Securing Software: An Evaluation of Static Source Code Analyzers

by

Misha Zitser

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

August 29, 2003

[September 2003]

Author_____
Department of Electrical Engineering and Computer Science
August 29, 2003

Certified
by_____
Richard Lippmann
Thesis Supervisor

Accepted
by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Securing Software: An Evaluation of Static Source Code Analyzers
by
Misha Zitser

Submitted to the
Department of Electrical Engineering and Computer Science

August 29, 2003

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis evaluated five static analysis tools - Polyspace C Verifier, ARCHER, BOON, Splint, and UNO - using 14 code examples that illustrated actual buffer overflow vulnerabilities found in various versions of Sendmail, BIND, and WU-FTPD. Each code example included a "BAD" case with one or more buffer overflow vulnerabilities and a "PATCHED" case without buffer overflows. The buffer overflows varied and included stack, heap, bss and data buffers; access above and below buffer bounds; access using pointers, indices, and functions; and scope differences between buffer creation and use. Detection rates for the "BAD" examples were low except for Splint and Polyspace C Verifier, which had average detection rates of 57% and 87% respectively. However, average false alarm rates, as measured using the "PATCHED" programs, were high for these two systems. The frequency of false alarms per lines of code was high for both of these tools; Splint gave on average one false alarm per 50 lines of code, and PolySpace gave on average one false alarm per 10 lines of code. This result shows that current approaches can detect buffer overflows, but that false alarm rates need to be lowered substantially.

Thesis Supervisor: Richard Lippmann
Title: Senior Scientist, MIT Lincoln Laboratory

Acknowledgements

I would like thank my advisor, Rich Lippmann, for his guidance and suggestions during the whole process. I am also very grateful for the time he spent in proofreading the thesis and in offering constructive criticism. I would also like to thank all the people at Lincoln Lab whom I've had many interesting discussions with on the subject of static code analysis, including Robert Cunnigham, Tim Leek, Roger Khazan, Kendra Kratkiewicz, and Jesse Rabek. I specifically would like to thank Rob and Tim - Rob for suggesting that I look at the off-by-one vulnerability in wu-ftpd, and Tim for all his help in setting up the tools, for his help in writing the small test cases and his help in analyzing the results. In addition, I would like to thank David Evans for his help with Splint, David Wagner for answering questions about BOON, Michael Howard for pointing out the availability of PREfast, Yichen Xie and Dawson Engler for their help with ARCHER, and Chris Hote and Vince Hopson for all their help on answering questions about PolySpace. I would also like to thank Doug Stetson for being a great office-mate, and fellow students, Dave Messing and Nick Malyska, for being cheerful neighbors. Last but not least, I would like to thank my parents and my sister, Katrina, for providing me with plenty of moral support. I sincerely apologize to anyone whom I might have forgotten to thank.

Table of Contents

Chapter 1 Introduction

1.0 Why is software security important?

Most of us rely on software everyday, whether knowingly or not, for such things as withdrawing money from an ATM, sending e-mail, word processing, surfing the web, doing our taxes, balancing our checkbook and sending out e-greetings. We have grown to trust computers and the software that runs on them. We expect software to behave as specified, to be secure and to be reliable. But what if the e-mail program we use, for instance Microsoft Outlook, had an exploitable security hole in it? Or what if the web browsers we use, such as Netscape Navigator or Internet Explorer, had security holes? We wouldn't feel as safe anymore.

Unfortunately, these are not mere hypothetical scenarios! Many serious security holes have previously been discovered in Microsoft Outlook, Netscape Navigator, Internet Explorer and many other popular applications. Fortunately, if such bugs are discovered by software developers or software auditors, patches, or upgrades, are released relatively quickly and the software can be protected. If a security hole in software gets discovered by a malicious hacker, then the situation is far more serious. Computer crime is a serious problem in today's society. According to the "2003 Computer Crime and Security Survey" conducted by the Computer Security Institute and the San Francisco Federal Bureau of Investigation's (FBI) Computer Intrusion Squad, 251 organizations reported a total of almost $202 million in financial losses due to cyber attacks [20]. Cyber attacks can range from computer viruses or worms infecting company networks and disrupting productivity, to hackers exploiting specific security holes in software to break into machines. Cyber attacks pose a serious threat to our nation's critical infrastructure as a whole. According to Richard Clarke, former special advisor to the president on cybersecurity, the number of software vulnerabilities is "at an all-time high", and the time between the discovery of a bug and the creation of an exploit code is diminishing. Also, as Ronn Bailey, CEO of Vanguard Integrity Professionals, provider of enterprise security software, has pointed out, "We are vulnerable to a cyberattack of the 9/11 category or greater, and that could happen at any moment" [28].

