before passing strings to this function, truncate all string inputs at a reasonable size.
getchar	Moderate Risk	If using this function in a loop, make sure to check your buffer boundaries.
fgetc	Moderate Risk	If using this function in a loop, make sure to check your buffer boundaries.
getc	Moderate Risk	If using this function in a loop, make sure to check your buffer boundaries.
read	Moderate Risk	If using this function in a loop, make sure to check your buffer boundaries.
bcopy	Low Risk	Verify that your destination buffer is as large as you say it is.
fgets	Low Risk	Verify that your destination buffer is as large as you say it is.
memcpy	Low Risk	Verify that your destination buffer is as large as you say it is.
snprintf	Low Risk	Verify that your destination buffer is as large as you say it is.

Chapter 5

Binary Reverse Engineering to Locate Security Flaws

Science is a way of trying not to fool yourself.

- Richard P. Feynmann

5.1 Introduction

This is a research that is motivated by problems of old technology, and by problems of new technology. In the past program maintenance has always depended on reverse engineering to some extent, especially when one considers the huge amounts of legacy code without detailed documentation. As the maintenance of legacy code continues there is a new problem which effects code that has no documentation available to the user and that is security. Security, until very recently, was taken for granted in the development of most software designated for mainstream usage. As this code is predominantly third party and propriety, we seek a means to satisfy an important curiosity. How secure is our program? Without the source code to serve as our documentation, do we trust the vendor? Experience has shown that a decision to trust would be most unwise. It is our aim to investigate a method that may provide an automatic detection of not-so-obvious security flaws in a binary file and borrowing from those who maintain legacy code systems, we will use reverse engineering. We believe that these flaws exist as patterns which can be recognized at the assembly level. The detection of these patterns requires the investigation of a range of subjects, from the basics of reverse engineering, to the behavior of higher level code constructs, to the representation of patterns at the assembly level as well as the tools and methods for detection. In broad terms, reverse engineering can be a daunting undertaking that spans across all disciplines of computer science. The best approach, like eating an elephant, is a small byte at a time.

5.2 Binary Image Basics

Before considering the reverse engineering of our executable image to find suspicious high level code constructs, the relations between the static binary code of the program and the actions performed at run-time to actually implement the program are presented. The representation of objects in a binary program are dependent on compiler design and the elementary data types such as characters, integers and reals. These are often represented by an equivalent data object that resides within the architecture of the machine (i.e. a fixed size number of bytes). This can be contrasted with aggregate objects such as structures, strings and arrays that can be represented in various different ways.

In order to discuss the concepts presented in this chapter in a manner that is unambiguous to the reader we will establish the following nomenclature. The word subroutine will be used as a generic term to denote either a function or a procedure. When we are certain as to what the subroutine really represents we will use the term ‘procedure’. This for a subroutine that does not return a value. We will use the term ‘function’ for a subroutine that returns a value. Likewise we use the terms binary executable, binary image, and binary file interchangeably as all representing a compiled ready to run program.

5.2.1 Compiling, Linking and Libraries

One major stumbling block common in the disassembly of programs written in modern high level languages, such as C, is the time wasted to identify then isolate library functions. We consider this time as lost because it does not bring us closer to gaining knowledge of the targeted application. It can be viewed as only a mandatory step that allows us to continue our analysis of the program in a effort to reach the more meaningful algorithms. In addition, it is an unfortunate fact to the reverse engineer that this process has to be repeated for each and every new disassembly. It can be shown how the disassembly of even the simplest programs can result in the generation numerous superfluous subroutines that result from the linkage of object and library files. A general understanding of compiling, linking and loading is important to the reverse engineer when analyzing a disassembled executable image as many subroutines can be discounted as a consequence of these processes.

5.2.1.1 Compiling

Simply stated compiling consists of translating human readable C source code into an assembly file. The compiler accomplishes this by converting source files into object files when building the executable image. Each object file contains instructions in machine code that correspond to statements in the high level source programming language. In general, these object files are broken into a collection of distinctive sections each corresponding to different parts of the source program. When the compiler is invoked, it scans the program for simple syntax errors. If it finds problems, it interrupts the build and typically lists each problem it finds along with a terse message and the line number associated with the syntax error. One of the import things to note is that behavior of the program can be different when one uses different compilers. The implications of this characteristic are that different compilers produce different executable images on any given machine. There are four basic steps to the compilation of C code. They are commonly described as preprocessing, compilation, assembly, and linking.

1) Preprocessing occurs during the first pass of any C compilation. It processes any include-files, conditional compilation instructions and macros.

2) Compilation is actually performed during the second pass. It utilizes the output of the pre-

processor, and the source code, then generates assembler source code.

3) Assembly is the third stage of compilation. It takes the assembly source code produced during compilation and creates an assembly listing with offsets. This assembler output is stored within an object file.

4) Linking is the final stage of compilation. It receives one or more object files and or libraries as input then combines them to produce a single executable file.

5.2.1.2 Linking

To build an executable file, the linker is tasked with the collection of all object files and libraries. The primary function of the linker is to bind symbolic names to memory addresses. The process of linking involves scanning the compiler output, the object files then concatenating all the object file sections to form one large file (data sections of all object files are concatenated, text sections are concatenated, and so on). It then scans the resulting file a second time to bind the symbol names to real memory addresses.

5.2.1.3 Libraries

When considering binary disassembly it is to our advantage to have a basic foundation in the methods associated with libraries. Any disassembled binary will have artifacts that reflect the library file representation. It is therefore important to the reverse engineer to have insight into this representation. There are two distinct types of library files and it is important that any disassembler we decide to use can recognize the difference and produce a disassembly that makes sense.

Within any higher level programming language there are many useful predefined routines, or functions, that are used repeatedly within the programming environment. To improve modularity and reusability one is able to group these commonly used functions into files called libraries. As we demonstrated in the previous chapter most, if not all, our dangerous C functions reside within these libraries. A library contains a set of object files used to implement subroutines and functions that in turn can be linked either statically or dynamically with other object files to produce a complete executable program. Static library files are linked at compile time with user defined routines (object files) to build complete executable programs or libraries can be linked dynamically at run time.

When a static library is referenced it is included during program linking. At compile time the linker makes a pass through the library file then adds all the code and data corresponding to the symbols used in the source program. Therefore whenever the linker includes a static library in a program, it copies data from the library to the target program where it remains as part of the executable image. Static libraries are easy to create and implement, but they come at a cost with a number of issues associated with resource utilization and software maintenance. Whenever we copy the contents of a static library into the target program it wastes memory and disk space. For example, if a copy of the C library was included in

every executable on a system the resident disk space of these programs would increase dramatically, and when active, they would each store their copies of the library functions within system memory, a considerable waste of resource. As far as software maintenance is concerned, whenever a change to a static library is required, everything linked using that library must be rebuilt in order for the changes to propagate through all released code.

Current practice has greatly reduced the maintenance and resource problems of static libraries. This is accomplished by using shared libraries or dynamic link libraries (DLLs). The main difference between static and dynamically linked libraries is that using dynamically linked libraries delays the actual task of linking to runtime, where it is accomplished by a dynamic linker-loader. Using this technique, a program and its libraries remain as separate entities until the program actually runs. This means that if an error, such as a buffer overflow flaw, is found in a commonly used library, the dll can be corrected then substituted for the original one and applications that formerly exhibited the flaw will be repaired without the need to re-compile and re-link applications referring to the library in question. In addition, system level optimizations are possible. If many programs are running and a large portion of them include the same library (e.g., the C library for example), the operating system can then load a single instance of the library's instructions into physical memory. This significantly reduces the use of memory resources and improves overall system performance.

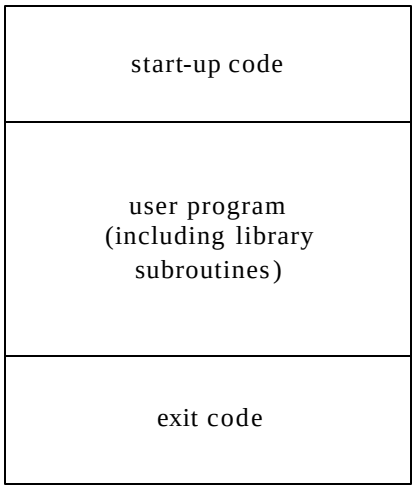
5.2.2 Loading

All high-level language programs are composed of one or more subroutines and are referred to as user subroutines. The corresponding binary program is composed of all the user subroutines, any library routines that were called by the user program, and any other subroutines required to provide support for the compiler and linked in by the linker at run-time. When a program that is linked with shared libraries runs, program execution does not begin immediately with that program's first statement. Instead, the operating system loads the required environment that includes execution of the dynamic linker. The dynamic linker then scans the list of library names embedded within the executable.

The general format of the binary image of a program is shown in Figure 17. The execution of a binary file at run time can be summarized as follows:

- The program begins by calling the compiler start-up subroutines that create the environment for the compiler.
- Next the subroutine for the user's main program executes, which invokes library routines linked in by the linker.
- Execution is finalized by a series of compiler subroutines that restore the original state of the machine before program termination.

Figure 17: General Format of a Binary Program



5.3 A Unique Executable File Format-Win32 PE

We are concerned with the type of programming flaws that manifest themselves as a buffer overflow. In addition we are concerned with third party programs and applications that operate within the modern enterprise environment. For this reason we look at with special interest, the family of software products that surround the Windows operating system. Common to these products is the Win32 portable executable file (PE) format. When the reverse engineer analyzes a Win32 disassembly a general understanding of the PE format is important.

5.3.1 PE File Background

Immediately prior to the PC's acceptance as 'the' enterprise computing tool, the architecture commonly used across industry were minicomputers or mainframes that used VAX VMS or UNIX operating systems. This is where the expertise was, so it is no secret that when Microsoft came along much of this heritage technology went into the Windows operating system. When the time came to design Windows NT, it made sense to use existing technology and incorporate previously written and tested tools in a effort to minimize things like the OS bootstrap time. The COFF (Common Object File Format) became the executable and object module format that the heritage technology tools produced. Artifacts of this past technology can be seen such as things within the COFF as references in octal format. A modern operating system such as Windows NT has little use for octal references, so while the basic COFF format was a good as a starting point, it needed to be further extended. As a result the COFF was updated to reflect cur-

rent requirements and ultimately became what is known as the Portable Executable format. This portability is maintained across all implementations of Windows NT on various platforms with x86, MIPS, Alpha, and so on all using the same executable format.

5.3.2 PE File Layout

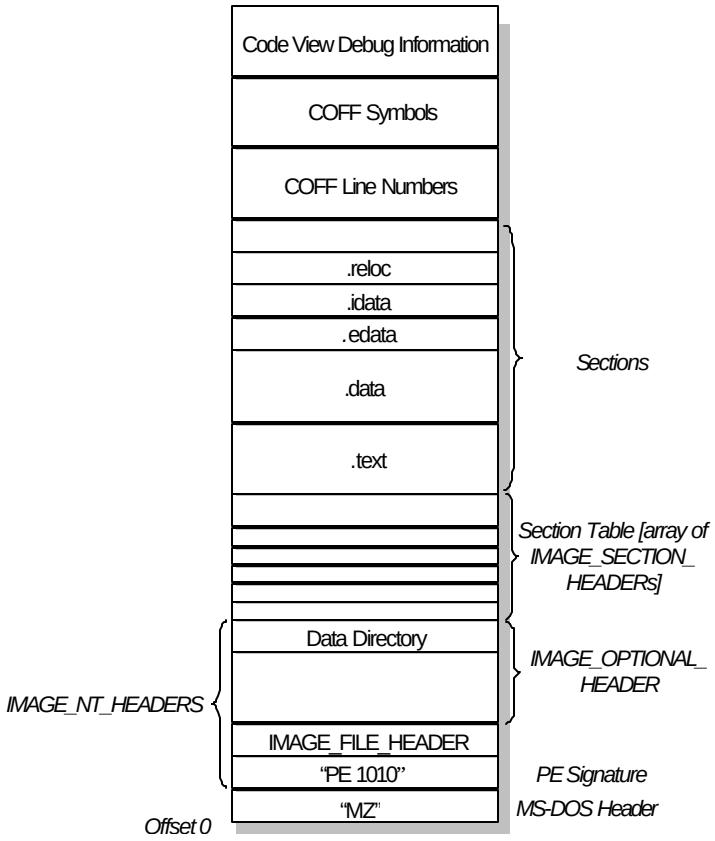
There are a few concepts which are fundamental to the layout and design of a PE file. We can use the term “module” to define all the things that are loaded into memory when one runs a executable file or a DLL. These ‘things’ include the code and data that the program uses directly as well as the all the data structures that are used by Windows to determine where in memory the code and date are to be located.

5.3.2.1 PE File Header

These data structures are defined in the Win32 PE header file. Common with all other executable file formats, the PE file has, at a reserved location, a collection of fields that describe the appearance of the rest of the file. This is known as the header and the information it contains includes the sizes and locations of the code and data areas, the initial stack size, the type operating system the file is intended for and other pieces of information that are vital to the loading and execution of the module.

Counterintuitively, the main header isn't located at the very beginning of the file, a trait that the PE format shares with other Microsoft executable formats. Within a typical PE file the header is preceded by a MS-DOS stub program which occupies the first few hundred bytes. This small program prints “This program cannot be run in MS-DOS mode.” If a Win32 program is run within an environment that doesn't support Win32, one will see this informative error message. After the Win32 loader maps a PE file in memory, the first byte of the mapped file is the first byte of this MS-DOS stub.

Figure 18: The PE File Format[1]



5.3.2.2 PE File Concepts

A key concept in the understanding of the PE file is that the executable image on disk mirrors what the module will look like in memory after Windows has loaded it. In other words, the Windows loader doesn't expend much effort when creating a process from the file on disk. A memory-mapped file mechanism is used by the loader to map the appropriate sections of the file into the virtual address space. In general, the PE file is essentially mapped into place in one piece, followed by a small amount of work to link it up to the associated DLL's. Ease of loading applies to any PE-format file, including PE-format DLLs. Once the module has been loaded, it's behavior is consistent with any other memory-mapped file. For Win32, all one needs to know is where in memory the loader mapped the file as all the memory used by the module for data, code, resources, import tables, export tables, and other required module data struc-

1. From: "*Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*."; Matt Pietrek; March 1994

tures is in one contiguous block. By following pointers that are stored as part of the image, one can easily find all the various pieces of the module.

A second concept one should be acquainted with is the Relative Virtual Address (RVA). Many of the fields within PE files are given in terms of RVAs. An RVA is nothing more than the offset of some item relative to where in memory the file is mapped. As an example, if the loader maps a PE file into memory starting at address 0x20000 within virtual address space and a certain table within the image starts at address 0x20464, then the table's RVA is given as 0x464. Conversely, simply add the RVA to the modules base address to convert an RVA into a usable pointer. With this concept in mind, one can appreciate the importance of the base address or the starting address of a memory-mapped EXE or DLL in Win32.

The final concept that one should be familiar with when working with PE files is sections. A section within a PE file can be considered as equivalent to the resources or a segment in an 16-bit file. Sections can contain either data or code. Sections are blocks of contiguous memory and unlike segments have no size constraints. Some sections contain data or code that your program declared for direct use, while other data sections are created by the linker working with libraries, and contain information necessary for use by the operating system.

5.3.2.3 Sections

The content of the PE file that we are most concerned with is divided into blocks called sections. The PE file section represents the code and/or the data representation. The representation for code is just code, however there are multiple representations for data. When one considers data one usually thinks in terms of read/write program data (such as global variables). Within the PE file there are other types of data as well which support the Win32 environment. The other types of data found in the PE sections include resources, and relocations as well as the API import and export tables. Common in-memory attributes are associated with each section and include not only basic read/write or read-only attributes but whether the data located within the section is shared between all processes that use the executable. In general all code or data within a section shares a common logical relationship and there are usually at least two sections, one for code the other for data, within a PE file.

5.3.2.4 Offsets and Alignment

As we will be analyzing disassembled PE files we must have a knowledge on how the file as well as the code and data sections are loaded and aligned in memory. The following represents a very simplified summarization of the loading of a PE file into memory that is appropriate for our discussion.

- When the PE file is invoked, the PE loader first scans the DOS MZ header for the offset value of the PE header. When found, it jumps to the PE header.
- The PE loader then checks to verify that the PE header is valid. If so, it jumps to the end of the header.
- The section table immediately follows the PE header. The PE header scans the information associated with the sections which are mapped into memory using file mapping. It also assigns to each section the attributes that are specified in the section table.
- After the memory mapping of the PE file is complete, the PE loader works elements such as the import table that are associated with the logical portions of the PE file.

The PE file header specifies two values associated with alignment of PE file sections, one value within the disk file and the other value within memory. It is important to note that each of these values can differ. The base of each section starts at an offset that is a given multiple of the alignment value. As an example, in the PE file, a typical alignment value would be 0x200. Therefore, every section mapped into memory starts at a file offset that's a multiple of 0x200. In addition, every section mapped into memory will always start on at least a page boundary. In other words, when a PE section is mapped into memory, the first byte of each section corresponds to a memory page. When considering the x86 architecture, pages are aligned at 4KB boundaries.

The reverse engineer is most concerned with the alignment and offset of the .text and .data sections of the PE file. The .text section is located at offset 0x400 within the PE file and will be located 0x1000 bytes above the KERNEL32 load address in memory. The .data section is located at file offset 0x74C00 and will be aligned 0x76000 bytes above the KERNEL32 load address in memory.

The preceding discussion can not, by any sense of the imagination, be seen as complete nor can it convey the true depth of the subject itself. It was meant to provide the minimal background in the format and behavior of the executable files that we will be working with. A wealth of knowledge exists on this subject as represented by a rich set of reports, articles and books that all await the more adventurous reader.

5.4 Reverse Engineering

Reverse engineering of software systems has been defined as the analysis of a subject system to identify the system’s components and their interrelationships, and to create a representation of the system in another form or at a higher level of abstraction [63]. The aim of reverse engineering of software systems is to gain a level of understanding of the system and its structure for the purposes, in the broad sense of the word, of maintenance. We use the term broadly because maintenance in the context of reverse engineering has meant everything from documenting legacy systems and patching bugs in existing programs to cracking copy protection schemes, and outright intellectual proprietary theft. The case of software

reverse engineering is unlike the case of hardware reverse engineering where the system is disassembled and analyzed in a effort to make a duplicate copy of it. Most of the reverse engineering environments reported in the literature either concentrate on the removal of software copy protection, with the generation of graphical representations, the recovery of high-level specifications based on missing original high-level source code or design documentation.

In this thesis, we present a partially automated reverse engineering environment for the recovery of targeted information from binary code (i.e. executable or application code)[70][71][72]. To provide a workable reverse engineering environment we are mainly concerned with disassembly technology and the types of tools that are available. In general a disassembler is a program that reads an executable program (binary file) and translates it into an equivalent assembler program.

When reverse engineering a binary file, the program to be translated is any arbitrary program compiled from any high-level language, in our case C. In broad terms, the disassembler parses the binary image of the program then translates it to assembler or some representation that is equivalent. Almost all disassemblers follow a conventional approach:

- Machine instructions are parsed starting at the entry point of the program then following all paths from this point.
- Whenever a transfer of control is reached, the new path is followed until an end of path instruction (e.g. a return of subroutine) is met.
- The target address of indirect and indexed jumps or calls is, in many times, hard to determine in a static disassembler. This is because these instructions do not provide complete information on the range of possible values at such an instruction. In these cases backwards slicing [64][65] analysis is sometimes capable of determining the range of values available at the indirect or indexed instruction. Using this method it possible to continue parsing instructions along that path(s).

5.4.1 Software Reverse Engineering - A Dispiriting Adventure

Reverse engineering within the traditional engineering disciplines involves end products which were designed under the limitations of the physical world. Within these disciplines, real world behavior is described by the language of physics that includes basic dimensions (length, time, charge, and mass) that are applied to a generic rule set to describe the physical universe. By specializing the equations used to calculate the concepts associated with natural phenomena, engineers, in the traditional sense, are able to design end products for real world application.These engineers are able to face the disorder of the real world without changing the concepts themselves. The primary means by which this is accomplished is a principled method of understanding when close enough is good enough in approximating real world events. In effect when we attempt to reverse engineer a traditional product we have formal, well under-

stood concepts that act as boundaries to our set of possible solutions. In other words, we are constrained by the laws of physics.

5.4.1.1 The Human Element

Software engineering is entirely a human activity. A program is created and intimately associated with the cognitive abstractions used by the programmer and can be defined in terms of behavioral characteristics.When dealing with cognitive or behavioral activities we have no physical laws that act as a boundary to our set of possible solutions. In our higher level programming languages we have made an attempt to add a universal understanding to the semantics of the language itself but this is not enough to describe the complexity issues that one will encounter when attempting the reverse engineering of software. This is a primary reason that methods of internal program documentation have been developed and have gained wide acceptance. The source code, depending on the language, can communicate the computational intent to a large extent and more importantly flow of execution but it cannot communicate the precise conceptual intent. In addition it is very poor at telling us how conceptual intent is related to the objectives of the software system in the context of a domain. If we remove the human element from the system, the static source code of a given higher level language without internal documentation does not communicate the programmers intent in a straight forward manner to other human beings.

With a binary executable the picture is even bleaker. With a binary file our only option in reverse engineering the product is to disassemble the file into assembly language instructions. In this situation the program has to be understood in the absence of the person who has created it and with no documentation internal or semantic. It gets worse, in the absence of any semantic information we are unable to make any assumptions associated with flow of execution. This is the fundamental reason why understanding a disassembled binary file is so difficult. Of course other reasons why reverse engineering in general is so difficult include large size, enormous complexity and the fact that product design documentation is either inadequate, non-existing or both.