1.1 Types of software bugs

The most common vulnerabilities in software can be subdivided into several classes, not all of them necessarily represent security vulnerabilities. Bugs in software include things such as buffer overflows, race conditions, format string bugs, integer overflows, integer underflows, memory leaks, dangling pointers, array indexing errors, divide by zero bugs, truncation errors, failure to drop privileges, bad type casts etc. These vulnerabilities are briefly described in Appendices A and B. Depending on the context in which the bug occurs in the software, it can cause varying degrees of damage. Many of these bugs can be catastrophic. For instance, a "divide by zero error" in a program might cause a rocket to crash or cause a pacemaker to stop working, resulting in cardiac arrest! Many of these bugs are often subtle and get triggered only under very

specific circumstances. The conditions under which a bug gets triggered might be so rare that the bug might be dormant in the program code for a long time without being noticed. You might recall the "Y2K Bug" that caused mass hysteria. This was a widespread problem affecting many software applications, and it did not gain software developers' attention for many decades. Many programmers who wrote code when computer memory was very expensive, chose to save memory by representing dates compactly, for instance, December 31st, 1999 was represented as "12/31/99". When the clock would strike midnight on December 31st, 1999, many program timers would simply wrap around, and January 1st, 2000 would get represented as "01/01/00". The results of this would be catastrophic. Computers would think that the year 1900 had rolled around! Just imagine the possible side effects of this... For example, the software on your bank's computer might all of a sudden think that your loans were 99 years overdue! [91] Many people believed that the side effects of the Y2K bug would be disastrous and would shake the world's economies and infrastructure. Fortunately, after putting in thousands of human hours, programmers were able to fix these problems in most affected software, without too many repercussions.

Certain kinds of software bugs represent security vulnerabilities that can be exploited by attackers to cause damage. For instance, by feeding a particular input to a program that triggers a certain bug to manifest itself, an attacker might cause the program to crash or even worse, be able to gain total control of the computer on which the program is running. Today, with the omnipresence of computer networks, i.e. the Internet, an attacker might not even have to be present at your computer to cause damage; your machine can be damaged remotely, from his own computer, somewhere halfway across the world. Although all of the bugs listed above are important and should be eliminated from software, this thesis will focus on only one type of bug, namely buffer overflows.

1.2 Why focus on buffer overflows?

Buffer overflows are one of the most discussed software security vulnerabilities, and they have become a frequently occurring problem in the past few years. The first widely publicized buffer overflow exploit appeared in 1988 as part of Robert T. Morris' infamous Internet Worm which brought down a considerable portion of the internet [75]. Ever since, buffer overflow exploits have become a popular choice among attackers. For instance, in 2002, buffer overflow vulnerabilities accounted for approximately 20% of all security vulnerabilities listed in ICAT's computer vulnerability database [45]. About two thirds of these buffer overflow vulnerabilities were given the highest level of severity and were remotely exploitable. In 2003, about 30% of all vulnerabilities in ICAT's database involved buffer overflows. About half of these buffer overflows were ranked at the highest severity level and were remotely exploitable. To cite another source, in 2002, about 64% of Carnegie Mellon CERT Coordination Center's security advisories involved buffer overflows (See Figure 1). This statistic jumped up to 75% for the first half of 2003 [16]. CERT, unlike ICAT, reports only the most serious of security vulnerabilities each year. The numbers just cited indicate that buffer overflows are by far the most common

and dangerous threat to software security. Sadly, the fraction of vulnerabilities related to buffer overflows does not seem to be falling.

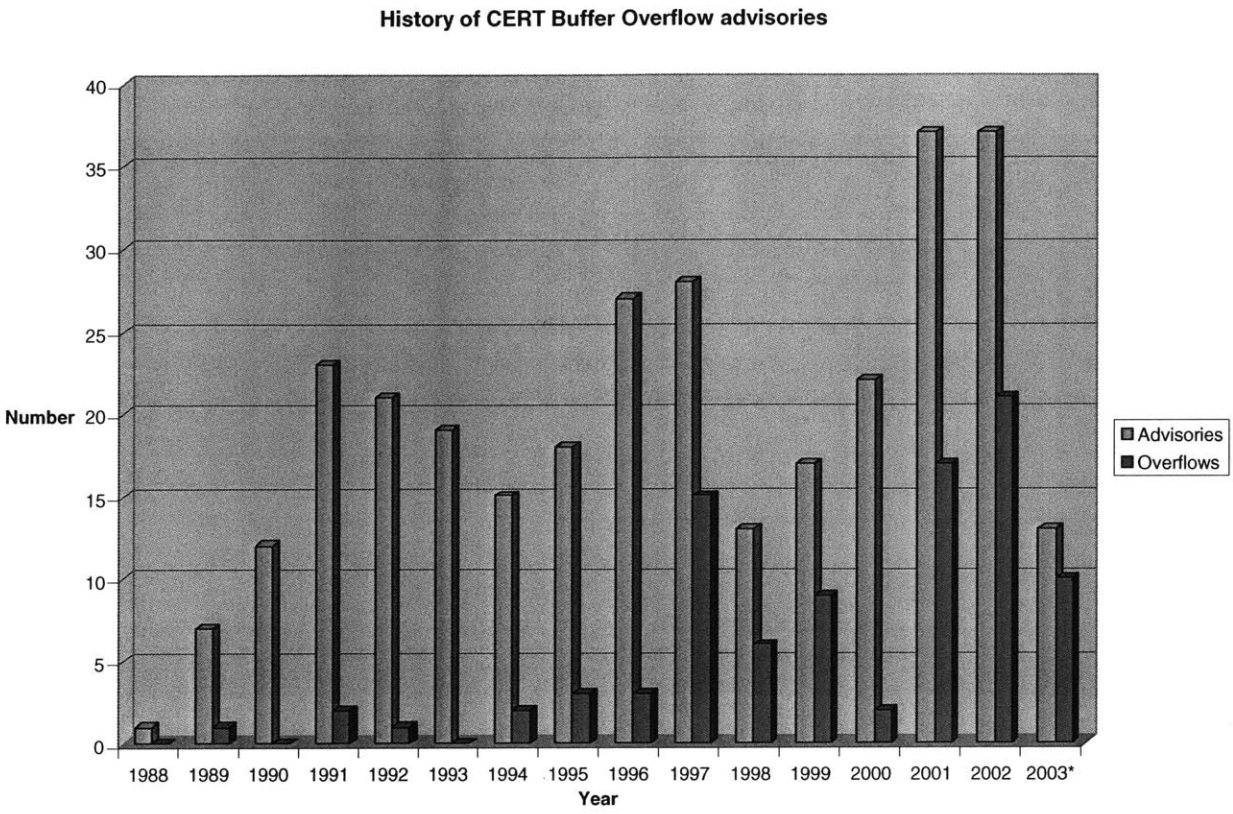

Figure 1. *Data for 2003 – January to July
Data for 1988 to 1998 was borrowed from [60].

1.3 A basic overview of buffer overflows

The basic notion of a buffer overflow is quite simple. A buffer is simply a contiguous block of memory where a program stores data (e.g., an array of characters). A buffer overflow occurs when data is written to a buffer and some of it gets written outside of the buffer bounds. This means that data may get written past the upper bound of the buffer or below the lower bound (buffer overrun or buffer underrun, respectively). As a result of the buffer overflow, memory locations adjacent to the buffer get overwritten. By supplying the data that gets written to the buffer and by controlling which memory locations get overwritten, an attacker may be able to change the execution flow of a program. In the worst case scenario, an attacker is able to supply arbitrary attack code that gets executed as a result of the changed execution flow. The attack code

in most cases simply starts an interactive program called a shell. The details of how the flow of execution can be changed are discussed in the next chapter.