5.4.2 Analysis Methods in Binary Disassembly

When considering binary disassembly in order to recover targeted high level programming information one should have a least a nominal understanding of the way disassemblers work. Currently available tools used for binary disassembly utilize the following techniques to recover information and present it in a human understandable form.

5.4.2.1 Analysis of Data Flow

Analysis of data flow is used to recover high-level language statements and expressions (other than so-called control transfer statements), function return values, actual parameters, and to remove any hardware references from the assembly code, such as pipeline references, stack references and registers. This analysis aims at techniques that are machine-independent for solving this problem and is used primarily on CISC and RISC architectures.

5.4.2.2 Analysis of Control Flow

Analysis of control flow is used to recover control flow structure information, such as conditional statements and loops, as well as their level of nesting. Recovery of the control structure information in a program is a problem associated with graph theory. In assembly code, control transfer is performed using conditional or unconditional jumps, procedure calls and returns. It is a known fact that potentially all jumps (conditional and unconditional) can be implemented by goto statements. This is one of the main reasons we moved away from BASIC and FORTRAN towards C++ and JAVA as a high level programming language. With control flow analysis we can determine whether such jumps are redundant, induce a loop, or will require goto statements. The aim is to minimize the number of goto statements thereby reducing the paths used throughout the code without increasing the complexity of the program. Analysis of flow control acts to improve the quality of the generated code.

5.4.2.3 Analysis of Type

Analysis of type analysis is used to recover high-level type information for function return types, actual and formal parameter types, and variables. Type analysis deals specifically with the recovery of the data type information of a given high level language. This is traditionally an area that has been studied and reported in the literature as being associated with the functional and untyped object-oriented languages. In these languages, the data type associated with a variable is inferred based on the variables context of use. It is important to note that type analysis has not been studied to any great degree within the context of translations from assembly code to a higher level language. It can be demonstrated that it is clearly desirable to regenerate high level language

code that makes use of base data types (e.g. byte, char, integer, and real) in the process. This can be extended to the more complex compound types (e.g.array, structure or record, and class) and is even more important when one considers regeneration of assembly to the object oriented languages. As we will demonstrate, this represents a weakness in our proposed method of binary disassembly. Many applications store data in large structures which are passed around between functions. The information about the layout of these structures is lost during the compilation.

5.4.3 Limitations

The main problem one must face with binary disassembly derives from the representation of instructions (code) and data in the Von Neumann architecture: they are indistinguishable. Thus, instructions can be located at random in between data. This is especially a concern with the many implementations of case or indexed jump tables. The nature of this representation along with self modifying code practices and idioms make it hard to disassemble an executable program without error. It is interesting to note that the separation of data and instructions is a problem that is unsolvable in general. In summary, if one could describe an algorithm to determine such separation, this algorithm would also solve the halting problem [67]. A second problem is the large amount of subroutines introduced by the inherent mechanisms of the compiler and the linker. These subroutines are bound in the executable program at compile time. The compiler produces start-up subroutines that establish the execution environment, and runtime support routines that are produced whenever required. These routines are usually written in assembler and in most cases cannot be translated into a higher-level representation. A third problem lies within the operating system itself. Some operating systems, such as Windows NT, have library routines that are maintained in separate files and linked dynamically at run time. This approach can help the reverse engineer as the library routines are referenced through a dynamic linkage table within the executable image. In operating systems that do not provide this type of mechanism to share libraries, executable programs are self-contained (statically linked) and library routines are bound into each executable image. Statically linked library routines are written in either the language the compiler was written in or assembler. This has major implications for the reverse engineer as the executable program contains not only the routines written by the programmer, but all the other static library routines linked in by the linker.

A thread common to every high level language program is the great number of standard library functions that are used, sometimes even up to 95% of all the functions called are standard functions. To give the reader a feel for the magnitude of this problem, the ubiquitous “hello world” program compiled in Borland C generates:

- Library functions - 58
- Function main() - 1

When one disassembles this example, one is only interested in the main() subroutine and not the other 58 or so subroutines. Of course, this example is an artificial one but it is a fact that real life programs contain 50% library functions on average. This is the primary reason that the reverse engineer who uses a disassembler is forced to waste better than half of his time isolating those library functions.

5.5 Legal Considerations

Several questions have been raised in the last years regarding the legality of reverse engineering. A debate between supporters of reverse engineering who claim fair competition is possible with the use of decompilation tools, and the opponents of reverse engineering who claim copyright is infringed by reverse engineering, is currently being held; in fact, this debate has been reported in the literature since 1984 [68]. The law in different countries is being modified to determine in which cases reverse engineering is lawful. At present, commercial software is being sold with software agreements that ban the user from disassembling or reverse engineering the product. For example, part of the End-User License Agreement (EULA) for Microsoft Office 2000 reads like this:

> *"Limitations on Reverse Engineering, Decompilation, and Disassembly. You may not reverse engineer, decompile, or disassemble the SOFTWARE PRODUCT, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation"*.

The final form of the Digital Millennium Copyright Act (DMCA) includes exceptions to copyright protections, such as the EULA, for the following reasons:

- Reverse engineering for interoperability
- Encryption research
- Security testing

It is not the purpose of this chapter to debate the legal implications of reverse engineering. This topic is not further covered in this thesis.

5.6 Recovery of High Level Abstractions From the Binary

Reverse engineering has its origins within the established engineering disciplines. When one considers the traditional model of reverse engineering a picture of the analysis of hardware for commercial or military advantage emerges. The concept behind reverse engineering is to deduce the original design decisions from the end product. This is often performed with little or no additional knowledge about the design requirements, design process, and manufacturing techniques involved in the original production.

The same approach can be applied to proprietary software systems. That is, without prior insight into the design process, the end product, in our case a binary executable, can be analyzed in order to capture higher level design of the product. As with the traditional model, software reverse engineering can also be used for industrial or defense ends. It can also be used as a tool to recover incorrect and incomplete artifacts within the original source or recover otherwise unavailable documentation. Broadly speaking,

software reverse engineering is a field of research that is devoted to developing methodologies and tools to aid in the understanding and management of released software systems. We are interested in the more practical aspects of software reverse engineering in that we wish to reverse engineer targeted products to recover a targeted defect.

The recovery of high-level abstractions from machine and assembly code is a field of study that has not been widely researched in recent years. This is partly due to the complexity of the problem and partly due to the negative connotation of software disassembly in general. Most of the techniques available in this field are widely published as tutorials associated with the cracking of software copy protection.

In the context of identifying buffer overflows, we have demonstrated that many organizations rely on proprietary closed source software. Enterprise class applications and operating systems are being delivered with little apparent review of the source code for security vulnerabilities. The end users have no access to the source code for their independent assessment or security audit. These facts point to the need for research into ways of translating machine and assembly code to the high-level language constructs which lead to security vulnerabilities. Work associated with the reassembling of data structures has been performed Halvar Flake one of the underground experts involved with binary auditing techniques[57].

Any reverse engineering effort typically consists of gathering the best understanding of the target software system that is possible. It starts with existing information such as marketing information, trial or beta copies of the software, user manuals, on-line help and news group postings. Depending on the copyright or security concerns, where any detailed design information is closely held, social engineering techniques may become an option.

Although many sources of information may be available, the actual code, the ultimate description of the current state of the system is, for all intents and purposes, unavailable. Hence the crux of the reverse engineering process is the problem of program understanding without the program. A difficult but not hopeless situation.

Chapter 6

A Novel Approach to The Discovery of Buffer Overflows in a Binary Image

What I can not create I can not understand.
- Richard Feynman

6.1 Introduction

In this chapter I describe the implementation of a novel method of locating an instance of a buffer overflow within a binary image. As the binary image, or executable, represents the deliverable product from the software vendor to the consumer, the technique presented is applicable to all third party proprietary software sold as a executable binary file.

As discussed in chapter 1, the software environment at the enterprise level is dominated by x86 architectures running Microsoft products. For this reason I have implemented the technique using a Windows OS platform. This allows for extension across most all applications running within the Windows environment.

The approach to the problem was a simple one and was grounded on the following observation. That is, all buffers that are created on the stack have a unique sequence of operations that create the stack space and as a result can be identified by this unique signature. In addition, all calls to library functions will have a unique signature. This includes the set of so-called “dangerous C functions” that were discussed in chapter 4. These signatures, represented as a series of assembly instructions, are created at compile time and therefore become a part of the binary image. By disassembling the executable file we propose that one might be able to locate certain types of buffer overflow vulnerabilities.

Based on the discussions in the previous chapters, the following goals have been identified to allow us to evaluate the success of our technique.

- Fast, as the original premise was to find a single instance of a buffer overflow vulnerability as rapidly as possible.
- The accurate identification of potential overflow conditions while rejecting false positives
- Demonstrate scalability from simple test programs to enterprise class software applications.

6.2 Tools

The tools we used in this thesis included compilers, disassemblers and text editors. We will capture the version used as well as a description to allow for an exact record of the configuration used.

6.2.1 Disassembler

This tool was the key in enabling the entire effort behind this thesis and as such deserves special attention. In general, a disassembler will take a binary executable file as an input then convert it into readable assembly language. The better tools will also include additional information like cross-references for subroutine calls and jumps. String literals, that exist within the high level program, will be shown as part of the output. The best disassemblers will also maintain a reference listing of API calls (e.g. the Win32 API associated with Windows). Whenever the application calls one of the native API routines, it will be displayed along with the right parameters that are passed to the routine. The output of a good disassembler will make our life much easier by giving us enough information to allow for the retrieval of program logic. Popular disassemblers include:

- Wdasm32 Windows Disassembler: This is a shareware class Windows program for disassembling Win32 programs. It is a decent disassembler that is easier to use than most and this feature makes it a good choice for beginners. The distribution includes a program called hilevel. This program can transform the assembler output into a structured format that includes definition of local variables and procedures. The output is what you see is what you get which presents a severe limitation for our purposes.
- Sourcer by V Communications: This is commercial program used for disassembling x86 binaries (PE, NE and EXE). Sourcer automatically detects code and data fragments and provides fairly good output.

These are characteristic of the class of disassemblers in general. The output is entirely dependent on the various algorithms internal to the disassembler and represents a best guess analysis of the binary image.

6.2.1.1 IDA (Interactive Disassembler) Pro

IDA, written by Ilfak Guilfanov, is a commercial program used for disassembling a wide selection of file types supporting the architectures of over 30 microprocessors. Supported binary file formats include: EXE, PE, COFF, NE, LX, LE, and OMF. When it comes to reverse engineering a binary or library file, IDA Pro is the most advanced tool available to the consumer. IDA has seen widespread use by intelligence agencies, security analysts, hackers as well as by Fortune 500 companies.

There are several reasons why this tool has enabled this research. IDA's internal FLIRT (Fast Library Identification and Recognition Technology) module identifies statically linked library functions from most of the common compilers. In chapter 5 we discussed the difference between static and dynamic library calls and therefore recognize the significance of this feature. This means that all of our dangerous function calls will be identified when statically linked providing a huge savings over hand auditing. IDA is interactive and allows the user, as logic is revealed, to modify elements within the disassembly then propagate these changes back through the entire disassembled file. In other words, the human is in charge

of the disassembly and is able to use intuitive adjustments to the output. IDA includes a powerful scripting language, similar to C, which will allow us to automate the search for buffer overflow vulnerabilities. In addition a very effective plug-in interface is available for full fledged C programs. For this thesis we will be using IDA Pro version 4.17.

6.2.2 Other Tools

The configuration that we used to perform our testing was accomplished on a single stand-alone machine (Pentium II 400MHz; 128MRam) running Windows 98 Second edition. A client server relationship was established using Microsoft Personal Web Server 2.0 along with a telnet daemon (Microsoft Telnet 1.0). Other tools that were required include:

- A C compiler; Microsoft Developer Studio 97; Visual C++ 5.0
- Text editor, Microsoft Notepad

6.3 Approach

What if it were possible to identify buffer overflow vulnerabilities within a binary image? Where would this leave you? You would know that the possibility for a buffer overflow existed within the binary file and little else. For instance, how would you trace it back to a point in the actual running program where the user provides input, what I call the program entry point? It is possible to trace back through the disassembly by hand to a look-up table that references commands used within the application itself. This is what “Barnaby Jack” demonstrated in his paper *Win32 Buffer Overflows (Location, Exploitation and Prevention)*[58]. This a tedious process to say the least, with no guarantee of success. The reverse engineer could easily spend several days tracing code through paths leading nowhere. Our original concept was to find an instance of a buffer overflow vulnerability as fast as possible and a hand audit of a disassembly is not what we had in mind. Lets stop and think for a minute about some of the so-called dangerous C functions. We have a disassembler that can identify these calls within the disassembly which is a great advantage.

6.3.1 The *gets()* function

The gets() function, if one can find it still being used, will not provide us much in the way of insight into where in the running application the user input is passed to the function. We can demonstrate this with the following simple program:

```
/* gets example */
#include <stdio.h>

int main()
{
  char string [256];
  printf ("Insert your full address: ");
  gets (string);
  printf ("Your address is: %s\n",string);
  return 0;
}
```

gets() sample program

The disassembly really leaves us without clues as to where in the application the user input would be provided. A target buffer of 256 bytes is pushed onto the stack as var_100 however any other information as to the context of the function within the running program will require hand auditing of the disassembly. The disassembly of the *gets()* sample program is provided below:

```
add     esp, 4
lea     eax, [ebp+var_100] //target buffer 256 bytes
push    eax
call    _gets
add     esp, 4
```

Disassembly of *gets()* function:

6.3.2 The *strn*()*

The *strn*()* family of functions for the most part manipulate the contents of two buffers, a source and a destination. Poor implementation in the handling of two buffers may result in a security flaw, however within the disassembly there will be little to point us to a entry point in the program. For example

lets look at a simple *strcpy()* program. This program copies a source buffer into a destination buffer of bytes. The example is provided below:

```
/* strcpy example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str1[]="Sample string";
  char str2[40];
  char str3[40];
  strcpy (str2,str1);
  strcpy (str3,"copy successful");
  printf ("str1: %s\nstr2: %s\nstr3: %s\n",str1,str2,str3);
  return 0;
}
```

Simple *strcpy()* program.

Again the disassembly really leaves us without clues as to where in the application the user input would be provided. A target buffer of 256 bytes is pushed onto the stack as var_100 however any other information as to the context of the function within the running program will require hand auditing of the disassembly. The disassembly of the *strcpy()* program sample is provided below:

```
lea     ecx, [ebp+var_10] // source
push    ecx               ; const char *
lea     edx, [ebp+var_38] //target buffer 40 bytes (destination)
push    edx               ; char *
call    _strcpy
add     esp, 8
push    offset aCopySuccessful ; const char * // source string
lea     eax, [ebp+var_60] // target buffer 40 bytes (destination)
push    eax               ; char *
call    _strcpy
add     esp, 8
```

Disassembly of *strcpy()* function.

6.3.3 The Format Family

When one thinks of the format family one usually thinks of the *printf()* function which prints to a stream. There also several that print formatted input to memory. Functions such as *snprintf()* and

vsnprintf() all print formatted input to memory however as they both include strict bounds checking on the target buffer these really do not have implications associated with buffer overflows. There is one function that is a member of the format family that prints to memory and has no feature to enforce target buffer size limitations. That function is the *sprintf()* function, which has seen widespread use. Not only is this function responsible for many of the buffer overflow security flaws it‘s incorrect use is responsible for the so-called format string security vulnerability. The *sprintf()* function is unique, in that it is used with a string literal in many cases that, depending on how it is used, provides us with a ready made entry point to the program. The following *sprintf()* program demonstrates this characteristic:

```
/* sprintf example */
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  char buf1 [50];
  char buf2[25];
  printf("Enter a string less than 25 characters: \n");
  scanf("%s", buf2);
    if (strlen(buf2)>20) // do not want to overflow!!
     {
       printf("Stringlength is greater than 25 characters");
       exit(1);
     }
    else
     {
//** string literal entry point
      sprintf (buf1, "\nThis is our input: %s \n", buf2); point

      printf ("%sIt represents a program entry point\n\n",buf1);
     }
  return 0;
}
```

Sample *sprintf()* program.

The disassembly demonstrates the how the string literal is represented along with the format specifier. This gives a solid clue, in many cases, as to where in the running program the function is called.

A target buffer of 50 bytes is pushed onto the stack as var_34 along with the string literal “This is our input”. The disassembly of the *sprintf()* program sample is provided below:

```
lea     edx, [ebp+var_50]
push    edx
push    offset aThisIsOurInput ; "\nThis is our input: %s \n"
lea     eax, [ebp+var_34]
push    eax
call    _sprintf
add     esp, 0Ch
```

Disassembly of *sprintf()* function.

With a function that inherently dangerous and at the same time may provide us with a entry point for user input to a running application, *sprintf()* will be the foundation for the algorithm that we use to search the binary image.

6.4 Search Algorithm

IDA Pro provides a powerful scripting language, similar to C, that provides over 200 unique functions appropriate for use within the disassembly environment[1]. The algorithm we will develop will utilize this capability. We know that we will be performing our analysis based on the *sprintf()* function. As a review, since a call to a *sprintf()* function can expand an arbitrary string using the “%s” format variable, any call to this function which expands dynamic user input data into a buffer of fixed size shall be considered suspicious. With this in mind we will want to verify that the call to *sprintf()* contains a “%s” format character and that it targets a static buffer on the stack. In addition we want to identify the target buffer by name and provide the string literal as output. The algorithm will be constructed in (4) modules or user defined functions:

1) A *main* program which accepts user input in the form of either a direct address to the *sprintf()* call or as a indirect address to the *sprintf()* call then calls the analysis module passing it the address value.

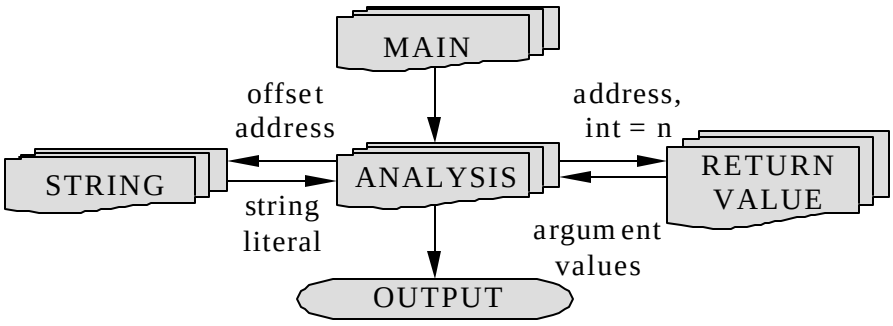
2) An *analysis* function which calls the return value module then cleans up the string arguments returned by the value module as an offset address of our string literal. It calls the string module that returns the string literal then scans the string for the presence of a “%s” format character and prints output.

3) A *return value* function which returns the values to the analysis module that have been pushed onto the stack for use by the *sprintf()* function. Of particular interest is the value at the n^{th} PUSH before the call as it represents the target buffer.

1. A listing of the functions used in the search algorithm is provided in Appendix D.

4) A *string* function which builds our string literal a byte at a time and returns it as a argument to the analysis module.

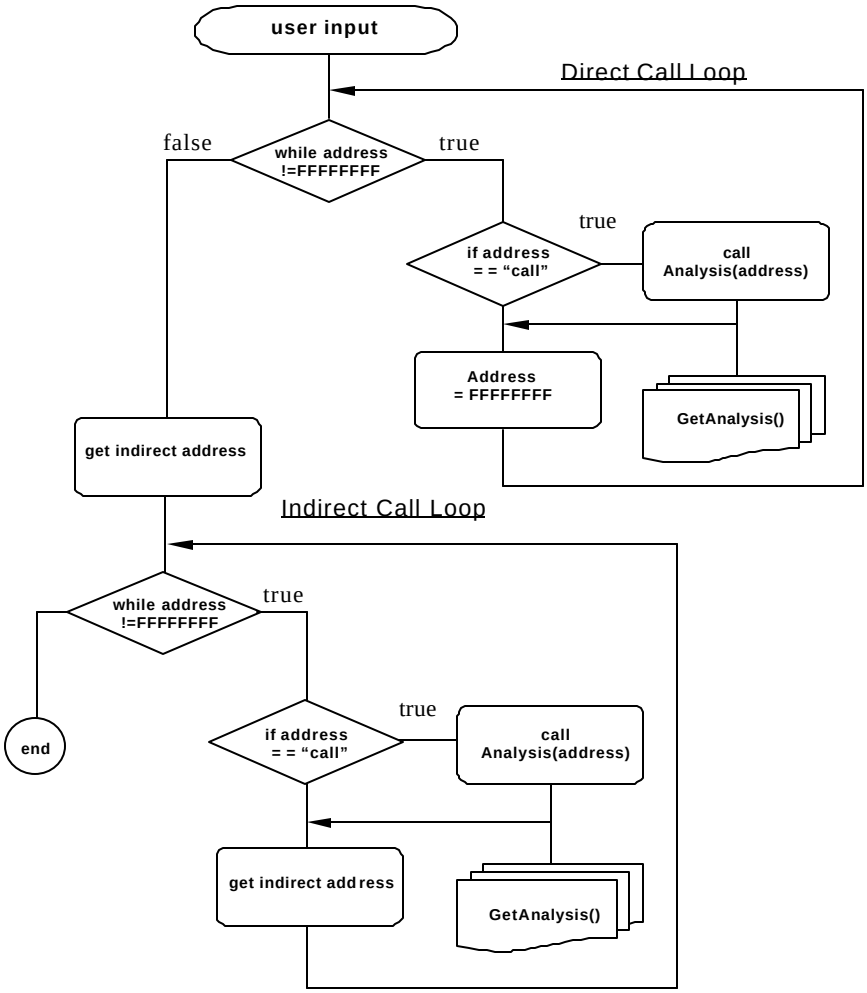
Figure 19: Program Layout and Data Passing



6.4.1 Main

The main function will accept user input in the form of an address of a `sprintf()` call. A call may either be direct, as will always be the case with statically linked library calls, or indirect. A indirect call results from referencing a dynamically linked library (i.e. msvcrt.dll) for the function code. This indirect call is referenced within the `i.data` segment of the PE file and contains all cross references to the call within the `.text` or code segment of the PE file. For this reason, we need the capability to run our analysis using either of the direct or indirect calls. To accomplish this we will use two loops, one for a direct call and one for a indirect call. For the direct call to `sprintf()` we will simply pass that address to our `GetAnalysis()` function. For the indirect call we will loop through all the references to `sprintf()` within the code section of the program, returning each of these values to the `GetAnalysis()` function. Both loops will terminate when the address value goes to -1 (FFFFFFFF).

Figure 20: Program Flow: *Main()*



When performing an analysis of a given program or application the user must find all occurrences of the *sprintf()* function by address. To accomplish this the user selects from IDA's tool bar (search --> text) and enters *sprintf()*.

The main function will rely heavily on IDA's built in cross referencing functions or *xref*s. These functions are particularly useful when traversing either up or down in address space within the execution stack from a known point of reference. For the interested reader, a listing of all *xref* functions has been provided in appendix D. Special attention should be given to the use of IDA's *xref* functions to traverse up and down the assembly listing in the context of the sprint_scan.idc program. In addition for direct calls, we use the *Rnext* function to force the *reference* variable to -1 (FFFFFFFF) to exit the first loop. A

similar technique is used with the *DnextB* function to exit the second loop for all indirect calls. The heavily commented code listing for main() is presented below.

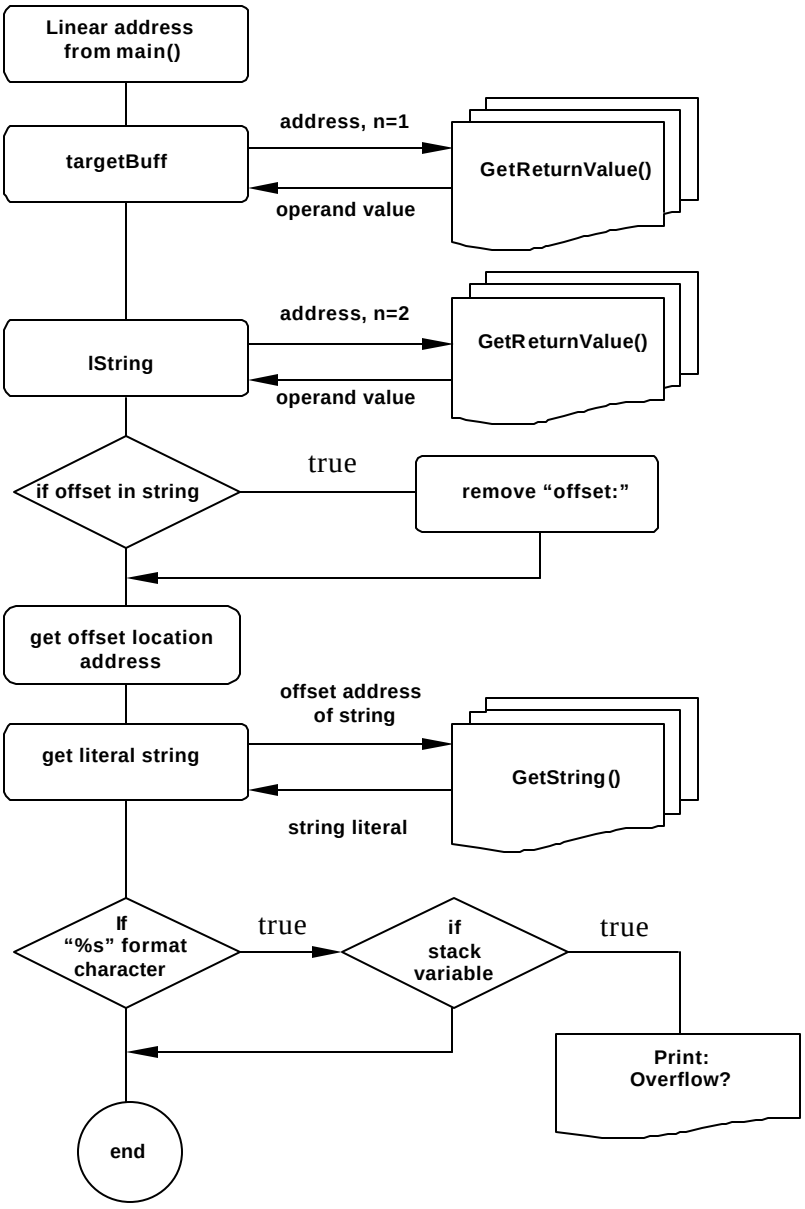
```
static main()
{
     auto   SprintfAddr, reference; // assign variables
     SprintfAddr = AskAddr(-1, "Enter address:"); // ask user for sprintf() address
     reference = (SprintfAddr); // assign point of reference to sprintf() address

//*** enter first loop for direct call to sprintf() ***//
     while(reference != -1) // while our address is not FFFFFFFF
     {
          if(GetMnem(reference) == "call") // is the mnemonic of the instruction a "call"?
               GetAnalysis(reference); // if it is a call, it is a direct reference; pass reference address to GetAnalysis
          reference = Rnext(SprintfAddr, reference); // make any reference address FFFFFFFF exiting first
loop
     }

reference = DfirstB(SprintfAddr); // get first indirect reference to sprintf() call

//*** enter second loop for indirect call to sprintf() ***//
     while(reference != -1)
     {
          if(GetMnem(reference) == "call") // is the mnemonic of the instruction a "call"?
               GetAnalysis(reference); // if it is a call, it is a indirect reference; pass reference address to GetAnalysis
          reference = DnextB(SprintfAddr, reference); // do this until FFFFFFFF
     }
}
```

Program Listing for *Main()*

6.4.2 GetAnalysis

The *GetAnalysis()* function can be viewed as a central switchboard, calling then passing data as arguments to other functions. The *GetAnalysis()* function is responsible for getting our target buffer value by address, retrieving our string literal in addition to cleaning up our arguments and printing our output. The *GetAnalysis()* function operates through the use of conditional statements as shown in the following flow chart.

Figure 21: Program Flow: *GetAnalysis()*



The code listing for *GetAnalysis()*, presented below, utilizes several of IDA's string related commands such as *strlen*, *strstr* and *substr*. First we call the *GetReturnValue()* function and pass it an address and a integer value. *GetReturnValue()* then returns the first two arguments to the *sprintf()* call. That is, the value pushed into our target buffer and the pushed offset value of our string literal. Using a substring search combined with a conditional we check to see if the word "offset" is part of our second pushed argument. If it is then we use the substring command to remove it. This second argument is now in a format to return a location address of the string literal itself. We then call the *Get-*

String() function with the offset address of the string literal and *GetString()* returns the literal string in character format. A nested conditional then checks, first for the presence of the "%s" format character and if it exists, for the existence of "var_" as a stack buffer value which indicates fixed size. If both conditions are met we have the potential for a buffer overflow condition by expanding string input with the "%s" format character into a stack buffer of fixed size. Our output information is printed within IDA's output screen and utilizes format characters in the *Message* command whose methods are similar to that of *printf()*.

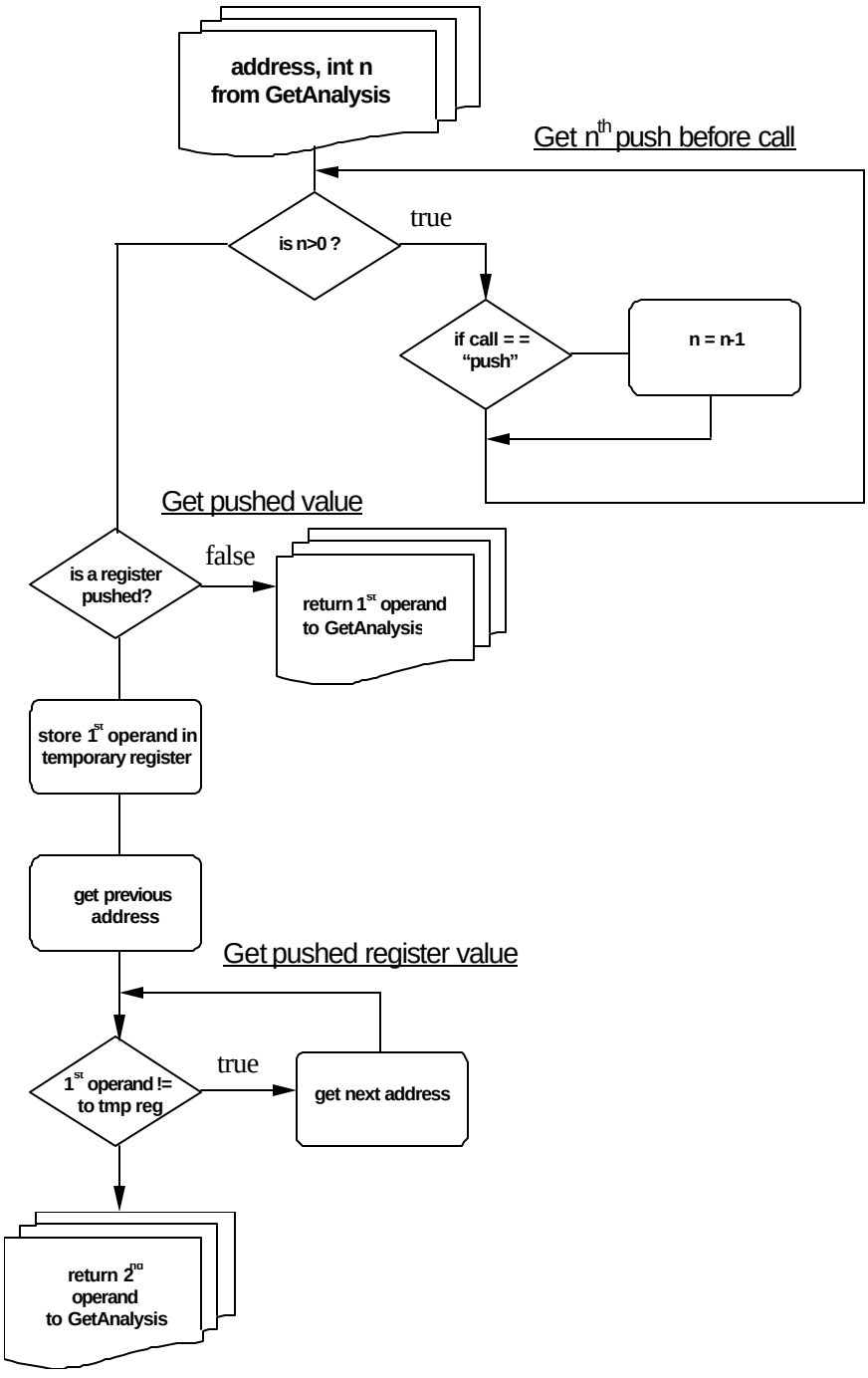
```
static      GetAnalysis(push)
{
     auto   literalString, literalStrAddr, targetBuffer; // assign variables

     targetBuffer = GetReturnValue(push, 1); // pass ref address + interger value (1) to GetReturnValue
     literalString = GetReturnValue(push, 2);// pass ref address + interger value (2) to GetReturnValue

     if(strstr(literalString, "offset") != -1) // does our literal string contain the word "offset:"?
          literalString = substr(literalString, 7, -1); // if so, remove it

     literalStrAddr = LocByName(literalString); // get the address of our literal string
     literalString = GetString(literalStrAddr); // pass the address of our literal string to GetString

//*** if conditions are met, print warning
    if(strstr(literalString, "%s") != -1) // does our literal string contain a "%s" format character
        if(strstr(targetBuffer, "var_") != -1) // if so, is the associated stack buffer variable in length
          Message("\n%lx --> POTENTIAL OVERFLOW? Target Buffer is: " + targetBuffer + "
                   String Literal is: \"%s\"\n\n\n", push, literalString);

}
```

Program Listing for *GetAnalysis()*

6.4.3 GetReturnValue

We are interested in the value that is pushed onto the stack immediately prior to the call to *sprintf()* as this is the value associated with the target buffer. The pushed value will be either in the form of a pushed register or a pushed offset value. We are also interested in the other arguments that are pushed onto the stack in preparation of the call to *sprintf()*. In *GetReturnValue()* we retrieve the n^{th} PUSH before the call using a looping structure. If a register is pushed, we trace back up through the disassembly to find where the register was last accessed and return that value. If an immediate offset was pushed we return that value. This accomplished with a conditional statement and another looping structure as exhibited in the following figure.

Figure 22: Program Flow: *GetReturnValue()*



The code listing for *GetReturnValue()* demonstrates the use of two while loops and a conditional statement. The first loop retrieves the n^{th} PUSH before the call using the integer value n=1 passed to *GetReturnValue()* by *GetAnalysis().* When the first PUSH is located, n is decremented and

goes to zero causing the loop to be terminated. A conditional checks to see if the first operand of the PUSH instruction is of type “register”. If the operand represents a register, we create temporary storage to hold the identity of the pushed register. We then enter a while loop that traces upward through each line of the disassembly until we find the line that last accessed our register. This accomplished by a line by line comparison with the stored register identity until a match occurs. When we find a match we return the text representation of the second operand, which is the value stored in the register, to `GetAnalysis()`. If the value of the n^{th} PUSH is not a register, we simply return the immediate value or the text representation of the first operand.

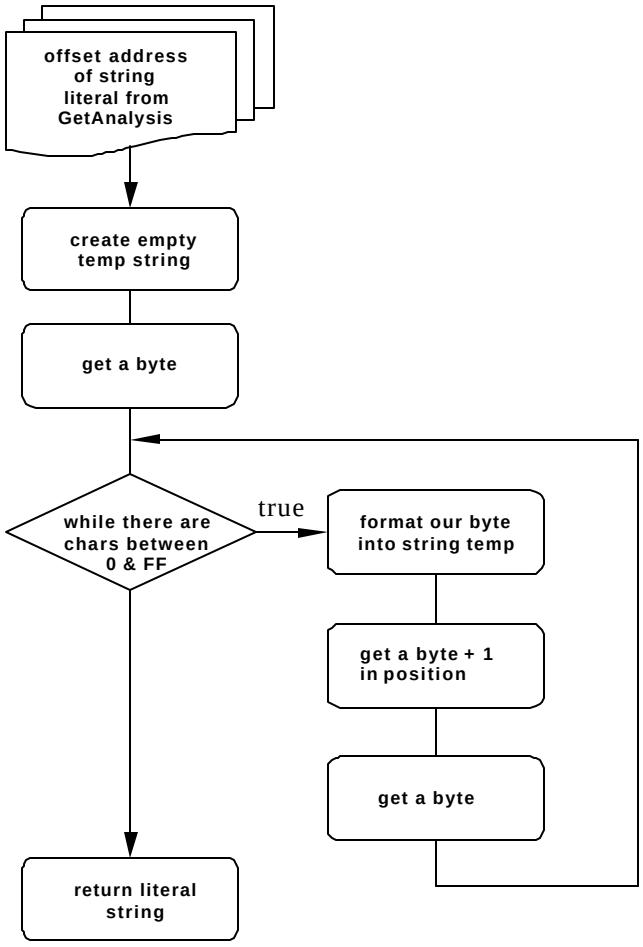
```
static      GetReturnValue(push, n)
{
     auto   temporaryRegister;  // initialize variables

//*** get nth push before call
     while(n > 0)
     {
          push = RfirstB(push);
          if(GetMnem(push) == "push")
               n = n-1;
     }


//*** is a register pushed
     if(GetOpType(push, 0) == 1)  // 1==a GetOpType register
     {
          temporaryRegister = GetOpnd(push, 0);  // create storage to hold the pushed register (i.e. eax,esi)
          push = RfirstB(push);  // get previous instruction


//*** while the pushed operand != our pushed register, keep looking until it does
          while(GetOpnd(push, 0) != temporaryRegister)
               push = RfirstB(push);  //get next previous instruction
          return(GetOpnd(push, 1));  // return the value pushed into the register
     }
     else return(GetOpnd(push, 0));  // if nth push was an immediate offset return it
}
```

Program Listing for `GetReturnValue()`

6.4.4 GetString

IDA does not include a function for string reassembly. For this reason we must construct a function that will reconstruct a string a byte at a time. The `GetString()` function performs this task after being passed the offset address of the string literal. The function creates temporary storage then reassembles the string a byte at a time using a looping structure until a null or 0xFF value byte is reached. A flowchart for `GetString()` follows.

Figure 23: Program Flow: *GetString()*



The following code for *GetString()* is fairly straight forward. *GetString()* is called within *GetAnalysis()* and is passed the offset address of the literal string. The function first creates a empty string to receive the characters a byte at a time. After the first byte is read the function enters a while loop and continues reading the string a byte at a time appending each character to our temporary string. When we reach a null or 0xFF byte the loop exits and the function returns the literal string to *GetAnalysis().*

```
static GetString(ourString)
{
      auto  temporaryString, character; // assign variables
      temporaryString = ""; // create empty string
      character = Byte(ourString); //get the 1st byte at the offset address

//*** while loop to read a character at a time and append each to our empty string
      while((character != 0)&&(character != 0xFF)) //read while we have bytes to read
     {
            temporaryString = form("%s%c", temporaryString, character); //format our empty string
            ourString = ourString + 1; // move to the next position
            character = Byte(ourString); // get the next byte
      }

      return(temporaryString); //return the literal string to GetAnalysis()
}
```

Program Listing for `GetString()`

6.4.5 Summary

We have developed a algorithm to locate potential buffer overflow vulnerabilities using a dangerous function call, `sprintf()`, within the context of the IDA disassembler. IDA provides a powerful scripting language which we have used along with our algorithm to develop a program[1], sprintf_scan.idc, to search a binary image for certain signatures of potentially dangerous `sprintf()` coding constructs within the disassembly. Now that we have a program, the next step will be the development of a small, simple test program to demonstrate how well the scanning technique performs.

6.5 Initial Testing

In order to assess the performance of our binary scanning technique we require a simple program with a known flaw in the use of the `sprintf()` function. This flaw should include a static buffer of fixed size located on the stack and a string literal combined with the format character “%s” that expands user input into a string. This will validate the two key features of our method:

1) Finding the value of the fixed buffer on the stack

2) Identification of the string literal for use as a program entry point.

To have a technique that performs well as measured by what it can identify is not enough. The technique should also be measured by what it does not identify as being suspicious. That is, there should be minimal false positives when the `sprintf()` call is used in a safe manner.

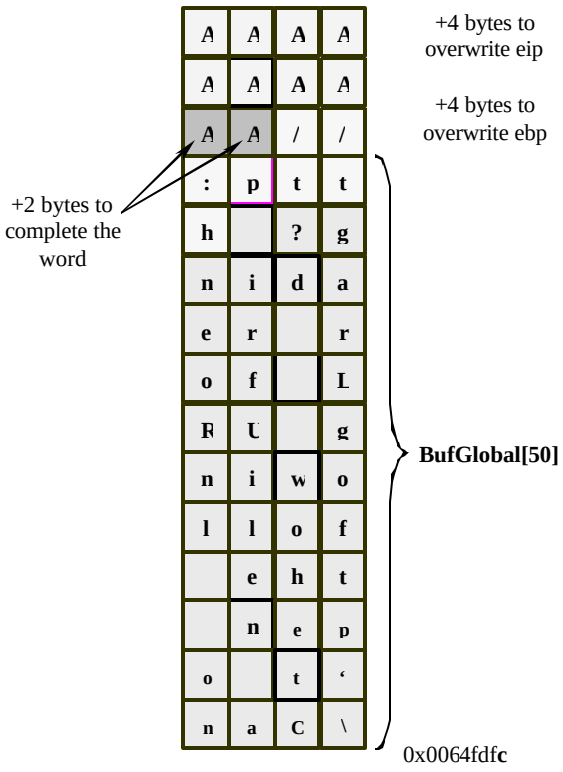
1. For a complete listing of the program reference Appendix E.

6.5.1 sprintf_crasher.c

Our test program[1] uses (3) calls to `sprintf()`, two are considered safe while the third is used in a manner that causes a buffer overflow from excessive user input. This is a command line application that scans user input into buffers of different sizes. The first call to `sprintf()` receives the scanned user input of up to 100 character into x.bufLarge a [100] character buffer. By using the format character ".3%s" only the first (3) characters are read into x.bufSmall a [60] character buffer. This represents the safe use of the `sprintf()` function. In the second call to `sprintf()`, a (3) character string literal is read directly into newBuf, a [25] character buffer. This also represents safe coding practice.

In the last call to `sprintf()`, x.bufLarge a [100] character buffer, receives up to 100 characters of user input. This input is formatted into bufGlobal, a [50] element buffer, along with the string literal "Can't open the following URL for reading? %s" of [42] characters. The format character "%s" will read every character in x.bufLarge[100] and attempt to write each one to bufGlobal[50]. The sprint_crasher.c program was compiled using Microsoft Developer Studio 97; Visual C++. The following discussion will reference figure 24.