Many applications run as suid (set uid) root. Two common suid root programs in UNIX are lpr and xterm. Such programs are able to receive special privileges from the system upon request, even if the user running the program is just an ordinary user with no special privileges. Gaining root privileges on a machine, or to use the hacker term "owning a machine", means gaining unlimited power over the machine. If a buffer overflow is exploited while a suid root program is running with root privileges, an attacker can spawn a root shell. Once a root shell has been launched, the attacker has the power of a system administrator. With this newly acquired power, the attacker can view private data such as users' passwords, read, delete or modify sensitive files, set up a monitoring station, install back doors (with a root kit), edit logs to hide tracks, masquerade as someone else etc. It should be noted that even buffer overflows in programs that are not suid root are dangerous. Very often a user of a particular application might not have any privileges at all. Gaining a shell would give the user the privileges that the vulnerable program is running with, which would be better than having no privileges at all. An attacker with a shell with limited privileges can try to escalate his/her privileges to root by exploiting another process. Such stepping-stone attacks are quite common with network servers [59]. Buffer overflow attacks will be discussed in greater detail in the next chapter.

1.4 Why are buffer overflows so common?

Most serious buffer overflow exploits occur in programs written in the unsafe languages of C or C++. Other languages, such as Java, ML, Lisp or Python, are not prone to buffer overflows because they provide automatic bounds checking of buffer and pointer accesses during run-time. The drawback of these languages is that they are considerably slower than C and C++. Thus, C and C++ are still the languages of choice for many speed-critical applications. Many of today's commonly used operating systems, such as Linux and OpenBSD, are written in C. As a result, there has been a major effort to prevent buffer overflow vulnerabilities in kernel code [5]. One of the most popular web servers, Apache, is written in C, as are DNS servers (e.g. BIND), and mail transfer agents (MTAs) such as Sendmail and QMAIL. A large amount of other legacy code written in C exists. Much of it is open-source and is security-sensitive. It would be too impractical to rewrite it all in a safer language. The fact that code is open-source means that anyone can inspect it for potential vulnerabilities. Thus, finding security vulnerabilities in open-source, legacy source code before an attacker can find them is an urgent problem.

As stated above, one of the main reasons for the widespread appearance of buffer overflows in C/C++ code is the fact that these languages are inherently not secure. These languages were designed to be fast, powerful, and convenient, but security was not the primary consideration. For instance, C and C++, unlike most other high level languages, allow the programmer to directly manipulate memory contents via pointers. Also, allocation and freeing of memory is left up to the programmer. By contrast, Java does

not allow programmers to manipulate memory at a low level, and garbage collection is done automatically. While having the ability to manipulate memory directly is a very powerful feature and can lead to highly optimized code, this feature is a major source of buffer overflows. Writing secure C/C++ code requires the programmer to be highly aware of security issues such as buffer overflows, race conditions, format bugs etc. This awareness includes a good understanding of how these problems arise and how to avoid them. Unfortunately, not all programmers have been trained to program with security in mind. Very often security gets "thrown in" at the end of the software development process only as an afterthought, rather than being incorporated into the program from the very beginning. This approach is very likely to lead to many overlooked security vulnerabilities. Let's look at some sources of programmer errors that lead to buffer overflows.