Figure 24: Test Program Stack Behavior

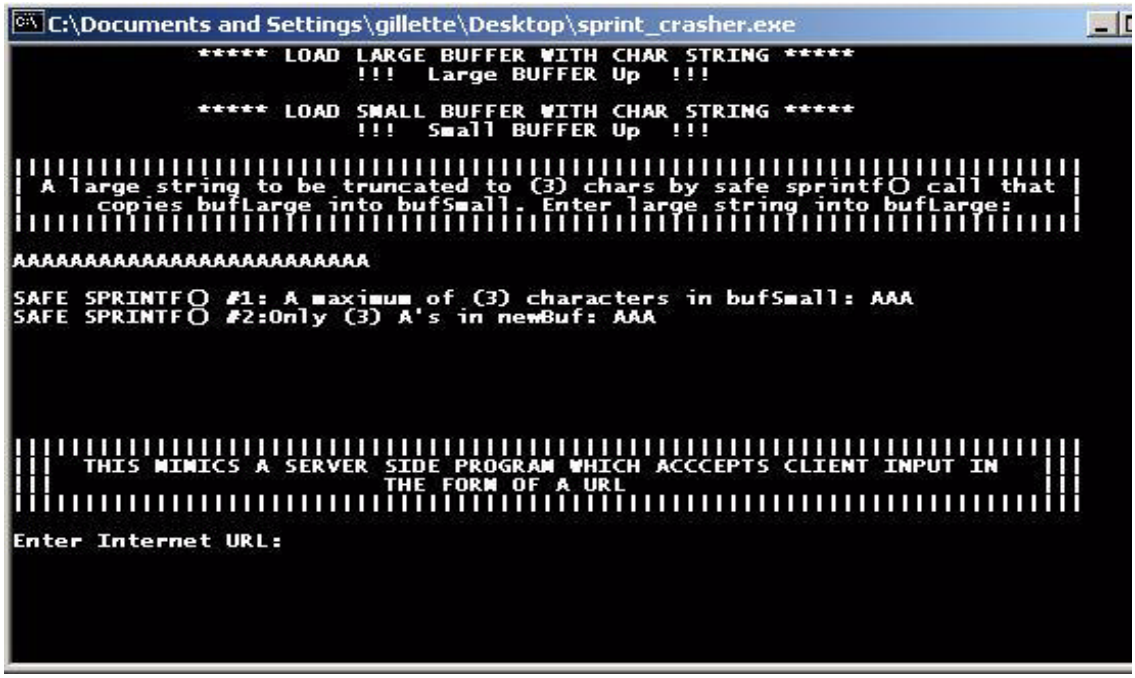


1. The code for sprintf_crasher.c is presented in Appendix F.

As the first 42 elements within bufGlobal are already occupied with the string literal it only takes (8) additional characters to begin overflowing the bufGlobal[50]. We write to memory a word at a time (4 bytes) and in bufGlobal[50] the character at element [50], the 7th character of user input, is in the middle of a word boundary. User supplied characters 8 & 9 complete the word with character (10) starting the corruption of the stack base pointer *(ebp)* causing the program to crash. With (17) user supplied characters we have completely overwritten the instruction pointer *(eip)*.

Running the program, we first enter a string of (25) capital A's and see that only (3) of the characters have been copied into memory by the first (2) safe `sprintf()` functions.

Exhibit 1: Test Program with (2) Safe `Sprintf()` returns



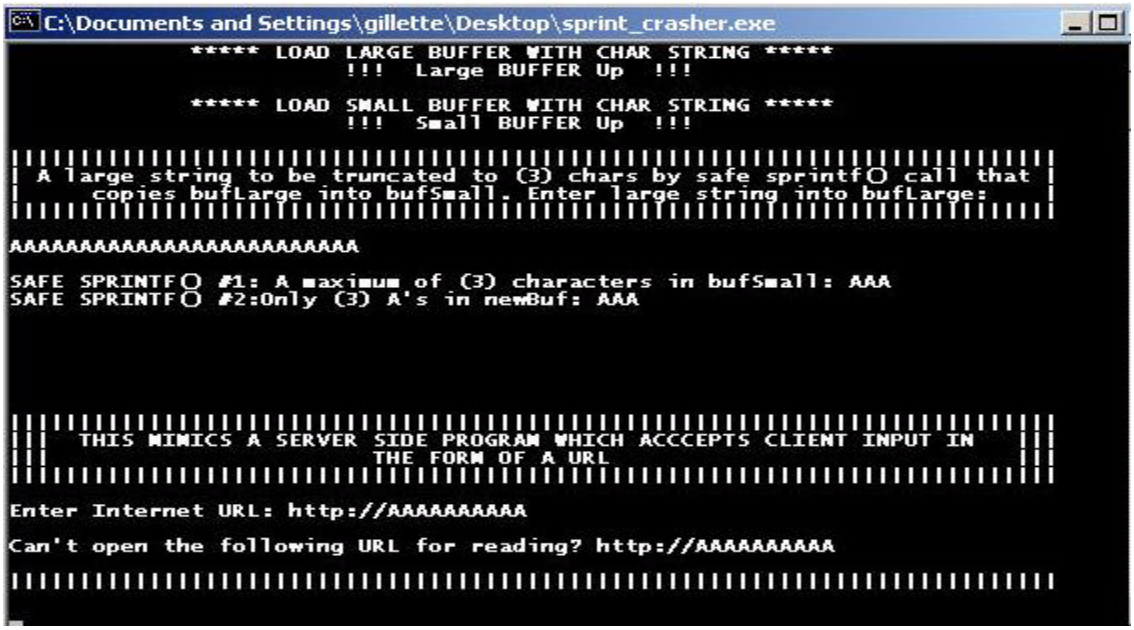
We then enter the following URL "http://AAAAAAAAAA", a string of (17) characters, and notice that we get the following message:

Exhibit 2: Test Program Page Fault

The error log shows that both ebp and eip have been over written with capital A's, (0x41) in hexadecimal notation.

```
eax=00000000    ebx=00540000    ecx=00414100
edx=00413fb0    esi=817e6de4    edi=00000000
eip=41414141    esp=0064fe00    ebp=41414141
```

Exhibit 3: Test Program Showing String Literal



It is also important to note how our string literal is returned after the call to *sprintf()*. "Can't open the following URL for reading?", with the user supplied string of "http://AAAAAAAAAA".

6.5.2 First Binary Scan

To perform our first binary scan using our program sprintf_scan.idc we load our test program into the disassembler and observe the following screen.

Exhibit 4: Disassembly of (2) Safe *Sprintf()* Calls in Test Program

```
                        ; CODE XREF: _main+109↑j
lea     eax, [ebp+var_174]
push    eax
push    offset a_3s     ; "%.3s"
lea     ecx, [ebp+var_110]
push    ecx
call    _sprintf
add     esp, 0Ch
lea     edx, [ebp+var_110]
push    edx
push    offset aSafeSprintf1AM ; "\nSAFE SPRINTF() #1: A maximum of (3) ch"...
call    _printf
add     esp, 8
push    offset off_0_413D50
lea     eax, [ebp+var_190]
push    eax
call    _sprintf
add     esp, 8
```

We see the first two, so-called "safe", calls to *sprintf()* along with the associated arguments that are pushed onto the stack. For the first call, we see that *ecx* is pushed onto the stack as the nth PUSH before the call. As the pushed value is a register, we need to find where it was last accessed. Register *ecx* is accessed in the next previous step and we note that *ecx* is being loaded with var_110. Similar behavior is observed with the second call to *sprintf()*.

The other call that we are interested in is the third call to *sprintf()*. This is the call that contains the programming error that leads to a potential buffer overflow vulnerability. The nth PUSH before this call is register edx and it is accessed in the next previous step where it is loaded with var_34.

Exhibit 5: Disassembly of Flawed *Sprintf()* Calls in Test Program

```
                        ; CODE XREF: _main+1EF↑j
lea     ecx, [ebp+var_174]
push    ecx
push    offset aCanTOpenTheFol ; "\nCan't open the following URL for readi"...
lea     edx, [ebp+var_34]
push    edx
call    _sprintf
add     esp, 0Ch
lea     eax, [ebp+var_34]
push    eax
push    offset aS_1     ; "%s\n"
call    _printf
add     esp, 8
push    offset asc_0_413F60 ; "\n||||||||||||||||||||||||||||||||||||||||"...
call    _printf
add     esp, 4
```

We perform a search for all the ***sprinf()*** references within the disassembly and observe the following addresses:

1) call #1-.text:0040113c
2) call#2-.text:00401164
3) call#3-.text:0040121f

Exhibit 6: Address Values for ***Sprintf()*** Calls in Test Program

```
.text:0040113C                  call    _sprintf
.text:0040114B                  push    offset aSafeSprintf1AM ; "\nSAFE SPRINTF() #1: A maximum of (3) ch"...
.text:00401164                  call    _sprintf
.text:00401173                  push    offset aSafeSprintf2On ; "SAFE SPRINTF() #2:Only (3) A's in newBu"...
.text:0040121F                  call    _sprintf
.text:00401360          ; [000000FA BYTES: COLLAPSED FUNCTION _sprintf. PRESS KEYPAD "+" TO EXPAND]
.rdata:00411060         aFormatNull     db 'format != NULL',0   ; DATA XREF: _printf+15|o _sprintf+45|o ...
.rdata:00411070         aSprintf_c      db 'sprintf.c',0        ; DATA XREF: _sprintf+24|o _sprintf+4E|o ...
.rdata:0041107C         aStringNull     db 'string != NULL',0   ; DATA XREF: _sprintf+1B|o
.rdata:00411230         aWsprintfa      db 'wsprintfA',0        ; DATA XREF: __CrtDbgReport+B6|o
.rdata:00411E38         aVsprintf_c     db 'vsprintf.c',0       ; DATA XREF: __vsnprintf+1E|o
.data:00413D0C          aSafeSprintf1AM db 0Ah                  ; DATA XREF: _main+146|o
.data:00413D54          aSafeSprintf2On db 'SAFE SPRINTF() #2:Only (3) A',27h,'s in newBuf: %s',0Ah,0
```

The three addresses are direct references which means that the ***sprintf()*** function code is from a statically linked library call. We performed a scan on all three references with the first two showing no results, as expected. On the third scan the following results are obtained.

Exhibit 7: sprintf_scan.idc Input Dialogue with Address of Flawed ***Sprintf()*** Call

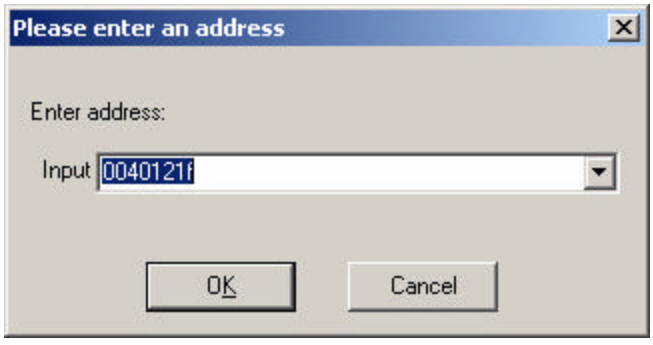
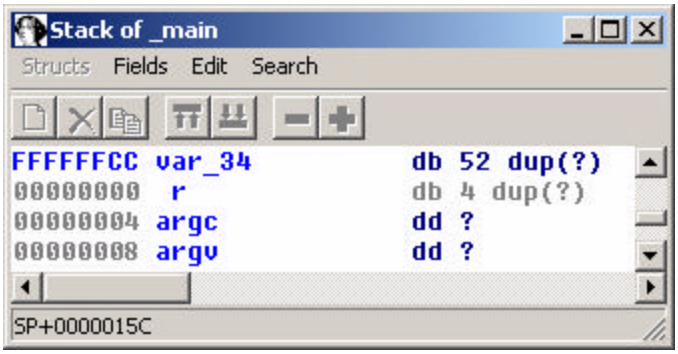


Exhibit 8: sprintf_scan.idc Output for Flawed *Sprintf()* Call

```
Compiling file 'C:\Program Files\DataRescue\IDA Pro v4.17\idc\sprintf_scan.idc'...
Executing function 'main'...

40121f --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_34]
String Literal is:
Can't open the following URL for reading? %s
```

This identifies a potential overflow at address 0x0040121f with a target buffer variable of var_34 and a string literal of “Can’t open the following URL for reading? %s”. When we look at the stack reference for var_34 we observe a buffer size of approximately 52 bytes which is very close to our allocated length of [50] elements.

Exhibit 9: Test Program Stack Showing [52] Byte Target Buffer



6.5.3 Summary of Initial Test Results

The sprint_scan.idc program performed according to our original design intent. This success was demonstrated using a small scale test program (sprint_crasher.c) with three calls to *sprintf()*. Two using correct programming practice, and one call to *sprintf()* with a known buffer overflow code flaw. This accomplishment exceeded our original specification of identifying certain instances of a buffer overflow vulnerability as fast as possible [1]. This enhanced level of performance was illustrated when the scanning program failed to identify identical functions, being used in a similar manner, as being possible security flaws. This selective identification enhanced the reliability of the scanning technique as it eliminated concerns associated with false positive results. We will move to extend the range of this technique by scanning third party, proprietary, software applications. We will begin with shareware class software

1. On a x86 800Mhz machine running Windows 2000 Professional the scan speed using sprint_scan.idc against sprintf_crasher.c was under (1) second.

as we believe that these types of programs are released with little regard for secure coding practices. We will move linearly up to enterprise class server products in an attempt to demonstrate the scalability of our technique.

6.6 Extended Testing

The testing methodology used to demonstrate the scalability of our technique was simple and straight forward. A target program was identified then the appropriate binary file was selected and scanned. If positive results were obtained, program documentation, specifications, RFC's etc. were obtained in order to gain insight into user input data format requirements, as well as methods related to how the user data would be passed to the target program. Several general techniques for remote passing of data streams to the host program were immediately identified. These included:

- Passing data strings within the web browser.
- Making a telnet connection to the proper port and transmitting data according to the related protocol.
- Using netcat, a simple utility which is used to read and write data across network connections.

Once the data format issues were addressed and one of the above techniques selected for passing user input, we attempted to exploit the identified buffer overflow vulnerability by trying to crash the program with excessive input. Success would be demonstrated when the target program stopped responding and two potential levels of achievement were identified.

1) Simple page fault error with no instruction pointer (*eip*) overwrite. This type of failure would indicate a strong possibility of creating a DOS condition at exploit time
2) Page fault error with complete instruction pointer overwrite. This type of failure would indicate a strong potential for the ability to remotely execute arbitrary instructions on the host machine.

It is important to note that each file tested involved unique protocols and peculiarities associated with how user data was formatted and passed to the program. In many instances, where an apparent overflow condition was identified, we simply did not have the time resources to learn all the program nuances to be able to demonstrate exploitability. In these cases we rapidly moved on to the next target. Target program candidates were identified based on the following criteria:

- Remotely accessed program with clear client host relationship
- Application with well known and documented buffer overflow vulnerability

With these ground rules in place, we attempted to start small with shareware class software progressing to enterprise class server products.

6.6.1 Shareware Testing

The idea behind shareware was that by being low cost solution it was also a high risk solution from a secure programming point of view. We approached this testing with the belief that little or no security audits are performed prior to release. This, despite the fact that the products identified share privileged process space on the host server.

6.6.1.1 Seattle Lab Internet Mail Server version 2.5.0.1065

The early versions of this shareware program were notorious for having numerous buffer overflow vulnerabilities. The buffer overflows were well documented with even a walk through disassembly in the paper “Win32 Buffer Overflows (Location, Exploitation and Prevention)” [58]. The binary file slmail.exe was loaded into the IDA disassembler and scanned with sprint_scan.idc. The disassembly container 350 references to the ***`sprintf()`*** function. Analysis results are provided below.

Exhibit 10: sprintf_scan.idc Output: SLMail

```
Compiling file 'C:\Program Files\DataRescue\IDA Pro v4.17\idc\sprintf_scan.idc'...
Executing function 'main'...
4035f8 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+178h+var_16C]
String Literal is: %s v%s
40eb43 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+54h+var_44]
String Literal is: %s version %s
40f54b --> POTENTIAL OVERFLOW? Target Buffer is: [esp+0E0h+var_C8]
String Literal is: %s v%s
412bc2 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_488]
String Literal is: mx-request-slmail-resolve-%s
4130a7 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_514]
String Literal is: nSmtpMx_RequestRemoteId; %s,%d,%d
421582 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+140h+var_D4]
String Literal is: MailboxAccess-%s
422529 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+8F8h+var_7C4]
String Literal is: User %s@%s is unknown.
424826 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+40Ch+var_1F4]
String Literal is: Invalid address found: %s
426bbe --> POTENTIAL OVERFLOW? Target Buffer is: [esp+140h+var_108]
String Literal is: %s v%s
42ccae --> POTENTIAL OVERFLOW? Target Buffer is: [esp+82Ch+var_414]
String Literal is: The system was unable to find the mailing list configuration
for the file: %s.

This file has been renamed to %s
to avoid this again in the future.

4312d1 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+1B4h+var_D0]
String Literal is: NET%04x: %-8s %s
434f2c --> POTENTIAL OVERFLOW? Target Buffer is: [esp+110h+var_D0]
String Literal is:   <%s.%s@%s>
4357f8 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+1B8h+var_19C]
String Literal is: From: Mailer-Daemon<MAILER-DAEMON@%s>
To: %s<%s>
435c57 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+210h+var_200]
String Literal is: u;%s;%s;%s;%s
435cb5 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+200h+var_200]
String Literal is: a;%s
435d15 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+200h+var_200]
String Literal is: f;%s
435d7a --> POTENTIAL OVERFLOW? Target Buffer is: [esp+200h+var_200]
String Literal is: r;%s;%s
435ddf --> POTENTIAL OVERFLOW? Target Buffer is: [esp+20Ch+var_200]
String Literal is: l;%s;%s;%s
4419e3 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+0A08h+var_834]
String Literal is: %s
441a17 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+0A04h+var_834]
String Literal is: %s
441a49 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+0A08h+var_631]
String Literal is: %s
441a79 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+0A04h+var_418]
String Literal is: %s
441aaa --> POTENTIAL OVERFLOW? Target Buffer is: [esp+0A14h+var_63D]
String Literal is: %s
441ad0 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+0A14h+var_53C]
String Literal is: %s
441b00 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+0A10h+var_424]
String Literal is: %s
441b3d --> POTENTIAL OVERFLOW? Target Buffer is: [esp+0A1Ch+var_548]
String Literal is: %s
```

Several addresses are immediately recognizable as accepting long strings of user data along with a corresponding program entry point. The two most promising addresses include:

1) 00422529: A long string could be supplied at some point in the program that looks up a user
2) 00424826: A long string could be supplied at some point in the program that handles addresses.

We played around with the program for quite a while, and without success, looking for these error messages to give us a clue as to where in the program they resulted from user input. In addition because numerous buffer overflows existed in the product, several being documented as being directly related to the poor implementation of the `strcpy()` function and often used in conjunction with a `sprintf()` call, it was felt that any buffer overflow found in this application would be inconclusive.

6.6.1.2 CesarFTP version 0.0.9.6[1]

This shareware program is reported as having a buffer overflow vulnerability associated with the "HELP" command. The binary file CesarFTP.exe was loaded into the IDA disassembler and exhibited (22) references to the `sprintf()` function which is dynamically linked through msvcrt.dll. The results of the binary scan are given below.

Exhibit 11: sprintf_scan.idc Output: CesarFTP

```
Compiling file 'C:\Program Files\DataRescue\IDA Pro v4.17\idc\sprintf_scan.idc'...
Executing function 'main'...
Compiling file 'C:\Program Files\DataRescue\IDA Pro v4.17\idc\sprintf_scan.idc'...
Executing function 'main'...

40be58 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+174h+var_10C]
String Literal is: Running on %s processor(s) at %s


40be74 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+170h+var_10C]
String Literal is: Running on %s processor(s)


40be94 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+170h+var_10C]
String Literal is: TCP/IP Stack: %s
```

While we get positive returns associated with two addresses, the parameters that are read by "%s" appear to by internally generated.

6.6.1.3 Winamp version 2.6.0.0

This shareware audio file player has several reported buffer overflow vulnerabilities associated with the earlier versions. While this application is not known for supporting any type of client host relationship, it is able to download MP3 music files directly from the internet using the AudioSoft AIP file format. These .aip files are parsed by winamp and a specially crafted file is able to overflow memory

1. Reference Appendix C: Case 3

space. Since these .aip files can be transferred across the internet and directly downloaded without user intervention the winamp application represents a unique case. Other reported buffer overflows leave us with the feeling that this program was not released with any concern for potential security issues. Winamp.exe was loaded into the IDA disassembler and (173) references to ***sprintf()*** were noted as dynamically linked imports from user.dll. The binary image was scanned with the results of the analysis presented below.

Exhibit 12: sprintf_scan.idc Output: WinAmp

```
Compiling file 'C:\Program Files\DataRescue\IDA Pro v4.17\idc\sprintf_scan.idc'...
Executing function 'main'...
405531 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_210]
String Literal is: %s\Winamp.lnk
40555e --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_210]
String Literal is: %s\What's new.lnk
4055ce --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_210]
String Literal is: %s\Uninstall Winamp.lnk
406303 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+8C4h+var_400]
String Literal is: %s\viscolor.txt
4065f9 --> POTENTIAL OVERFLOW? Target Buffer is: [esp+410h+var_400]
String Literal is: %s\%s
40e94e --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_84]
String Literal is: Balance: %d%% %s
40edbf --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_8C]
String Literal is: EQ: %s: %s%d.%d db
40f31b --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_408]
String Literal is: %s\%s
40ffe7 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_12F4]
String Literal is: GET %s HTTP/1.0
User-Agent: Winamp/2
Host: %s
Accept: */*

41089f --> POTENTIAL OVERFLOW? Target Buffer is: [esp+0B0h+var_7C]
String Literal is: hi.%s
411b95 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_408]
String Literal is: http://www.winamp.com/update/updatelinks.jhtml?i=%c&v=%s&r=%c
41d97d --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_2C8]
String Literal is: Winamp base skin v%s
42015d --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_25C]
String Literal is: %s    [%s]
420475 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_258]
String Literal is: %s    [%s]
420be2 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_12A0]
String Literal is: %s\%s
420e93 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_F74]
String Literal is: %s\%s
420f54 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_F74]
String Literal is: %s   [%s]
42148a --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_136C]
String Literal is: %s\%s
421667 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_D50]
String Literal is: %s\%s
423414 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_418]
String Literal is: %s\%s
42390e --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_124]
String Literal is: http://%s
4250d5 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_104]
String Literal is: %s%s%s
4250e9 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_104]
String Literal is: %s
4257bc --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_5A8]
String Literal is: %s\%s
42920b --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_550]
String Literal is: %s\%s
42985b --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_110]
String Literal is: %s\%s
429931 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_110]
String Literal is: %s\%s
42c1f5 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_84]
String Literal is: Balance: %d%% %s
42c44c --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_418]
String Literal is: %s\%s
42cefa --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_540]
String Literal is: http://www.winamp.com%s
42cf22 --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_ED0]
String Literal is: GET %s HTTP/1.0
User-Agent: Winamp/2
Host: www.winamp.com
Accept: */*

```

Address 0042390e “http://<user input>” as read by “%s” certainly looks promising. A URL string is accepted as part of the playlist functionality however extensive testing using strings of various length failed to crash the program.

6.6.1.4 OmniHTTPd version 1.01

The image map CGI that is distributed with OmniHTTPd has a buffer overflow associated with the server side extension imagemap.exe. The binary file Imagemap.exe was loaded into the IDA disassembler and (2) references to the `sprintf()` function were noted. The file was scanned with no positive results.

6.6.2 Enterprise Class Server Applications

6.6.2.1 fp30reg.dll[1] version 4.0.2.3406

This is the .dll within Microsoft’s FrontPage 2000 server extensions that was exploited by the code red worm. According to published information, fp30reg.dll parses client input and in the case where fp30reg.dll receives an invalid parameter (method), the following message was returned: “The server is unable to perform the method <parameter provided by the user> at this time”. The <parameter provided by the user> we hoped to be a `sprintf()` “%s” write to memory. As a matter of fact published documentation included the statement “...fp30reg.dll calls USER32.wsprintfA() to form the return message.”[69]. We really had our hopes up on this one as it represented a contemporary enterprise class server application however, when we performed our analysis with sprintf_scan.idc, it returned no results.

6.6.2.2 Microsoft ftp Client version 5.0.2134.1

FTP programs have a reputation for exploitable buffer overflows. Although no direct information was available for this particular version as to the presence of a buffer overflow vulnerability the following information led us to believe that indeed a vulnerability exists. The file was obtained from a corporate implementation of Windows 2000 where the file had been globally removed as part of a system wide security audit. The binary file was loaded into the IDA disassembler where (7) references to the `sprintf()` function were noted. The scan of the file returned no positive results.

6.6.2.3 Microsoft Frontpage 2000 Server Extensions

Two components of FrontPage 97, 98 and 2000 Server Extensions, htimage.exe and imagemap.exe are documented as containing buffer overflow vulnerabilities with htimage being reported to crash with the string “http://myserver/cgi-bin/htimage.exe/<741 A’s>?0,0”. It is important to note that

1. Reference Appendix C: Case 7

there is no documentation associated with Microsoft's imagemap.exe other than the basic report that a vulnerability exists.

Image mapping allows for the attachment of a hyperlink to a image within a web page. As an example, a web site might display a picture of a football team, and by clicking on an individual player it might lead to a page showing his family or interests. When you select a point on the active image, the x and y position (in pixels) are sent to the specified URL, using the GET method, as a query string, like this: GET /cgi-bin/program/example?62,58 with the top left corner being 0,0.

The technique of integrating the image with the hyperlink is known as image mapping. Most contemporary browsers support client side image mapping. That is the browser can display the image and integrate the hyperlink natively. However, with legacy browsers (for example, NCSA's Mosaic or the early Internet Explorer versions) client side image mapping was not supported and components on the server side were required to allow this functionality. The two components, htimage.exe and imagemap.exe, perform this function to maintain cross compatibility with the two original specifications, CERN's and NCSA's.

The binary file imagemap.exe was loaded into the IDA disassembler with (3) references to *`sprintf()`* being noted.An analysis was performed using sprintf_scan.idc and the following results returned.

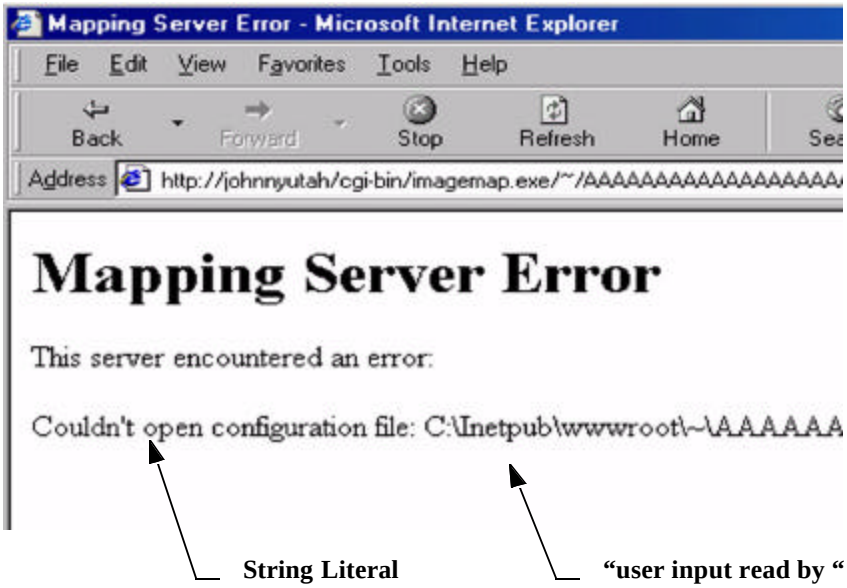
Exhibit 13: sprintf_scan.idc Output: imagemap.exe

```
Compiling file 'C:\Program Files\DataRescue\IDA Pro v4.17\idc\sprintf_scan.idc'...
Executing function 'main'...

40166e --> POTENTIAL OVERFLOW? Target Buffer is: [ebp+var_C0C]
String Literal is: Couldn't open configuration file: %s
```

The string literal "Couldn't open configuration file: <user input>" with user input being read by "%s" and written to memory certainly looks interesting. As imagemap.exe is a server side program we need to run it within the context of host application that receives client input. To accomplish this we will run the application from Microsoft's Personal Web Server version 2.0 within Windows 98. This configuration is similar to the Microsoft's IIs server products. Imagemap.exe is installed in the *.../cgi-bin* directory and accepts query strings through the client browser in the form of "http://<myserver>/cgi-bin/imagemap.exe/<other directory>/<query string>?(map coordinates) (i.e. 0,0).

We start the Personal Web Server Service, the server name is johnnyutah, and pass the query string "http://johnnyutah/imagemap.exe/~/AAAAAAAAAAAAAAAAAAAA?0,0" to imagemap.exe with the following results.

Exhibit 14: Imagemap Server Error with String Literal



We have found our string literal reference and the format character is reading our dynamic input and writing it to memory. This is demonstrated by the screen message "Couldn't open configuration file: C\Inetpub\wwwroot\~\AAAAAAAAAAAAAAAAAAAA". We look within the disassembly at the *sprintf()* call referenced in our output results; address 0040166e.

Exhibit 15: Imagemap.exe Disassembly Showing Flawed *Sprintf()* Call

```
                        ; CODE XREF: _main+C0↑j
                        ; _main+DB↑j ...
lea     eax, [ebp+var_824]
push    offset aR       ; "r"
push    eax
call    ds:fopen
pop     ecx
mov     [ebp+var_C], eax
test    eax, eax
pop     ecx
jnz     short loc_0_401682
lea     eax, [ebp+var_824]
push    eax
lea     eax, [ebp+var_C0C] ; target buffer register load
push    offset aCouldnTOpenCon ; "Couldn't open configuration file: %s"
push    eax             ; target buffer push register eax
call    ds:sprintf
add     esp, 0Ch
lea     ecx, [ebp+var_C0C]
call    sub_0_401CD1
```

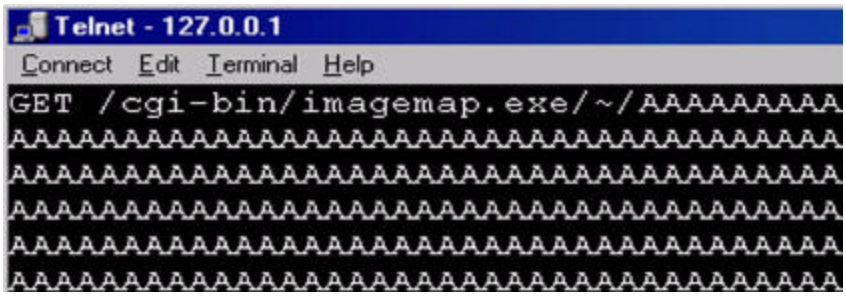
We observe that the target buffer PUSH is register *EAX* with the corresponding register load represented by variable [var_C0C]. When we explore the stack we observe that var_C0C represents a static stack buffer of 500 bytes.

Exhibit 16: Imagemap.exe Stack Space Showing Target Buffer

```
FFFFEDB4
FFFFEDB4 var_124C        dq ?
FFFFEDBC var_1244        dq 199 dup(?)
FFFFF3F4 var_C0C         db 500 dup(?) ; target buffer
FFFFF5E8 var_A18         db 500 dup(?)   suspicious sprintf()
FFFFF7DC var_824         db 500 dup(?)   call
FFFFF9D0 var_630         db 500 dup(?)
FFFFFBC4 var_43C         db 500 dup(?)
FFFFFDB8 var_248         db 6 dup(?)
FFFFFDBE var_242         dd 7 dup(?)
FFFFFDDA                 db ? ; undefined
FFFFFDDB                 db ? ; undefined
FFFFFDDC var_224         db ?
FFFFFDDD var_223         db 499 dup(?)
FFFFFFD0 var_30          dq ?
FFFFFFD8 var_28          dq ?
FFFFFFE0 var_20          dq ?
FFFFFFE8 var_18          db 12 dup(?)
FFFFFFF4 var_C           dd ?
FFFFFFF8 var_8           dd ?
FFFFFFFC var_4           dd ?
00000000  s              db 4 dup(?)
00000004  r              db 4 dup(?)
00000008 argc            dd ?
0000000C argv            dd ?                    ; offset (FFFFFFFF)
00000010 envp            dd ?                    ; offset (FFFFFFFF)
00000014
00000014 ; end of stack variables
```

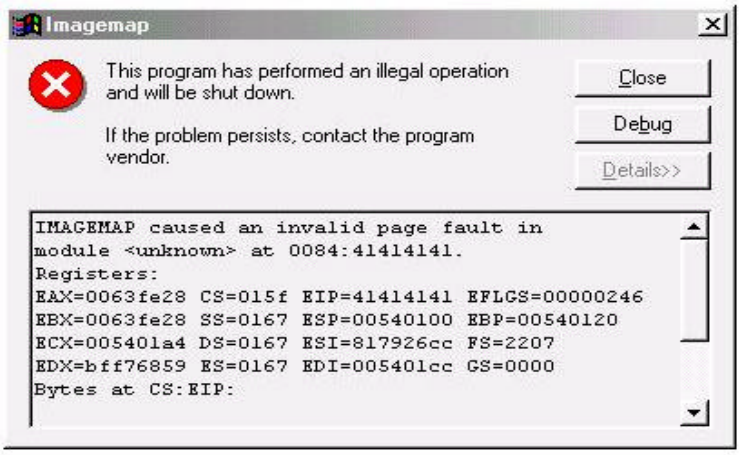
We now have a high confidence that imagemap.exe contains a buffer overflow vulnerability. As there is quite a bit of buffer space allocated above var_C0C we anticipate quite a large string (>2500 characters) will be required to corrupt the *ebp* register and create a anomaly within the running application. The client browser that we are using, Microsoft Internet Explorer 5 version 5.00.2614.3500, truncates a URL entry at about 2000 characters and we are forced to use a different method to query the server. This will be accomplished through a telnet connection to port 80 where we will use the HTTP GET method to submit the query string to the server.

Exhibit 17: Telnet Session to Port 80 Localhost with <~2700 character string>



The above represents our telnet request to the Personal Web Server (localhost 127.0.0.1). We are using the HTTP GET method to pass the following query string to the server: GET /cgi-bin/ imagemap.exe/~/<2565A's>?0,0. We observe the following page fault error message:

Exhibit 18: Imagemap.exe Page Fault Message



In addition to the error message the server stops responding. We believe this to be a fully exploitable buffer overflow as *eip* has been completely over written.

Chapter 7

Conclusion

The real problem is not whether machines think but whether men do.

- B. F. Skinner

With the explosive growth of computer and networking technology information we have become almost totally reliant these systems to manage all of our information including that which is privileged and sensitive. This technology has enabled information management on a grand and global scale and has been the key driver in the huge increases in productivity over the last 15 years. The acceleration of information technology, being propelled by Moores law[1], has come at a price. Those who got in early are now the dominant players across an entire industry. Not only do they provide the products that serve as a foundation in information management they, in effect, have become entrusted with protecting our most sensitive information. One would think this an awesome responsibility however in light of continuing security flaws, in our most trusted software, nothing could be further from the truth. Perhaps the most frightening reality associated with our trust in just a few vendors providing the majority of our critical software is that there is no simple method for the consumer to verify that the product they rely on is secure. Instead, we rely on the relentless assault of hackers to break released systems and hope we can patch our system before any real damage is done. I have demonstrated a technique that can be used as a tool to identify security flaws in proprietary software, one that could scan a binary image and tell us if there was a potential problem. Not only would we avoid the product, but the software vendors would soon have to confront the fact that lousy code cannot be obscured from the user.

1. The observation that the logic density of silicon integrated circuits has closely followed the curve (bits per square inch) = 2^(t - 1962) where t is time in years; that is, the amount of information storable on a given amount of silicon has roughly doubled every year since the technology was invented. This relation, first uttered in 1964 by semiconductor engineer Gordon Moore (who co-founded Intel four years later) held until the late 1970s, at which point the doubling period slowed to 18 months. The doubling period remained at that value through time of writing (late 1999). From: http://www.tuxedo.org/~esr/jargon/html/entry/Moore's-Law.html

7.1 Results and Contributions

We propose binary scanning as an approach for finding certain instances of the buffer overflow vulnerability as fast as possible in proprietary software. We make this bold assertion with little knowledge of the behavior of the buffer overflow post compile time or an understanding of the difficulties associated with reverse engineering using the disassembled binary as our only guide. For these reasons, our approach encompassed a journey through the body of knowledge associated with software security flaws in general and the characteristics of such flaws, as manifest in the binary image, specifically. Additionally, we have been treated to a healthy helping of assembly language, C programming and runtime stack behavior. Along the way we were able to propose a unique classification strategy for the buffer overflow vulnerability.

Traditionally the buffer overflow is viewed as a programming flaw. For our classification scheme we viewed the buffer overflow as an attack. This allows our taxonomy to be unique in that it classifies across two vectors. That is, it uses a two dimensional approach of system dependencies as well as attack dependencies to create a rich interwoven classification technique. We populate a select group of buffer overflow vulnerabilities within our classification scheme.

Armed with this knowledge we are able to leverage insight to develop a technique, based on reverse engineering, to find security flaws in the binary image without benefit of source code. The main contribution of this thesis is that the concept of scanning the binary image to find security faults is a valid technique that can be applied to proprietary software.

Our reverse engineering scheme allows us to focus directly on those areas of the disassembly that have the greatest potential for security related flaws such as the buffer overflow vulnerability. We investigated several of the so-called dangerous C functions and came to the conclusion that a suspicious '*buffer overflow*' signature is present and identifiable within a disassembly for each of the functions reviewed. Extending this conclusion, we developed a scanning algorithm based on the `sprintf()` function call. It was our insight into the use of `sprintf()`, with it's associated string literal providing a possible program entry point, that allowed us to demonstrate the success of this technique in a relatively short period of time. With a good idea of the context of user data being passed, combined with criteria that identifies a function as being suspicious, we were able to develop a compact and efficient scanning routine.

A substantial number of experiments were performed to systematically validate the implementation of our concept. We successfully demonstrated our technique, first on a simple test program where we illustrated two capabilities:

1) We identified a known buffer overflow security flaw that involved the unsafe use of `sprintf()`
2) We eliminated the concern with false positive identification by failing to identify those loca-

tions where *sprintf()* was used in a safe manner.

We then extended our technique to include contemporary enterprise class applications. We identified a potential buffer overflow vulnerability in the binary image of a contemporary Microsoft server application then went on to successfully demonstrate that vulnerability in the actual running application.

The main characteristic of this technique is the speed at which certain instances of a buffer overflow vulnerability can be identified. This distinguishes our method from other testing schemes and brute forcing techniques. While our method is not advocated over the traditional source review, it is able to make a bold statement. When one is able to find an instance of a certain type of fully exploitable buffer overflows in under (1) second, major developers of retail software should take notice. Programming flaws can be detected by much simpler methods than binary disassembly and reverse engineering when one has access to the source code. Despite this fact it is clear that these steps are, for the most part, being neglected. The fact that we were able to discover a security flaw, in a major application, using this technique speaks volumes not only of the lack attention paid to the security aspects of modern software development but to the power of reverse engineering. The success of this technique should not be measured by the fact that we were able to find a buffer overflow in a contemporary, server class product, it should be measured by the success of the concept itself.

7.1.1 Technique Limitations

The technique we present is limited in that it can only find buffer overflow vulnerabilities that are tied to the *sprintf()* function that is reading dynamic user input with the format character %s. We believe that the concept of binary scanning however can be extended to include wide range of security applications where the original source code is unavailable. The main limitation is that by scanning a disassembly, a compile time object, one will miss the vulnerabilities that are tied to the run time environment.

7.2 Research Directions

Reverse engineering a disassembled binary image is a very powerful technique with the main limitations being the potential time required, the expanded size of the disassembly and the level of expertise required. All three of these constraints scale well when machine automated, a fact we demonstrated with our simple program. We believe that other programs could be developed to cope with the more challenging problems associated with other abused C functions such as *strcpy()* and *strncpy()*. The challenge with these functions quickly becomes one of structure reconstruction when attempting to tease critical details from a disassembly.

7.2.1 Structure Reconstruction

Applications store their data not in discrete variables but in large structures that are passed around between functions. Many overflows happen within the context of the structure and without knowing what we are overwriting it is very hard to determine if the condition is an exploitable one. In addition, most overflows and security related problems occur within dynamic memory, on the heap. This memory region is used to hold large structures that contain connection data, error strings etc. In order to check function calls such as `strncpy()` one has to be able to estimate the size of each individual structure member. We believe that when one is able to rebuild the actual structures within a disassembly a much better picture will emerge as to how and where user input finds it way into a function. Structure reconstruction becomes an even bigger issue when reverse engineering C++ object oriented programs.

7.2.2 Class Reconstruction

Within the C++ programming language structures play a big part of object definition, after all a class is nothing more than a collection of functions that all use the same structure and all are completely lost at compile time. Many classes have an associated *vtable* or listing of memory locations associated with the properties and methods implemented in the class interface. In other words, this table can provide the reverse engineer a listing of all functions that access a given structure (i.e. the class itself). With this information is certainly possible to imagine that automated techniques may exist for the reconstruction of class data structures and with it the reconstruction of individual member boundaries. As we move more towards a more object oriented approach where large structures are defined as part of the object itself, structure reconstruction becomes a key element in the reverse engineering of contemporary code.

7.3 Final Thoughts

We have focused on a single security issue based on a small set of library functions that handled single variables and were able to demonstrate a security auditing concept using the power of reverse engineering. The area we focused on, the buffer overflow vulnerability, is a vanishing species, as the problem has been documented for over (10) years and for the most part is trivial to exploit. The push is on across the major suppliers of software to put an end to this type of programming flaw. Yes, there is legacy code, but a whole new frontier awaits the reverse engineer.

The modern program paradigm is a move to embrace object oriented design. Therefore new code, for the most part, is object oriented C++, that makes extensive use of the Standard Template Library (STL) for string, stream and container manipulation. We propose that there exists undiscovered exploitable security pitfalls within the family of STL constructs, as they manipulate user input, and that these new security flaws will manifest themselves as heap overruns. Heap overruns, due to their elusive nature,

will make stress testing useless and therefore it will be the reverse engineers with the advantage. As heap overflows are almost entirely compiler specific, the reverse engineers will have the advantage as they will be able to document the nuances of their compiler behavior for themselves. Then there is the area of writing exploits themselves. As the new CPU architectures move to non-executable data pages as a standard feature, shell code will become a thing of the past. The future attacker will not insert a new subroutine within corrupted stack space, they will subvert the logic of the application itself (i.e. bool paswrdValidated == true). In attempting to corrupt application logic the advantage will again go to the reverse engineer. Without this skill, exploitation of future proprietary applications will be close to impossible.