C offers a wide spectrum of string manipulation functions that are quite powerful and convenient. Unfortunately, many of these functions can be easily misused and can lead to buffer overflows. Just to illustrate the concept... Strcpy(buffA, buffB) copies the contents of buffB into buffA, but it does not check to see if buffA is large enough to store all of buffB. If the current length of buffB is greater than the allocated size of buffA, a buffer overflow occurs. Thus, strcpy() can only be used safely if the programmer can guarantee that the output buffer will be large enough to store the input. Ideally, the programmer would include some sort of a size check before invoking the strcpy() routine. Similarly, strcat(buffA, buffB) concatenates the contents of buffB to the contents of buffA, storing the result in buffA, without checking to see if buffA is capable of storing the result of the concatenation. Again, the programmer should check to make sure that enough space is left in the output buffer before calling strcat(). If a character buffer comes from an untrusted source and then gets used as an input to one of these unsafe functions, a buffer overflow is likely to occur if the input is not sanitized properly. The list of potentially hazardous C functions is quite long. Some more examples are sprintf(), gets(), memcpy(), wcsncat(), lstrcat(), wcsncpy() [44, p.714]. Many of these unsafe functions have safer variants, for instance strncpy() and strncat(), but even these functions can be misused. Many of these functions follow very dissimilar rules of usage, and it is easy to get the order of arguments mixed up, or to forget an argument, or to misuse an argument. For example, one important difference between strncat() and strncpy() is that strncat() always null terminates the destination buffer. Strncat() and strncpy(), although considered the safe variants of strcpy() and strcat(), encourage off-by-one bugs (see Appendix B.3). In order to use these function safely, the programmer has to be extremely careful.

One source of information about the usage of different C functions is the UNIX man pages. The man pages often give warnings about the potential hazards of unsafe functions, such as the ones shown below for the functions cuserid() and gets():

"Nobody knows precisely what cuserid() does - avoid it in portable programs - avoid it altogether - use getpwuid(geteuid()) instead, if that is what you meant. DO NOT USE cuserid()."

"Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead."

Despite such warnings, one can still find real program code out there that uses these functions. Gets() is one of the deadliest string functions and is a likely source of buffer overflows!

Many string functions should be disallowed altogether because it is practically impossible to use them safely. Many functions are deprecated, but nevertheless programmers continue to use them. Examples of deprecated string functions include bcmp(), bcopy(), and bzero(). In addition, some string functions are not portable to many platforms, and as a result programmers end up using unsafe variants of these functions to accomplish what they want.

1.5 What's being done to stop buffer overflows?

A significant amount of time and energy has been devoted to finding buffer overflows in software. Most of the efforts up until very recently have involved manual code audits. For instance, in September of 1996, an extensive manual security audit of Sendmail was performed by security experts. Four months later, a buffer overflow was discovered that had been missed by the manual audit. This is not surprising, for inspecting approximately 50 thousand lines of code is a difficult and painstaking task even for experts. In 1998, Reliable Software Technologies found three buffer overflows in WU-FTPD version 2.4 using a dynamic analysis technique known as fault injection [37]. Security experts did not think these buffer overflows were exploitable. In other words, they deemed it very unlikely that a user input could find its way to one of these buffers, cause a buffer overflow and lead to a security violation. Approximately one year later, a CERT advisory appeared showing that one of the three detected buffer overflows was an exploitable vulnerability after all [59]. This is just another example illustrating that manual security audits are not always perfect. Not only that, this example shows that even when a potential buffer overflow is discovered, determining whether or not it is exploitable can be a difficult task (See section 2.6).

Recently, Bill Gates of Microsoft started a strong push towards making Microsoft's software more secure. Microsoft has begun working on developing a highly reliable and secure platform. This initiative is known as Trustworthy Computing. In an email that Bill Gates sent to Microsoft's programmers on January 15th, 2002, he defined Trustworthy Computing as "computing that is as available, reliable and secure as electricity, water services and telephony". In another excerpt from the email, Gates stated clearly the importance of software security:

"In the past, we've made our software and services more compelling for users by adding new features and functionality, and by making our platform richly extensible. We've done a terrific job at that, but all those great features won't matter unless

customers trust our software. So now, when we face a choice between adding features and resolving security issues, we need to choose security. Our products should emphasize security right out of the box, and we must constantly refine and improve that security as threats evolve" [39].

As part of this push towards greater security awareness, Microsoft started an intensive training course to teach its programmers secure programming techniques. Microsoft also published a book on software security, "Writing Secure Code", by Michael Howard and David LeBlanc [44]. Another source of information on writing secure software is a book called "Building Secure Software" by Gary McGraw and John Viega [82]. This book has served as a basis for several secure programming training courses. Microsoft also provides its programmers with code analysis tools such as PREfix[15] and PREfast[71] that help catch security bugs. It would be great if more software companies were to follow Microsoft's example and begin placing stronger emphasis on software security.