Automated binary disassembly must certainly be an area of interest within the intelligence and security community due its ability to open doors that would otherwise be shut. We have forwarded the premise that vendors hide their security flaws, or lack of security, by labeling their code as a trade secret, releasing only a compiled binary file. At the same time we can extend this reasoning to developers of our most secure software, the software based security systems themselves. All of these systems are based on the principle of “security by obscurity” as well. The concepts on which these programs are based are hidden within the program logic itself, logic which we believe can and will be corrupted in future attacks. All that is needed is a piece of binary code and a disassembler, assembler code cannot be made more difficult. It can be encrypted, obfuscated or made difficult in general but at some point the machine has to be able to read the instructions and if the machine can so can the reverse engineer: you can run but you can’t hide.

“Where do you want to go today?.....”

References

[1] D. FARMER. Abstract: *"Shall we dust Moscow?"* (Security survey of key internet hosts & various semi-relevant reactions), December 1996. Published on-line at http://www.fish.com/survey/.

[2] Computer Security Institute. Issues and trends: *"2000 CSI/FBI computer crime and security survey"*, March 2000.

[3] WILLIAM R. CHESWICK and STEVEN M. BELLOVIN. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[4] CRISPIN COWAN, PERRY WAGLE, CALTON PU, STEVE BEATTIE, and JONATHAN WALPOLE. *"Buffer overflows: Attacks and defenses for the vulnerability of the decade"*. In Proc. 2000 DARPA Information Survivability Conf. and Exp. (DISCEX '00), pages 154{163. IEEE Comp. Soc., 1999.

[5] M.W. EICHIN, J.A. ROCHLIS, *"With microscope and tweezers: an analysis of the Internet virus of Nov. 1988,"* 1989 IEEE Symp. Security and Privacy.

[6] JEFF COLLYER. *"Risks of unchecked input in C programs"*. Forum on Risks to the Public in Computers and Related Systems, ACM Committee on Computers and Public Policy, Volume 7, Issue 74 (10 Nov 1988), lines 200--231.

[7] ELIAS LEVY (Aleph One). *"Smashing the stack for fun and profit"*. On-line. Phrack Online. Volume 7, Issue 49, File 14 of 16. Available: www.fc.net/phrack/, November 9 1996.

[8] G. HELMER, *"Incomplete list of Unix vulnerabilities"*, http://www.cs.iastate.edu/~ghelmer/unixsecurity/unix_vuln.html.

[9] http://www.infilsec.com/vulnerabilities/.

[10] The bugtraq mailing list, http://www.securityfocus.com/

[11] Caida.org, *"CAIDA Analysis of Code-Red"*, http://www.caida.org/analysis/security/code-red/ , August 2001.

[12] *Rootshell*, http://www.rootshell.com

[13] *Fyodor's Playhouse*, http://www.insecure.org

[14] *Legacy Hacker Site* http://www.jabukie.com/The_Legacy_Main_Page.htm

[15] Nipc.gov, *"E-CommerceVulnerabilities"*, http://www.nipc.gov/warnings/advisories/2001/01-003.htm

[16] idefense.com, *''Israeli-Palestinian Cyber Conflict (IPCC)''*, www.idefense.com

[17] DAN FARMER AND WIETSE VENEMA, *"Improving the Security of Your Site by Breaking Into It"*, USENET posting (Dec. 1993)

[18] TSUTOMU SHIMOMURA and JOHN MARKOFF. *"Takedown"*. Hyperion Books, 1996.

[19] KATIE HAFNER and JOHN MARKOFF. *"Cyberpunk: Outlaws and Hackers on the Computer Frontier"*. Touchstone, 1992.

[20] SIMSON GARFINKEL and EUGENE SPAFFORD. *"Practical Unix and Internet Security"*. O'Reily and Associates, second edition, 1996.

[21] PETROSKI, H. 1992. *"To Engineer is Human: The Role of Failure in Successful Design"*. Vintage Books, New York, NY, 1992.

[22] PETER G. NEUMANN, "*Computer System Security Evaluation"*, 1978 National Computer Conference Proceedings(AFIPS Conference Proceedings 47), pp. 1087-1095, June 1978

[23] BREHMER,C.L.AND CARL, J. R. 1993. "*Incorporating IEEE Standard 1044 into your anomaly tracking process"*. CrossTalk, J. Defense Software Engineering, 6, (Jan. 1993), 9-16.

[24] CHILLAREGE, R., BHANDARI, I. S., CHAAR, J. K., HALLI-DAY, M. J., MOEBUS, D. S., RAY,B.K.,AND WONG, M-Y. 1992. "*Orthogonal defect classification—a concept for in-process measurements"*. IEEE Trans. on Software Engineering 18, 11, (Nov. 1992), 943-956.

[25] FLORAC, W. A. 1992. *"Software Quality Measurement: A Framework for Counting Problems and Defects"*. CMU/SEI-92-TR-22, Software Engineering Institute, Pittsburgh, PA, (Sept.).

[26] R. BISBEY II and D. HOLLINGSWORTH, "*Protection Analysis Project Final Report"*, ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute (May, 1978).

[27] JIM CARLSTEAD, RICHARD BIBSEY II, and GERALD POPEK. "*Pattern-directed protection evaluation"*. Technical report, Information Sciences Institue,University of Southern California, June 1975.

[28] RICHARD BIBSEY, GERALD POPEK, and JIM CARLSTEAD. "*Inconsistency of a single data value overtime"*. Technical report, Information Sciences Institute,University of Southern California, December 1975.

[29] R.P. ABBOTt et al. "*Security Analysis and Enhancements of Computer Operating Systems"*. Technical Report NBSIR 76-1041, Institute for Computer Science and Technology, National Bureau of Standards, 1976.

[30] CARL LANDWHER et al. "*A taxonomy of computer program security flaws"*. Technical report, Naval Research Laboratory, November 1993.

[31] BRIAN MARICK. *"A survey of software fault surveys".* Technical Report UIUCDCS-R-90-1651, University of Illinois at Urbana-Champaign, December 1990.

[32] M. BISHOP, *"A Taxonomy of UNIX System and Network Vulnerabilities,"* Tech. Rep. CSE-95-10, Purdue University, May 1995.

[33] T. ASLAM, *"A Taxonomy of Security Faults in the Unix Operating System,"* M.S. Thesis, Purdue University, West Lafayette, IN, 1995.

[34] T. ASLAM, *"Use of a Taxonomy of Security Faults,"* Tech. Rep. 96-05, COAST Laboratory, Department of Computer Science, Purdue University, March 1996.

[35] KRSUL, I., *"Software Vulnerability Analysis,"* Ph.D. Thesis, Department of Computer Sciences, Purdue University, West Lafayette, IN (1998).

[36] M. CROSBIE, B. DOLE, T. ELLIS, I. KRSUL, and E. H. SPAFFORD, *"IDIOT - User's Guide,"* Tech. Rep. TR-96-050, COAST Laboratory, Purdue University, September 4, 1996.

[37] U. LINDQUIST and E. JONSSON, "*How to Systematically Classify Computer Security Intrusions,"* Proc. 1997 IEEE Symp. on Security and Privacy, Oakland, CA, May 4-7, 1997, pp. 154 - 163.

[38] P. G. NEUMANN and D. B. PARKER, "*A Summary of Computer Misuse Techniques,"* Proc. of the 13th National Computer Security Conference, Baltimore, MD, October 10-13, 1989, pp. 396-407.

[39] E. A. FISCH, "*Intrusion Damage Control and Assessment: A Taxonomy and Implementation of Automated Responses to Intrusive Behavior,"* Ph.D. Dissertation, Texas A&M University, College Station, TX, 1996.

[40] http://icat.nist.gov/icat.cfm

[41] http://www.cert.org/

[42] http://www.neohapsis.com/

[43] JOHN HOWARD, *"An Analysis of Security Incidents on the Internet 1989-1995,"* Carnegie Mellon University, April 1997

[44] B. W. KERNIGHAN and D. M. RITCHIE, *"The C Programming Language",* Prentice-Hall: Englewood Cliffs, NJ, 1978. Second edition, 1988

[45] DILDOG, "*The Tao of Windows Buffer Overflows*", http://www.newhackcity.net/win_buff_overflow/

[46] CRISPIN COWAN, CALTON PU, DAVE MAIER, HEATHER HINTON, PEAT BAKKE, STEVE BEATTIE, AARONGRIER, PERRY WAGLE, and QIAN ZHANG. "*StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*". In 7th USENIX Security Conference, pages 63-77, San Antonio, TX, January 1998.

[47] MUDGE. *How to Write Buffer Overflows.* http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html

[48] ALFRED HUGER. *Historical Bugtraq Question*. Bugtraq mailing list, http://geek-girl.com/bugtraq, September 30 1999.

[49] MIXTER. *Writing Buffer Overflow Exploits-a tutorial for beginners.* http://mixter.warrior2k.com/exploit.txt

[50] KLOG. "*The Frame Pointer Overwrite*". Online. Phrack Online. Volume 9, Issue 55, File 8 of 19. Available: www.fc.net/phrack/, September 9, 1999.

[51] ZAN. "*winnt/2k remote shellcode*". http://www.deepzone.org/editions/others/shell-e.pdf

[52] SCUT/TESO. "*Writing MIPS/IRIX Shellcode*". Online. Phrack Online. Volume 10, Issue 56, File 15 of 16. Available: www.fc.net/phrack/, September 9, 1999.

[53] SMILER. "*The Art of Writing Shellcode*". Online. http://193.226.6.55/papers/Bufferoverflow/art-shellcode.txt

[54] ALFREDV. AHO, R.HOPCROFT, and JEFFREYD. ULLMAN. *"Compilers: Principles, Techniques and Tools"*. Addison-Wesley, Reading, Mass., 1985.

[55] STEFAN AXELSSON, *"A Comparison of the Security of Windows NT and UNIX"*, 1998 http://www.securityfocus.com/data/library/nt-vs-unix.pdf

[56] NISHAD HERATH, (Joey_) *"Advanced Windows NT Security"*, in Black Hat Asia Conference, 2000. http://www.blackhat.com/html/bh-asia-00/bh-europe-00-speakers.html#Joey

[57] HALVAR FLAKE, "Auditing binaries for security vulnerabilities"in Black Hat Asia Conference, 2000. http://www.blackhat.com/html/bh-asia-00/bh-europe-00-speakers.html

[58] BARNABY JACK, aka (dark spyrit), *"Win32 Buffer Overflows (Location, Exploitation, and Prevention)"*, Online. Phrack Online. Volume 9, Issue 55, File 15 of 19. Available: http://www.fc.net/phrack/, September 9, 1999.

[59] RYAN RUSSEL, RAIN FOREST PUPPY, ELIAS LEVY, BLUE BOAR, DAN KAMINSKY, OLIVER FRIEDRICHS, RILEY ELLER, GREG HOGLUND, JEREMY RAUCH, and GEORGI GUNINSKI, *"Hack Proofing Your Network Internet Tradecraft"*, Syngress, 2000.

[60] JOEL SCAMBRAY, STUART MCCLURE, GEORGE KURTZ, *"Hacking Exposed, Network Security Secrets & Solutions"* , Second Edition, Osborne/McGrawHill , 2001 Nishad Herath, (Joey_) Advanced Windows NT Security, in Black Hat Asia Conference, 2000. http://www.blackhat.com/html/

[61] SCUT/TEAM TESO, *"Exploiting Format String Vulnerabilities"*, Available: http://teso.scene.at/articles/formatstring/

[62] http://www.cert.org

[63] E. CHIKOFSKY and J. CROSS. *"Reverse engineering and design recovery: A taxonomy"*. IEEE Software, 7:13–17, Jan. 1990.

[64] M.WEISER. *"Program slicing"*. IEEE Transactions on Software Engineering, 10(4):352–357, July 1984.

[65] T. BALL and J. LARUS.*"Optimally profiling and tracing programs"*. Transactions of Programming Languages and Systems, 16(4):1319–1360, July 1994.

[66] L. FREEMAN. *"Don't let Missing Source Code stall your Year 2000 Project"*. Year 2000 Survival Guide.

[67] R.N. HORSPOOL and N. MAROVAC. *"An approach to the problem of detranslation of computer programs"*. The Computer Journal, 23(3):223-229, 1979.

[68] H. SWARTZ. *"The case for reverse engineering."* Business Computer Systems, 3(12):22-25, December 1984.

[69] http://www.nsfocus.com/english/homepage/sa01-03.htm

[70] C. CIFUENTES and K.J. GOUGH. "*Decompilation of binary programs.*"Software Practice and Experience *"*, 25(7):811-829, July 1995.

[71] CRISTINA CIFUENTES. *"Partial Automation of an Integrated Reverse Engineering Environment of Binary Code"* , Research Paper

[72] CRISTINA CIFUENTES. *"Reverse Compilation Techniques"*, Ph.D. Disseration; Queensland University of Technology. 1994

[73] NICOLE LaROCK DECKER. *"Buffer Overflows: Why, How and Prevention."* Available http://rr.sans.org/threats/buffer_overflow.php, November 13, 2000.

[74] DAVID LAROCHELLE AND DAVID EVANS. *"Statically Detecting Likely Buffer Overflow Vulnerabilities."* In 2001 USENIX Security Symposium, Washington, D. C., August 13-17, 2001.

[75] http://dmoz.org/Computers/Programming/Software_Testing/Software_Testing_Tools/

[76] "*A Brief History of the Internet and Related Networks*"; http://www.isoc.org/internet/history/cerf.shtml

[77] BRUCE SCHNEIER. *"The Process of Security."; http://www.infosecuritymag.com/articles/april00/columns_cryptorhythms.shtml*

[78] ROBERT W. SEBESTA. *"Concepts of Programming Languages.";* Addison-Wesley, 1999.

[79] RIK FARROW. *"Blocking Buffer Overflow Attacks".* http://www.networkmagazine.com/article/NMG20000511S0015

[80] http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci550882,00.html

[81] http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci550815,00.html

[82] COL. STANISLAV LUNEV. "Red Mafia Operating in the U.S. – Helping Terrorists"; http://www.newsmax.com/archives/articles/2001/9/28/90942.shtml

[83] ALAN CLEMENTS. "*Brief History of Microprocessors";* http://wheelie.tees.ac.uk/users/a.clements/History/History.htm

Appendix A

Archive/Compressed Archives		
Gnu tar format	gtar	application/x-gtar
4.3BSD tar format	tar	application/x-tar
POSIX tar format	ustar	application/x-ustar
Old CPIO format	bcpio	application/x-bcpio
POSIX CPIO format	cpio	application/x-cpio
UNIX sh shell archive	shar	application/x-shar
DOS/PC - Pkzipped archive	zip	application/zip
Macintosh Binhexed archive	hqx	application/mac-binhex40
Macintosh Stuffit Archive	sit sea	application/x-stuffit
Fractal Image Format	fif	application/fractals
Binary, UUencoded	bin uu	application/octet-stream
PC executable	exe	application/octet-stream
WAIS "sources"	src wsrc	application/x-wais-source
NCSA HDF data format	hdf	application/hdf
Downloadable Program/Scripts		
Javascript program	js ls mocha	text/javascript
		application/x-javascript
VBScript program		text/vbscript
UNIX bourne shell program	sh	application/x-sh
UNIX c-shell program	csh	application/x-csh
Perl program	pl	application/x-perl
Tcl (Tool Control Language) program	tcl	application/x-tcl
Atomicmail program scripts (obsolete)		application/atomicmail
Slate documents - executable enclosures (*BBN*)		application/slate
Undefined binary data (often executable progs)		application/octet-stream
RISC OS Executable programs (*ANT Limited*)		application/riscos
Animation/Multimedia		
Andrew Toolkit inset		application/andrew-inset
FutureSplash vector animation (*FutureWave*)	spl	application/futuresplash
mBED multimedia data (*mBED*)	mbd	application/mbedlet
Macromedia Shockwave (*Macromedia*)		application/x-director
Sizzler real-time video/animation		application/x-sprite
PowerMedia multimedia (*RadMedia*)	rad	application/x-rad-powermedia
Presentation		
PowerPoint presentation (*Microsoft*)	ppz	application/mspowerpoint
PointPlus presentation data (*Net Scene*)	css	application/x-pointplus
ASAP WordPower (*Software Publishing Corp.*)	asp	application/x-asap
Astound Web Player multimedia data (*GoldDisk*)	asn	application/astound
Special Embedded Object		
OLE script e.g. Visual Basic (*Ncompass*)	axs	application/x-olescript
OLE Object (*Microsoft/NCompass*)	ods	application/x-oleobject
OpenScape OLE/OCX objects (*Business@Web*)	opp	x-form/x-openscape
Visual Basic objects (*Amara*)	wba	application/x-webbasic
Specialized data entry forms (*Alpha Software*)	frm	application/x-alpha-form
client-server objects (*Wayfarer Communications*)	wfx	x-script/x-wfxclient
General Applications		
Undefined binary data (often executable progs)		application/octet-stream
CALS (U.S. D.O.D data format - RFC 1895)		application/cals-1840
Pointcast news data (*Pointcast*)	pcn	application/x-pcn
Excel spreadsheet (*Microsoft*)		application/vnd.ms-excel
		application/x-msexcel
		application/ms-excel
PowerPoint (*Microsoft*)	ppt	application/vnd.ms-powerpoint
		application/ms-powerpoint
Microsoft Project (*Microsoft*)		application/vnd.ms-project
Works data (*Microsoft*)		application/vnd.ms-works
MAPI data (*Microsoft*)		application/vnd.ms-tnef
Artgallery data (Microsoft)		application/vnd.artgalry
SourceView document (*Dataware Electronics*)	svd	application/vnd.svd
Truedoc (*Bitstream*)		application/vnd.truedoc

MPEG audio	mpa abs mpega	audio/x-mpeg
MPEG-2 audio	mp2a mpa2	audio/x-mpeg-2
compressed speech (*Echo Speech Corp.*)	es	audio/echospeech
Toolvox speech audio (*Voxware*)	vox	audio/voxware
RapidTransit compressed audio (*Fast Man*)	lcc	application/fastman
Realaudio (*Progressive Networks*)	ra ram	application/x-pn-realaudio
NIFF music notation data format		application/vnd.music-niff
MIDI music data	mmid	x-music/x-midi
Koan music data (*SSeyo*)	skp	application/vnd.koan
		application/x-koan
Speech synthesis data (*MVP Solutions*)	talk	text/x-speech
Video Types		
MPEG video	mpeg mpg mpe	video/mpeg
MPEG-2 video	mpv2 mp2v	video/mpeg-2
Macintosh Quicktime	qt mov	video/quicktime
Microsoft videc	avi	video/x-msvidec
SGI Movie format	movie	video/x-sgi-movie
VDOlive streaming video (*VDOnet)*	vdo	video/vdo
Vivo streaming video (*Vivo software*)	viv	video/vnd.vivo
		video/vivo
Special HTTP/Web Application Types		
Proxy autoconfiguration (*Netscape* browsers)	pac	application/x-ns-proxy-autoconfig
		application/x-www-form-urlencoded
		application/x-www-local-exec
(*Netscape* extension)		multipart/x-mixed-replace
		multipart/form-data
Netscape Cooltalk chat data (*Netscape*)	ice	x-conference/x-cooltalk
Interactive chat (*Ichat*)		application/x-chat
Application Types		
Text-Related		
PostScript	ai eps ps	application/postscript
Microsoft Rich Text Format	rtf	application/rtf
Adobe Acrobat PDF	pdf	application/pdf
		application/x-pdf
Maker Interchange Format (*FrameMaker*)	mif	application/vnd.mif
		application/x-mif
Troff document	t tr roff	application/x-troff
Troff document with MAN macros	man	application/x-troff-man
Troff document with ME macros	me	application/x-troff-me
Troff document with MS macros	ms	application/x-troff-ms
LaTeX document	latex	application/x-latex
Tex/LateX document	tex	application/x-tex
GNU TexInfo document	texinfo texi	application/x-texinfo
TeX dvi format	dvi	application/x-dvi
MacWrite document	??	application/macwriteii
MS word document	??	application/msword
WordPerfect 5.1 document	??	application/wordperfect5.1
SGML application (RFC 1874)		application/sgml
Office Document Architecture	oda	application/oda
Envoy Document	evy	application/envoy
Wang Info. Tranfer Format (*Wang*)		application/wita
DEC Document Transfer Format (*DEC*)		application/dec-dx
IBM Document Content Architecture (*IBM*)		application/dca-rft
CommonGround Digital Paper (*No Hands Software*)		application/commonground
FrameMaker Documents (*Frame*)	doc fm frm frame	application/vnd.framemaker
		application/x-framemaker
Remote printing at arbitrary printers (RFC 1486)		application/remote-printing

Net Install - software install (*20/20 Software*)	ins	application/x-net-install
Carbon Copy - remote control/access (*Microcom*)	ccv	application/ccv
Spreadsheets (*Visual Components*)	vts	workbook/formulaone
Cybercash digital money (*Cybercash*)		application/cybercash
Format for sending generic Macintosh files		application/applefile
Active message -- connect to active mail app.		application/activemessage
X.400 mail message body part (RFC 1494)		application/x400-bp
USENET news message id (RFC 1036)		application/news-message-id
USENET news message (RFC 1036)		application/news-transmission
Multipart Types (mostly email)		
Messages with multiple parts		multipart/mixed
Messages with multiple, alternative parts		multipart/alternative
Message with multiple, related parts		multipart/related
Multiple parts are digests		multipart/digest
For reporting of email status (admin.)		multipart/report
Order of parts does not matter		multipart/parallel
Macintosh file data		multipart/appledouble
Aggregate messages; descriptor as header		multipart/header-set
Container for voice-mail		multipart/voice-message
HTML FORM data (see Ch. 9 and App. B)		multipart/form-data
Infinite multiparts - See Chapter 9 (*Netscape*)		multipart/x-mixed-replace
Message Types (mostly email)		
MIME message		message/rfc822
Partial message		message/partial
Message containing external references		message/external-body
Message containing USENET news		message/news
HTTP message		message/http
2D/3D Data/Virtual Reality Types		
WIRL - VRML data (VREAM)		x-world/x-vrml
Play3D 3d scene data (*Play3D*)	vrw	x-world/x-vream
Viscape Interactive 3d world data (*Superscape*)	p3d	application/x-p3d
WebActive 3d data (*Plastic Thought*)	svr	x-world/x-svr
QuickDraw3D scene data (*Apple*)	wvr	x-world/x-wvr
	3dmf	x-world/x-3dmf
Scientific/Math/CAD Types		
Chemical types -- information about chemical models		chemical/* (several subtypes)
Mathematica notebook	ma	application/mathematica
Computational meshes for numerical simulations	msh	x-model/x-mesh
Vis5D 5-dimensional data	v5d	application/vis5d
IGES models-CAD/CAM (CGM) data	igs	application/iges
Autocad WHIP vector drawings	dwf	drawing/x-dwf
Largely Platform-Specific Types		
Silicon Graphics Specific Types		
Showcase Presentations	showcase slides sc sho show	application/x-showcase
Insight Manual pages	ins insight	application/x-insight
Iris Annotator data	ano	application/x-annotator
Directory Viewer	dir	application/x-dirview
Software License	lic	application/x-enterlicense
Fax manager file	faxmgr	application/x-fax-manager
Fax job data file	faxmgrjob	application/x-fax-manager-job
IconBook data	icnbk	application/x-iconbook
?	wb	application/x-inpview
Installable software in 'inst' format	inst	application/x-install
Mail folder	mail	application/x-mailfolder
?	pp ppages	application/x-ppages
Data for printer (via lpr)	sgi-lpr	application/x-sgi-lpr
Software in 'tardist' format	tardist	application/x-tardist
Software in compressed 'tardist' format	ztardist	application/x-ztardist
WingZ spreadsheet	wkz	application/x-wingz
Open Inventor 3-D scenes	iv	graphics/x-inventor

Appendix B

PORT NUMBER	SERVICE
1	tcpmux
5	rje
7	echo
9	discard
11	systat
13	daytime
15	netstat
17	qotd
18	send/rwp
19	chargen
20	ftp-data
21	ftp
22	ssh, pcAnywhere
23	Telnet
25	SMTP
27	ETRN
29	msg-icp
31	msg-auth
33	dsp
37	time
38	RAP
39	rlp
42	nameserv, WINS
43	whois, nickname
49	TACACS, Login Host Protocol
50	RMCP, re-mail-ck
53	DNS
57	MTP
59	NFILE
63	whois++
66	sql*net
67	bootps
68	bootpd/dhcp
69	Trivial File Transfer Protocol (tftp)
70	Gopher
79	finger
80	www-http
88	Kerberos, WWW
95	supdup
96	DIXIE
98	linuxconf
101	HOSTNAME
102	ISO, X.400, ITOT
105	cso
106	poppassd
109	POP2
110	POP3
111	Sun RPC Portmapper
113	identd/auth
115	sftp
117	uucp
119	NNTP
120	CFDP
123	NTP
124	SecureID
129	PWDGEN
133	statsrv
135	loc-srv/epmap
137	netbios-ns
138	netbios-dgm (UDP)
139	NetBIOS
143	IMAP
144	NewS
152	BFTP
153	SGMP
161	SNMP
175	vmnet
177	XDMCP
178	NextStep Window Server
179	BGP
180	SLmail admin
199	smux
210	Z39.50
218	MPP
220	IMAP3
259	ESRO
264	FW1_topo
311	Apple WebAdmin
350	MATIP type A
351	MATIP type B
363	RSVP tunnel
366	ODMR (On-Demand Mail Relay)
387	AURP (AppleTalk Update-Based Routing Protocol)
389	LDAP
407	Timbuktu
434	Mobile IP
443	ssl
444	snpp, Simple Network Paging Protocol
445	SMB
458	QuickTime TV/Conferencing
468	Photuris
500	ISAKMP, pluto
512	biff, rexec
513	who, rlogin
514	syslog, rsh
515	lp, lpr, line printer

PORT NUMBER	SERVICE
517	talk
520	RIP (Routing Information Protocol)
521	RIPng
522	ULS
531	IRC
543	KLogin, AppleShare over IP
545	QuickTime
548	AFP
554	Real Time Streaming Protocol
555	phAse Zero
563	NNTP over SSL
575	VEMMI
581	Bundle Discovery Protocol
593	MS-RPC
608	SIFT/UFT
626	Apple ASIA
631	IPP (Internet Printing Protocol)
635	mountd
636	sldap
642	EMSD
648	RRP (NSI Registry Registrar Protocol)
655	tinc
660	Apple MacOS Server Admin
666	Doom
674	ACAP
687	AppleShare IP Registry
700	buddyphone
705	AgentX for SNMP
901	swat, realsecure
993	s-imap
995	s-pop
1062	Veracity
1080	SOCKS
1085	WebObjects
1227	DNS2Go
1243	SubSeven
1338	Millennium Worm
1352	Lotus Notes
1381	Apple Network License Manager
1417	Timbuktu
1418	Timbuktu
1419	Timbuktu
1433	Microsoft SQL Server
1434	Microsoft SQL Monitor
1494	Citrix ICA, MS Terminal Server
1503	T.120
1521	Oracle SQL
1525	prospero
1526	prospero
1527	tlisrv
1604	Citrix ICA, MS Terminal Server
1645	RADIUS Authentication
1646	RADIUS Accounting
1680	Carbon Copy
1701	L2TP/LSF
1717	Convoy
1720	H.323/Q.931
1723	PPTP control port
1755	Windows Media .asf
1758	TFTP multicast
1812	RADIUS server
1813	RADIUS accounting
1818	ETFTP
1973	DLSw DCAP/DRAP
1985	HSRP
1999	Cisco AUTH
2001	glimpse
2049	NFS
2064	distributed.net
2065	DLSw
2066	DLSw
2106	MZAP
2140	DeepThroat
2301	Compaq Insight Management Web Agents
2327	Netscape Conference
2336	Apple UG Control
2427	MGCP gateway
2504	WLBS
2535	MADCAP
2543	sip
2592	netrek
2727	MGCP call agent
2628	DICT
2998	ISS Real Secure Console Service Port
3000	Firstclass
3031	Apple AgentVU
3128	squid
3130	ICP
3150	DeepThroat
3264	ccmail
3283	Apple NetAssitant
3288	COPS
3305	ODETTE
3306	mySQL
3389	NT Terminal Server
3521	netrek
4000	icq, command-n-conquer
4321	rwhois

PORT NUMBER	SERVICE
4333	mSQL
4827	HTCP
5004	RTP
5005	RTP
5010	Yahoo! Messenger
5060	SIP
5190	AIM
5500	securid
5501	securidprop
5423	Apple VirtualUser
5631	PCAnywhere data
5632	PCAnywhere
5800	VNC
5801	VNC
5900	VNC
5901	VNC
6000	X Windows
6112	BattleNet
6502	Netscape Conference
6667	IRC
6670	VocalTec Internet Phone, DeepThroat
6699	napster
6776	Sub7
6970	RTP
7007	MSBD, Windows Media encoder
7070	RealServer/QuickTime
7778	Unreal
7648	CU-SeeMe
7649	CU-SeeMe
8010	WinGate 2.1
8080	HTTP
8181	HTTP
8383	IMail WWW
8875	napster
8888	napster
10008	cheese worm
11371	PGP 5 Keyserver
13223	PowWow
13224	PowWow
14237	Palm
14238	Palm
18888	LiquidAudio
21157	Activision
23213	PowWow
23214	PowWow
23456	EvilFTP
26000	Quake
27001	QuakeWorld
27010	Half-Life
27015	Half-Life
27960	QuakeIII
30029	AOL Admin
31337	Back Orifice
32777	rpc.walld
40193	Novell
41524	arcserve discovery
45000	Cisco NetRanger postofficed
Multicast	hidden
ICMP Type	hidden
32773	rpc.ttdbserverd
32776	rpc.spray
32779	rpc.cmsd
38036	timestep

Appendix C

Case: 1

All-Mail multiple SMTP buffer overflows

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2000-0985 (under review); bugtraq 1789

Description:
Nevis System All-Mail version 1.1 is vulnerable to multiple buffer overflows. All-Mail is a mail server written for Windows. By sending long commands such as "mail from" or "rcpt to" a remote attacker can overflow a buffer and execute arbitrary code on the system. Several static buffers in the SMTP component are susceptable. Overflow input is sent remotely to TCP port 25.

Intent:
a) Penetrate; Remote: get root

b) DOS; Remote

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 25; SMPT

Target Hardware:
Windows: 2000, NT4

Target Software:
Application; All-Mail (1.1)

Case: 2

CesarFTP long command buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2001-0826 (under review); bugtraq id 2972

Description:
CasesarFTP is a Windows FTP server from ACLogic. By sending a long string of characters argumenting any of several FTP commands, an attacker can cause a stack overflow. A remote user could supply a properly-structured argument to an affected command, designed to exceed the maximum length of the input buffer. The values stored in this buffer can overflow onto the stack, potentially overwriting the calling functions' return address with values that can alter the program's flow of execution.

Intent:
a) Penetrate; Remote: get root

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 21; FTP

Target Hardware:
Windows: NT

Target Software:
Application; ACLogic Caesar FTP 0.98b; *server.exe*

Case: 3

CiscoSecure ACS CSAdmin buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CVE-2000-1054; bugtraq id 1705

Description:
CiscoSecure ACS for Windows NT versions 2.4.2 and earlier are vulnerable to a buffer overflow in the CSAdmin software module. By sending an oversized packet to TCP port 2002, an unauthenticated remote attacker can overflow the buffer and execute arbitrary code or cause the CSAdmin software module to crash. The effects of this vulnerability vary, depending on the exact versions of Windows NT and CiscoSecure ACS on the server.

Intent:
a) Penetrate; Remote: get info?; illegal disk write b) DOS Remote

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 2002;

Target Hardware:
Windows: NT

Target Software:
Application; CiscoSecure ACS for Windows NT versions 2.4.2 and earlier

Case: 4

EFTP '.lnk' file buffer overflow

Source:
bugtraq 3330

Description:
Encrypted FTP (EFTP) is a client/server program developed by Khamil Landross and Zack Jones that allows users to transfer files securely and is based on the 448bit Blowfish Encryption Algorithm and the FTP protocol. EFTP version 2.0.7.337 is vulnerable to a buffer overflow. After uploading a *.lnk file containing a large amount of "A" characters and issuing a LIST command, a remote attacker can overflow a buffer and execute arbitrary commands on the system or launch a denial of service attack.

Intent:
a) Penetrate; Remote: get root

b) DOS; Remote

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 23; FTP

Target Hardware:
Windows: 95, 98, 2000, ME, NT4; Cisco iCDN 2.0

Target Software:
Application; Khamil Landross and Zack Jones EFTP 2.0.7.337

Case: 5

DocumentDirect GET buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2000-0826 (under review); bugtraq id 1657

Description:
A number of unchecked static buffers exist in Mobius' DocumentDirect for the Internet program. Depending on the data entered, arbitrary code execution or a denial of service attack could be launched under the privilege level of the corresponding service.

Buffer Overflow #1 - Issuing the following GET request will overflow DDICGI.EXE: GET/ddrint/bin/ddicgi.exe?[string at least 1553 characters long]=X HTTP/1.0

Buffer Overflow #2 - Entering a username consisting of at least 208 characters in the web authorization form will cause DDIPROC.EXE to overflow. If random data were to be used, a denial of service attack would be launched against the DocumentDirect Process Manager which would halt all services relating to it.

Buffer Overflow #3 - Issuing the following GET request will cause an access validation error in DDICGI.EXE:GET /ddrint/bin/ddicgi.exe HTTP/1.0\r\nUser-Agent: [long string of characters]\r\n\r\n

Intent:
a) Penetrate; Remote: get privilege

b) DOS; Remote

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 80; HTTP

Target Hardware:
Windows: NT4

Target Software:
Application; Mobius DocumentDirect for the Internet 1.2

Case: 6

FrontPage 98 Server Extensions DVWSSR.DLL file buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2000-0260 (under review); bugtraq id 1109

Description:
Microsoft FrontPage 98 Server Extensions installs the file DVWSSR.DLL in the /_vti_bin/_vti_aut directory on Windows 95/98 and Windows NT Web servers. This file is normally used to connect to the site with the Microsoft InterDev program. A malicious user could overflow an unchecked buffer in the DVWSSR.DLL file to crash the server and execute arbitrary code.

Intent:
a) Penetrate; Remote: get privilege

b) DOS; Remote

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 80; HTTP

Target Hardware:
Windows: Microsoft FrontPage 98 Server Extensions for IIS, Microsoft FrontPage 98, Microsoft IIS 4.0, Microsoft NT Option Pack for NT 4.0, Microsoft InterDev 1.0, Microsoft Windows NT 4.0

Target Software:
Application; Microsoft FrontPage 98 Server Extension Dynamic Link Library (.DLL) File: *dvwssr.dll*

Case: 7

FrontPage Server Extensions Visual Studio RAD Support sub-component buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CVE-2001-0341; bugtraq id 2906

Description:
Microsoft FrontPage Server Extensions (FPSE) for Windows NT and Windows 2000 is vulnerable to a buffer overflow in the Visual Studio RAD (Remote Application Deployment) Support sub-component. FrontPage Server Extensions are components used in Microsoft Internet Information Server (IIS) versions 4.0 and 5.0. If the Visual Studio RAD Support sub-component is installed, a remote attacker can send a specially-crafted packet to the server to overflow a buffer. When *fp30reg.dll* receives a URL request that is longer than 258 bytes, a stack buffer overflow will occur. An attacker could exploit this vulnerability to execute arbitrary code on the system and possibly gain complete control over the affected Web server.

Intent:
a) Penetrate; Remote: get root

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 80; HTTP

Target Hardware:
Windows: FrontPage 2000 Server Extensions: All Versions, Microsoft IIS 4.0, Microsoft IIS 5.0, Windows 2000 Advanced Server, Windows 2000 Server, Windows 2000: All Versions, Windows NT 4.0

Target Software:
Application; Microsoft FrontPage 2000 Server Extension Dynamic Link Library (.DLL) File : *fp30reg.dll*

Case: 8

GuildFTPD SITE command buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2001-0770

Description:
GuildFTPD is a free Windows FTP server. GuildFTPD version 0.97 is vulnerable to a buffer overflow in the SITE command. By sending a SITE command containing 261 bytes or more, a remote attacker can overflow a buffer in 'sitecmd.dll' to execute arbitrary code on the system.

Intent:
a) Penetrate; Remote: get privilege

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 21; FTP

Target Hardware:
Windows: All versions

Target Software:
Application; GuildFTPD version 0.97 File: *sitecmd.dll*

Case: 9

IIS buffer overflow in HTR requests can allow remote code execution

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CVE-1999-0874; bugtraq id 0307

Description:
Microsoft Internet Information Server (IIS) version 4.0 is vulnerable to a denial of service attack caused by a buffer overflow involving the way that .HTR, .STM, and .IDC files are processed. IIS version 4.0 can perform various server-side processing with specific file types. Requests for files ending with .HTR, .STM, and .IDC extensions are passed to the appropriate external DLL for processing. By sending a mal-formed request, an attacker can overflow a buffer and cause the service to crash. It may be possible for an attacker to use this vulnerability to execute arbitrary code on the system.

Intent:
a) Penetrate; Remote: get privilege

b) DOS; crash vulnerable IIs processes

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 80; HTTP

Target Hardware:
Windows: Microsoft IIS 4.0 ; Cisco Building Broadband Service Manager 5.0; Cisco Call Manger 1.0; Cisco Call Manger 2.0; Cisco Call Manger 3.0; Cisco ICS 7750; Cisco IP/VC 3540; Cisco Unity Server 2.0; Cisco Unity Server 2.2; Cisco Unity Server 2.3; Cisco Unity Server 2.4; Cisco uOne 1.0; Cisco uOne 2.0; Cisco uOne 3.0; Cisco uOne 4.0; Microsoft BackOffice 4.0; Microsoft BackOffice 4.5; Microsoft Windows NT 4.0 Option Pack

Target Software:
Application; Microsoft IIS 4.0 Server File: *ism.dll*

Case: 10

IIS idq.dll ISAPI extension buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CVE-2001-0500;

Description:
Microsoft Internet Information Server (IIS) versions 4.0, 5.0, and 6.0 beta are vulnerable to a buffer overflow in the handling of ISAPI (Internet Services Application Programming Interface) extensions. An unchecked buffer in the code that handles idq.dll ISAPI extensions in the Indexing Service for IIS could allow a remote attacker to overflow a buffer and execute code by sending a specially-crafted Indexing Service request. An attacker could exploit this vulnerability to gain complete control over the affected server.

This vulnerability is exploitable via the "Code Red" and "Code Red II" worm. The "Code Red" worm is a self-propagating worm that scans random IP addresses on port 80 searching for vulnerable Web servers. Once a vulnerable Web server is found, the worm performs malicious activity before propagating to other vulnerable hosts. The "Code Red II" worm does not deface Web sites, as the original version of the worm did, but it carries a more serious threat -- it contains a Trojan Horse payload, which could allow any remote attacker to further compromise infected systems. The "Code Red II" worm also has the ability to scan for vulnerable hosts much faster than previous versions, which has already been reported to cause failures in certain network components by overloading them with network traffic.

Intent:
a) Penetrate; Remote: get root

b) DOS; Remote

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 80; HTTP

Target Hardware:
Windows: Microsoft IIS 4.0; Microsoft IIS 5.0; Microsoft IIS 6.0 beta; Microsoft Index Server 2.0; Microsoft Indexing Service All versions; Windows 2000: All Versions; Windows NT 4.0; Windows NT: All Versions; Windows XP beta

Target Software:
Application; Microsoft Internet Information Server (IIS) File: *idq.dll* ISAPI extension

Case: 11

IIS 5.0 ISAPI Internet Printing Protocol extension buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CVE-2001-0241; bugtraq id 2674

Description:
Microsoft Internet Information Server (IIS) version 5.0 installed on Microsoft Windows 2000 is vulnerable to a buffer overflow in the handling of ISAPI (Internet Services Application Programming Interface) extensions. An unchecked buffer exists in the code that handles input parameters for the Internet Printing Protocol (IPP) ISAPI extension. Windows 2000 Internet printing ISAPI extension contains msw3prt.dll which handles user requests. Due to an unchecked buffer in msw3prt.dll, the following maliciously crafted HTTP .printer request will allow the execution of arbitrary code. *GET /NULL.printer HTTP/1.0 Host: [buffer]*. Where [buffer] is aprox. 420 characters. Typically a web server would stop responding in a buffer overflow condition; however, once Windows 2000 detects an unresponsive web server it automatically performs a restart. Therefore, the administrator will be unaware of this attack. An attacker can use this vulnerability to gain complete control over the affected server.

Intent:
a) Penetrate; Remote: get privilege

Offensive Access Requirements:
Standard Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 80; HTTP

Target Hardware:
Windows: Microsoft IIS 5.0: Microsoft Windows 2000 Advanced Server; Microsoft Windows 2000 Advanced Server SP1; Microsoft Windows 2000 Advanced Server SP2; Microsoft Windows 2000 Datacenter Server SP1; Microsoft Windows 2000 Datacenter Server SP2; Microsoft Windows 2000 Professional; Microsoft Windows 2000 Professional SP1; Microsoft Windows 2000 Professional SP2; Microsoft Windows 2000 Server; Microsoft Windows 2000 Server SP1; Microsoft Windows 2000 Server SP2

Target Software:
Application; Microsoft IIS 5.0 Server File: *msw3prt.dll*

Case: 12

IIS remote FTP buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CVE-1999-0349; bugtraq id 0192

Description:
There is a Denial of Service / Buffer Overflow condition in Microsoft IIS4 FTP service when using the Name List (NLST) command. A user having user or anonymous access to the FTP server may initiate this attack. Connecting to the FTP server and issuing an ls command with 316 characters will cause the inet-info.exe service to crash (and the connection to be reset). Passing more than 316 characters will cause the stack to be overwritten. Up to 505 characters may be passed.

Intent:
a) DOS; Remote: crash/freeze host

Offensive Access Requirements:
Server must either have anonymous access rights or an attacker must have an account

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 21; FTP

Target Hardware:
Windows: Microsoft IIS 3.0; Microsoft IIS 4.0; Microsoft Personal Web Server 1.0; Windows NT: All Versions

Target Software:
Application; Microsoft IIS (Internet Information Server) FTP service File: *inetinfo.exe*

Case: 13

IIS specially-crafted SSI directives buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2001-0506; bugtraq id 3190

Description:
Microsoft Internet Information Server (IIS) versions 4.0 and 5.0 are vulnerable to a buffer overflow in the ssinc.dll code that processes Server Side Include (SSI) directives. By loading a file to the Web server that contains a specially-crafted SSI directive, an attacker can overflow a buffer, that is limited to 2550 bytes, once the user requests the vulnerable file. An attacker can use this vulnerability to execute arbitrary commands on the system to gain local system level privileges.