In order to eliminate security vulnerabilities such as buffer overflows from software, programmers need good tools. Manual source code auditing is effective only to a certain extent. Fortunately, there exist several tools that automate the detection of security bugs in software. Several approaches have been taken in building these tools. Some tools try to detect buffer overflows dynamically, or during run-time, and require running the programs on different test inputs. Other tools try to detect buffer overflow vulnerabilities in programs via static analysis by parsing the source code and looking for dangerous code. Some researchers have tried to eliminate buffer overflows by implementing safer string function libraries, and others have chosen to approach this problem at the operating system level by developing kernel patches. Yet another approach has been to try to dynamically prevent buffer overflows, not necessarily detect them, by using special compiler-based tools. All of these approaches will be discussed in greater detail in chapter three.

1.6 Goals of this thesis

The goals of this thesis are the following:

1) To try to assess the effectiveness of the state-of-the-art static analysis tools for detecting buffer overflows.

2) To determine the types of buffer overflows that the existing static analysis tools can detect.

The following static analysis tools will be looked at in detail:

PolySpace C Verifier [70]

ARCHER [89]

Splint [34, 35]

BOON [83]

UNO [43]

Chapter 2 Types of buffer overflow attacks

2.0 How do buffer overflows work?

There are four regions in the program's memory where buffer overflows can occur. They are the stack, the heap, the data region, and the bss region. Each of these regions is used for storing specific types of variables. The stack is used to store local, fixed-size buffers, other local variables, function arguments, as well as the return addresses of functions (i.e., the address of the next instruction following the function call), and some other state, including environment variables. The entire C mechanism of function calls relies heavily on the stack. The heap stores dynamically allocated buffers. In C, these buffers are usually allocated using malloc(), calloc(), or realloc() calls. The data region stores initialized global and static variables. The bss region stores uninitialized global and static variables. A buffer overflow exploit usually needs to be tailored to the buffer's location. The figure below shows the typical layout of process memory in UNIX. As can be seen from Figure 2, the stack grows down (i.e. from higher memory addresses to lower), whereas the heap, bss, data and text segment grow up (i.e. from lower to higher memory addresses)

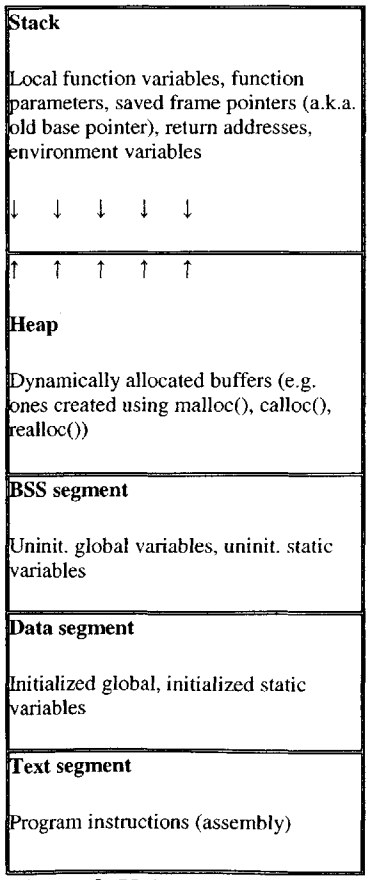

Figure 2. Unix Process Memory Layout.
The top of the stack is at the highest memory address, e.g., 0xffffffff, and the bottom of the text segment is at the lowest address, e.g., 0x00000000.

Here is a simple example program followed by a view of the stack.

```
#include <stdio.h>
void test (int a, int b, char *p) {
int c;
char buf [12];
strcpy(buf, p);
c = a + b;
return c;
}
int main(){
char *p = "PumpkinPie";
test(1,2,p);
printf("PumpkinPie acquired!\n");
return 0;
}
```

Assuming a 32-bit word architecture, the corresponding stack contents right before exiting test() would look something like:

...	...	...	...	**Stack grows this way ↓**
...	...	...	...	
0x00	0x00	0xEF	0xBE	
0x00	0x00	0xEF	0xBE	← pointer p in main()
0	0	0	2	← arg 3 of test()
0	0	0	1	← arg 2 of test()
EIP_3	EIP_2	EIP_1	EIP_0	← arg 1 of test()
EBP_3	EBP_2	EBP_1	EBP_0	← saved return address
0	0	0	3	← saved base pointer
0	0	e	i	← local variable c
P	n	i	k	←buf[]
p	m	u	P	← buf[]
				← Beginning of buf[].

Figure 3.

Here 0xBEEF is a hypothetical location in memory of the string "PumpkinPie" (i.e. the address pointed to by p). This address is located somewhere in the data region of the program's memory space. When the function main() gets called, it gets allocated its own stack frame which stores main's local variables as well as some other state such as environment variables (the main() routine is a special case). The only local variable in

main() is the pointer p = 0xBEEF. During the execution of a program, a register called the "base pointer" keeps track of the local context, i.e. which function we're in at the moment. The base pointer usually points to the first local variable in the current function's stack frame. So, upon entering main(), the base pointer points to the location on the stack containing the pointer p, i.e. 0xBEEF. Then, test() gets called, and a new stack frame gets set up for test(). The first thing that happens is the arguments to test() get "pushed" onto the stack in reverse order, first the pointer p (0xBEEF), then b, then a. Then, the return address, EIP, gets pushed on the stack. EIP is the address of the next instruction that should be executed once the program returns from test(). In this case, it is the address of the printf() instruction. The base pointer register needs to be updated to point to the first local variable of the test() function, but we do not wish to lose track of where the stack frame for main() is located. Thus, the old value of the base pointer register gets saved as a word on the stack, shown as EBP, and the new value of the base pointer gets set to the address of the variable c. As you can see from the above picture, in addition to allocating a word of stack memory for the variable c, three words of memory get set allocated for the twelve-element buffer, buf[]. The picture above reflects the contents of the stack right before returning from test().

Now imagine what would happen if p pointed to a string that was longer than twelve characters, e.g. "The great bald eagle 0xFFEEDDCC". In this case, right before exiting the function test(), the stack would look like:

...	...	...	...	
...	...	...	...	**Stack grows this way ↓**
B	E	E	F	
B	E	E	F	← pointer p in main()
0	0	0	2	← arg 3 of test()
0	0	0	1	← arg 2 of test()
0xCC	0xDD	0xEE	0xFF	← arg 1 of test()
E	l	g	a	← saved return address
E		d	l	← saved base pointer
A	b		t	← local variable c
A	e	r	g	←buf[]
	e	h	T	← buf[]
				← Beginning of buf[].

Figure 4.

As you can see, the variable c, the saved frame pointer (EBP) and the return address (EIP) have been overwritten. This example illustrates the basic concept of a stack buffer overflow. Now, when the function test() begins to return, it will not return to the printf() instruction in main(), but rather, to the address 0xFFEEDDCC, which in all likelihood will cause the program to crash. There are many variants of buffer overflows, some of which we will now describe.

Table 1 below lists some common types of buffer overflow attacks. These attacks differ in the type of information that gets overwritten and the location of the attack code. We describe these attacks below.

Attack Type	What gets overwritten?	Where is the attack code?
Stack-Smashing [3]	Function Return Address	Injected into overflowed buffer on the stack. Injected into a second, attacker-controlled buffer which can be in any of the four regions of memory. An existing program function. Injected into an environment variable
Return-into-libc [87]	Function Return Address	Libc
Off-by-one [66]	Saved Frame Pointer	Usually the stack
Malloc attacks [49]	Can overwrite function pointer in Global Offset Table (unlink() technique) via a heap buffer overflow	Injected into overflowed heap buffer. Injected into a second, attacker-controlled buffer which can be in any of the four regions of memory. Any existing function
Setjmp/longjmp [23]	Longjmp buffer (can be stored on stack, heap, bss or data region)	Injected into overflowed buffer on stack, heap, bss or data region. Injected into a second, attacker-controlled buffer which can be in any of the four regions of memory. Any existing function
Overwrite Function Pointer [23]	Overwrite function pointer stored on stack, heap, bss or data region	Injected into overflowed buffer on stack, heap, bss or data region. Injected into a second, attacker-controlled buffer which can be in any of the four regions of memory. Any existing function
Logic-based [23]	Variables such as filenames, user id's, or other program variables	----------

Table 1: Buffer Overflow Attacks.