Intent:
a) Penetrate; Remote: get local system level privilege

Offensive Access Requirements:
Attacker must have write access to the web root of the target web server

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 80; HTTP

Target Hardware:
Windows: Microsoft IIS 4.0; Microsoft IIS 5.0; Windows 2000: All Versions; Windows NT: All Versions

Target Software:
Application; Microsoft IIS (Internet Information Server) Server Side Include (SSI) directive File: *ssinc.dll*

Case: 14

InterScan RegGo.dll buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2001-0678; bugtraq id 2907

Description:
Trend Micro InterScan VirusWall for Windows NT 3.5 prior to version 3.51 Build 1349 and InterScan WebManager version 1.2 are vulnerable to a buffer overflow in the reggo.dll file. This file is used to support a web management console feature in InterScan WebManage. By using a long string containing 820 characters, an attacker can overflow a buffer to execute arbitrary code on the system.

Intent:
a) Penetrate; Local: get privilege

Offensive Access Requirements:
Attacker must have local access

Offensive Platform:
Windows: Windows 2000; Windows NT 4.0; Windows NT 3.5 (for VirusWall).

Delivery Strategy:
a) Local Access

Target Hardware:
Windows: Windows 2000; Windows NT 4.0; Windows NT 3.5 (for VirusWall).

Target Software:
a) Application; InterScan VirusWall 3.51; InterScan WebManager 1.2 File: *reggo.dll*

b) OS; Windows NT 3.5-4.0; Windows 2000

Case: 15

InterScan WebManager HttpSave.dll buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2001-0761; bugtraq id 2959

Description:
Trend Micro InterScan WebManager version 1.2 is an application that inspects http traffic flowing into a network for known malicious code. This application also has the capability to restrict access to adult/ unproductive web sites, manage and monitor web usage, monitor and control http traffic, and provide digital certificate revocation checking in SSL connections. If a secure Web site's digital certificate has been revoked, InterScan WebManager has the capability to terminate the transaction. A remotely exploitable buffer overflow exists in the RegGo dynamic link library module included in Trend Micro InterScan WebManager. This module provides management features for the system administrator over an http interface. By sending a long argument to a particular configuration parameter in the HttpSave.dll file, a remote attacker can overflow a buffer and execute arbitrary code on the system with system privileges.

Intent:
a) Penetrate; Remote: get privilege; get info

Offensive Access Requirements:
Attacker must have account access (?)

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 80; HTTP

Target Hardware:
Windows: Windows 2000; Windows NT 4.0

Target Software:
a) Application; InterScan WebManager 1.2 File: *HttpSave.dll*

b) OS; Windows NT 4.0; Windows 2000

Case: 16

Windows Media Player .ASF marker buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2001-0719; bugtraq id 3156

Description:
Microsoft Windows Media Player is a multimedia player for many formats of music and video files. Windows Media Player versions 6.4, 7.0, 7.1 ans XP are vulnerable to a buffer overflow in the processing of .ASF video files. By sending a specially-crafted .ASF file containing an overly long marker, a remote attacker can overflow a buffer and crash the application or execute arbitrary code on the user's computer.

Intent:
a) Penetrate; Remote get privilege; get info

b)DOS; Remote crash/freeze app

Offensive Access Requirements:
Host must open file

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 80; .asf file

Target Hardware:
Windows: All

Target Software:
a) Application; Windows Media Player 6.4; Windows Media Player 7.0; Windows Media Player 7.1; Microsoft Windows Media Player XP

b) OS; Windows All

Case: 17

Microsoft Media Player .ASX buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2000-1113; CAN-2001-0242; bugtraq id 1980, 2677, 2686

Description:
Microsoft Windows Media Player is a multimedia player for many formats of music and video files. Versions 6.4 and 7.0 are vulnerable to a buffer overflow in the code that parses Active Stream Redirector (.ASX) files. The ASX enables a user to play streaming media residing on an intranet or external site. .ASX files are metafiles that redirect streaming media content from a browser to Windows Media Player. The contents of ASX files, when being interpreted by Windows Media Player, are copied into memory buffers for run-time use. When this data is copied, it is not ensured that the amount of data copied is within the predefined size limits. As a result, any extraneous data will be copied over memory boundaries and can overwrite neighbouring memory on the program's stack. Depending on the data that is copied, a denial of service attack could be launched or arbitrary code could be executed on the target host. Windows Media Player runs in the security context of the user currently logged on, therefore arbitrary code would be run at the privilege level of that particular user. If random data were entered into the buffer, the application would crash and restarting the application is required in order to regain normal functionality. If a user was misled to download a hostile .ASX file to the local machine, they would only have to single click on the file within Windows Explorer to activate the code. This is due to the 'Web View' option that is used by Windows Explorer to preview web documents automatically while browsing (this feature is enabled by default). In addition, a malformed .ASX file could be embedded into a HTML document and be configured to execute when opened via a browser or HTML compliant email client.

Intent:
a) Penetrate; Remote get privilege; get info

b)DOS; Remote crash/freeze app

Offensive Access Requirements:
Host must open file

Offensive Platform:
Any

Delivery Strategy:
a) a) IP; TCP; port 80; .asx file

Target Hardware:
Windows: All

Target Software:
a) Application; Windows Media Player 6.4; Windows Media Player 7.0

b) OS; Windows All

Case: 18

Netscape Directory Server RCPT TO excessive quotes buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CAN-2001-0164

Description:
Netscape Directory Server versions 4.1 and 4.12 are vulnerable to a buffer overflow. By default, Netscape Directory Server is installed as part of Netscape Messaging Server version 4.15SP3. A remote attacker can connect to the SMTP service and insert into the "RCPT TO" field a specially-crafted name containing excessive quote (Hex 0x22) characters to overflow a buffer and execute arbitrary code on the server or cause a denial of service attack.

Intent:
a) Penetrate; Remote get privilege; get info

b)DOS; Remote crash/freeze app

Offensive Access Requirements:
Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 25; SMTP

Target Hardware:
Windows NT: All Versions

Target Software:
a) Application; Netscape Directory Server 4.1; Netscape Directory Server 4.12 File: *libslapd.dll*

b) OS; Windows NT

Case: 19

CASSANDRA NNTP server buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CVE-2000-0341; bugtraq id 1156

Description:
The Cassandra NNTP v1.10 server by Atrium Software is vulnerable to denial of service attack caused by a buffer overflow. Cassandra NNTP is a Windows-based newsgroup server that can be accessed and configured by a remote user. A remote attacker can telnet to port 119 and overflow the login buffer by entering a long username containing 10,000 characters or more.

Intent:
a)DOS; Remote crash/freeze app/server

Offensive Access Requirements:
Network Access

Offensive Platform:
Any

Delivery Strategy:
a) IP; TCP; port 119; NNTP

Target Hardware:
Windows: Windows 95: All Versions; Windows 98: All Versions; Windows NT: All Versions

Target Software:
a) Application; CASSANDRA NNTPServer 1.10

b) OS; Windows 95: All Versions; Windows 98: All Versions; Windows NT: All Versions

Case: 20

Windows NT RAS client contains an exploitable buffer overflow

Source:
Common Vulnerabilities and Exposures (CVE); http://cve.mitre.org/; CVE-1999-0715; bugtraq id 1156

Description:
The portion of the Remote Access Service (RAS) client for Windows NT 4.0 that processes phone book entries is vulnerable to a denial of service attack caused by a buffer overflow. With the RAS service comes RASSRV.EXE, which implements the Remote Access Server service and is used for accepting incoming calls, RASMAN.EXE which implements the RAS Autodial Manager and RAS Connection Manager services which are used to dial out. RASPHONE.EXE is the application used when a user manual dials out, as well as editing the Phone Book. RASDIAL.EXE is also used to dial out. RASSRV.EXE and RASMAN.EXE are system processes and run in the security context of the system where as RASPHONE.EXE and RASDIAL.EXE normally run in the security context of the user who starts the process. The buffer overruns occur because the RAS API functions, such as RasGetDialParams( ), perform no bounds checking and fill structures that contain character arrays. For instance, when the Autodial Manager dials out it uses the RasDailGetParams ( ) function to read in such things as the telephone number from the Phonebook, rasphone.pbk. It places these into the RASDIALPARAMS structure that contains characters arrays. Because no bounds checking is performed if the rasphone.pbk contains an overly long telephone number it will cause RASMAN.EXE to access violate. If the phone number is over 299 characters in length we overwrite the processor's EIP and can completely change the programs order of execution and execute arbitary code, though more on this later. By default rasphone.pbk gives Everybody the Change NTFS permission meaning that anyone with access to this file may edit its contents and cause the buffer overflow.

Intent:
a)Penetrate; Local; get privilege

Offensive Access Requirements:
Local Access

Offensive Platform:
Windows NT: all versions; Windows 2000: all versions

Delivery Strategy:
Local Access

Target Hardware:
Windows: Windows NT: all versions; Windows 2000: all versions

Target Software:
a) OS; Windows NT: all Versions; Windows 2000: all versions; Remote Access Service (RAS) File: *rasfil32.dll* earlier than April 28th 1999

Appendix D

IDC language

IDC: variables

All variables in IDC are automatic local variables (sic!). A variable can contain: - a 32-bit signed long integer - a character string (max 255 characters long) - a floating point number (extra precision, up to 25 decimal digits)

A variable is declared in this way:
auto var;

This declaration introduces a variable named 'var'. It can contain a string or a number. All C and C++ keywords are reserved and cannot be used as a variable name. The variable is defined up to the end of the function.

NOTE: to emulate global scope variables you may use array functions and create global persistent arrays.

IDC: Functions

A function in IDC returns a value. There are 2 kinds of functions:

- built-in functions
- user-defined functions

A user-defined function is declared in this way:

```
static func(arg1,arg2,arg3)
 {
   statements ...
 }
```

where arg1,arg2,arg3 are the function parameters,'func' is the function name. It is not nesessary to specify the types of the parameters because any variable can contain a string or a number. All necessary type conversions are handled automatically.

IDC: Statements

In IDC there are the following statements:

```
expression;       (expression-statement)
if (expression) statement
if (expression) statement else statement
for ( expr1; expr2; expr3 ) statement
while (expression) statement
do statement while (expression);
break;
continue;
return <expr>;
return;           the same as 'return 0;'
{ statements... }
;                 (empty statement)
```

IDC: Expressions

In the IDC expressions you can use almost all C operations except:

complex assigment operations as '+='
, (comma operation)

You can use the following construct in the expressions:

[s, o]

This means to calculate linear (effective) address for segment 's' offset 'o'. The calculation is made using the following formula:

(s << 4) + o

If a string constant is specified as 's', it denotes a segment by its name.

There are 3 type conversion operations:

long(expr) float number is truncated during conversion
char(expr)
float(expr)

However, all type conversions are made automatically:

- addition:
  if both operands are strings,
    string addition is performed (strings are concatenated);
  if floating point operand exists,
    both operands are converted to floats;
  otherwise
    both operands are converted to longs;
- subtraction/multiplication/division:
  if floating point operand exists,
    both operands are converted to floats;
  otherwise
    both operands are converted to longs;
- comparisions (==,!=, etc):
  if both operands are strings, string comparision is performed;
  if floating point operand exists,
    both operands are converted to floats;
  otherwise
    both operands are converted to longs;
- all other operations:
  operand(s) are converted to longs;

List of IDC functions used in algorithm

Asks

```
char    AskStr          (char defval,char prompt); // ask a string
char    AskFile         (long forsave,char mask,char prompt);   // ask a file name
long    AskAddr         (long defval,char prompt); // BADADDR - no or bad input
long    AskLong         (long defval,char prompt); // -1 - no or bad input
long    AskSeg          (long defval,char prompt); // BADSEL - no or bad input
char    AskIdent        (char defval,char prompt);
long    AskYN           (long defval,char prompt); // -1:cancel,0-no,1-ok
void    Message         (char format,...); // show a message in msg window
void    Warning         (char format,...); // show a warning a dialog box
void    Fatal           (char format,...); // exit IDA immediately
```

Byte

```
// Get value of program byte
//      ea - linear address
// returns: value of byte. If byte has not a value then returns 0xFF

long    Byte            (long ea);              // get a byte at ea
```

form

```
// Return a formatted string.
//      format - printf-style format string.
//               %a - means address expression.
//               floating point values are output only in one format
//               regardless of the character specified (f,e,g,E,G)
//               %p is not supported.
// The resulting string must be less than 255 characters

char    form            (char format,...);      // works as sprintf
                                                // The resulting string should
                                                // be less than 255 characters.
```

GetMnem

```
 // Get mnemonics of instruction
//      ea - linear address of instruction
// returns: "" - no instruction at the specified location
// note: this function may not return exactly the same mnemonics
// as you see on the screen.

char    GetMnem         (long ea);              // get instruction name
```

GetOpnd

```
// Get operand of an instruction
//      ea - linear address of instruction
//      n  - number of operand:
//                   0 - the first operand
//                   1 - the second operand
// returns: the current text representation of operand
char    GetOpnd         (long ea,long n);          // get instruction operand
                                                        // n=0 - first operand
```

GetOpType

```
// Get type of instruction operand
//      ea - linear address of instruction
//      n  - number of operand:
//                0 - the first operand
//                1 - the second operand
// returns:
//      -1      bad operand number passed
//      0       None
//      1       General Register (al,ax,es,ds...)
//      2       Memory Reference
//      3       Base + Index
//      4       Base + Index + Displacement
//      5       Immediate
//      6       Immediate Far Address
//      7       Immediate Near Address
//      8       FPP register
//      9       386 control register
//      10      386 debug register
//      11      386 trace register
//      12      Condition (for Z80)
//      13      bit (8051)
//      14      bitnot (8051)

long    GetOpType       (long ea,long n);       // get operand type
```

LocByName

```
// Get linear address of a name
//      from - the referring address.
//             Allows to retrieve local label addresses in functions.
//             If a local name is not found, then address of a global name is returned.
//      name - name of program byte
// returns: address of the name
//       BADADDR - no such name

long    LocByName       (char name);
long    LocByNameEx     (long from, char name);
```

strlen

```
// Return length of a string in bytes
//      str - input string
// Returns: length (0..n)

long    strlen          (char str);             // calculate length
```

strstr

```
// Search a substring in a string
//      str    - input string
//      substr - substring to search
// returns: 0..n - index in the 'str' where the substring starts
//         -1   - if the substring is not found

long    strstr          (char str,char substr); // find a substring, -1 - not found
```

substr

```
// Return substring of a string
//      str - input string
//      x1  - starting index (0..n)
//      x2  - ending index. If x2 == -1, then return substring
//            from x1 to the end of string.

char    substr          (char str,long x1,long x2); // substring [x1..x2-1]
                                                    // if x2 == -1, then till end of line
```

Xrefs

```
//      Flow types:
#define fl_CF   16              // Call Far
#define fl_CN   17              // Call Near
#define fl_JF   18              // Jump Far
#define fl_JN   19              // Jump Near
#define fl_US   20              // User specified
#define fl_F    21              // Ordinary flow
                                        // Mark exec flow 'from' 'to'
void    AddCodeXref(long From,long To,long flowtype);
long    DelCodeXref(long From,long To,int undef);// Unmark exec flow 'from' 'to'
                                       // undef - make 'To' undefined if no
                                       //       more references to it
                                       // returns 1 - planned to be
                                       // made undefined

// The following functions include the ordinary flows:
long    Rfirst  (long From);                    // Get first xref from 'From'
long    Rnext   (long From,long current); // Get next xref from
long    RfirstB (long To);                      // Get first xref to 'To'
long    RnextB  (long To,long current);   // Get next xref to 'To'

// The following functions don't take into account the ordinary flows:
long    Rfirst0 (long From);
long    Rnext0  (long From,long current);
long    RfirstB0(long To);
long    RnextB0 (long To,long current);

//      Data reference types:
#define dr_O    1               // Offset
#define dr_W    2               // Write
#define dr_R    3               // Read
#define dr_T    4               // Text (names in manual operands)

void    add_dref(long From,long To,long drefType);      // Create Data Ref
void    del_dref(long From,long To);    // Unmark Data Ref

long    Dfirst  (long From);            // Get first refered address
long    Dnext   (long From,long current);
long    DfirstB (long To);              // Get first referee address
long    DnextB  (long To,long current);

long    XrefType(void);                 // returns type of the last xref
                                        // obtained by [RD]first/next[B0]
                                        // functions. Return values
                                        // are fl_... or dr_...

// set number of displayed xrefs
#define XrefShow(x)             SetCharPrm(INF_XREFNUM,x)
```

xtol

```
// Convert ascii string to a binary number.
// (this function is the same as hexadecimal 'strtol' from C library)
long    xtol            (char str);             // ascii hex -> number
                                                  // (use long() for atol)
```

Appendix E

sprintf_scan.idc

```
static main()
{
     auto      SprintfAddr, reference;
     SprintfAddr = AskAddr(-1, "Enter address:");
     reference = (SprintfAddr);

     while(reference != -1)
     {
          if(GetMnem(reference) == "call")
               GetAnalysis(reference);
          reference = Rnext(SprintfAddr, reference);
     }

     reference = DfirstB(SprintfAddr);

     while(reference != -1)
     {
          if(GetMnem(reference) == "call")
               GetAnalysis(reference);
          reference = DnextB(SprintfAddr, reference);
     }
}


static  GetAnalysis(push)
{
     auto      literalString, literalStrAddr, targetBuffer;
     targetBuffer = GetReturnValue(push, 1);
     literalString = GetReturnValue(push, 2);

     if(strstr(literalString, "offset") != -1)
          literalString = substr(literalString, 7, -1);

     literalStrAddr = LocByName(literalString);
     literalString = GetString(literalStrAddr);

     if(strstr(literalString, "%s") != -1)
          if(strstr(targetBuffer, "var_") != -1)
              Message("\n%lx --> POTENTIAL OVERFLOW? Target Buffer is: " + targetBuffer + "   String
                        Literal is: \"%s\"\n\n\n",  push, literalString);

}


static GetString(ourString)
{
     auto  temporaryString, character;
     temporaryString = "";
     character = Byte(ourString);

     while((character != 0)&&(character != 0xFF))
     {
          temporaryString = form("%s%c", temporaryString, character);
          ourString = ourString + 1;
          character = Byte(ourString);
     }
     return(temporaryString);
}


static  GetReturnValue(push, n)
{
     auto      temporaryRegister;
     while(n > 0)
     {
          push = RfirstB(push);
          if(GetMnem(push) == "push")
               n = n-1;
     }

     if(GetOpType(push, 0) == 1)
     {
          temporaryRegister = GetOpnd(push, 0);
          push = RfirstB(push);

          while(GetOpnd(push, 0) != temporaryRegister)
               push = RfirstB(push);
          return(GetOpnd(push, 1));
     }
     else return(GetOpnd(push, 0));
}
```

Appendix F

sprintf_crasher.c

```
////////////////////////////////////////////////////////////////////////////////
//                            ** sprintf() Crasher **                         //
//                                                                            //
//  THIS PROGRAM CREATES A SET OF BUFFERS ACCEPTING INPUT IN TWO CONTEXTS. WHEN THE    //
//  sprintf() COMMAND IS INVOKED IT ATTEMPTS TO COPY THE CONTENTS FROM A LARGE BUFFER  //
//  INTO A SMALLER BUFFER. THIS CAUSES A BUFFER OVERFLOW WHICH ALLOWS THE OVERWRITE OF //
//  FRAME POINTER.                                                            //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
struct Buffer // buffer structure
       {
       char bufLarge[100];
       char bufSmall[60];
       };
void Print(struct Buffer p)  // function to print contents of buffer structure
       {
       printf("            ***** LOAD LARGE BUFFER WITH CHAR STRING *****\n %s ", p.bufLarge);
       printf("           ***** LOAD SMALL BUFFER WITH CHAR STRING *****\n %s", p.bufSmall);
}

main()
{

    char bufGlobal[50]; // create third buffer instance two run in a second context
    char newBuf[25];
    struct Buffer test= {"                    !!!  Large BUFFER Up  !!!\n\n",
                         "                    !!!  Small BUFFER Up  !!!\n\n"};

       struct Buffer x; // second instance of struct Buffer
       Print(test);  // prints contents of bufLarge and bufSmall using print function

    printf("|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n");
    printf("| A large string to be truncated to (3) chars by safe sprintf() call that |\n");
    printf("|     copies bufLarge into bufSmall. Enter large string into bufLarge:    |\n");
    printf("|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n\n");

       scanf("%s", x.bufLarge);
    if (strlen(x.bufLarge)>99)
       {
         printf("String length is greater than [100] character x.bufLarge");
         exit(1);
       }
    else
       {
        sprintf(x.bufSmall,"%.3s", x.bufLarge); //safe sprintf() #1; only reads first (3) chars
        printf("\nSAFE SPRINTF() #1: A maximum of (3) characters in bufSmall: %s\n", x.bufSmall);
        sprintf(newBuf,"AAA"); //safe sprintf() #2; only (3) chars
       }

    printf("SAFE SPRINTF() #2:Only (3) A's in newBuf: %s\n", newBuf);
    printf("\n\n\n\n\n\n\n");
    printf("|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n");
    printf("|||  THIS MIMICS A SERVER SIDE PROGRAM WHICH ACCCEPTS CLIENT INPUT IN   |||\n");
    printf("|||                          THE FORM OF A URL                         |||\n");
    printf("|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n\n");
    printf("Enter Internet URL: ");

       scanf("%s", x.bufLarge);
    if (strlen(x.bufLarge)>99)
       {
         printf("String length is greater than [100] character size of x.bufLarge");
         exit(1);
       }
    else
       {
       sprintf(bufGlobal,"\nCan't open the following URL for reading? %s",x.bufLarge); // copy string into bufGlobal
        printf("%s\n", bufGlobal); // local print contents of bufGlobal
        printf("\n|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n\n");
       }
  return 0;
}
```