2.1 Overwriting Code Pointers

Most buffer overflow attacks seek to corrupt code pointers and thus cause attack code to be executed. Code pointers corrupted by buffer overflows typically are function pointers, longjmp buffers or pointers in the stack activation record (i.e., the return address and the saved frame pointer). Such buffer overflow attacks usually overwrite a code pointer to point to some other code that gets executed later.

The classic buffer overflow attack is known as "stack-smashing" [3]. In a stack-smashing attack, a local buffer, such as a character array, is stored on the stack and gets overflowed, and the function's return address gets overwritten. An attacker can overwrite the return address with the address of the attack code, and thus, when the function returns, it returns to the attack code instead of the normal return point. Usually, the attack code contains instructions to start a shell. If the exploited program is running with root privileges, an attacker can launch a root shell, thus gaining complete access to the victim's machine. An attacker with root privileges can steal user's passwords, read, delete, or modify sensitive files, set up back-doors, or cause other damage to the victim's system.

Instead of overwriting the return address of the function, an exploit can overwrite the saved frame pointer, which is stored right before the return address in the stack activation record. A frame pointer is used to keep track of the stack frames corresponding to the different function calls, and it usually points to the first local variable in the current stack frame. As a setup for an attack, an attacker would construct a fake stack frame somewhere in memory with the return address pointing to the attack code. Subsequently the attacker could overflow a buffer and overwrite the saved frame pointer to point to the fake stack frame. Upon exiting the current function, the stack pointer would be made to point to the fake stack frame. Once inside the fake stack frame, the fake return address would get invoked and control would be transferred to the attack code. It is interesting to note that overwriting even a single byte of the saved frame pointer can be enough to yield execution control to the attacker [66]. This type of an attack could be used to exploit an "off-by-one bug" in code. Well known off-by-one bugs appeared in Apache's mod_ssl and WU_FTPD's glob [44, p.138].

It is possible to change the execution flow of a program by overwriting a function pointer. To carry out this attack, an attacker needs to find a function pointer that is stored adjacent to a buffer. This buffer can be located in any of the four regions of process memory, the stack, the heap, the bss region or the data region. The attacker overflows the buffer and overwrites the function pointer with a pointer to some other code. When the program tries to follow the function pointer, it ends up jumping to the attacker's desired location. This type of an attack was used against the superprobe program for Linux [23]. Another example of an attack that overwrites function pointers uses the unlink() technique introduced by Solar Designer[77]. This attack tricks Doug Lea's Malloc into processing a carefully crafted fake component of heap memory using the unlink() macro. As a result, an attacker is able to overwrite a function pointer stored in the Global Offset Table with a pointer to desired code. This attack has been carried out in the wild against

vulnerable versions of the Netscape browser, traceroute (a utility for tracing the route an IP packet follows from one host to another), and slocate (a program for securely indexing and searching for files on a file system).

Finally, it is possible to mount an attack by overwriting a longjmp buffer. Setjmp and longjmp are special constructs that may be used to control program flow in C. By calling setjmp(buffer), the address of the next program instruction is stored in the specified buffer. Following the setjmp call, one can later jump to the address specified in the buffer by calling longjmp(buffer). An attacker can overwrite this buffer with a pointer to a desired location. Just like function pointers, the buffer may be stored in any of the four regions of process memory. This type of an attack appeared against Perl 5.003 [23]. A longjmp was used by Perl to recover when a buffer overflow was detected. The attacker corrupted the longjmp buffer by overflowing an adjacent buffer. When the buffer overflow was detected, the recovery mode was induced, causing a jump to the attack code.

2.2 Logic-based Attacks

It is possible to have logic-based buffer overflow attacks where the attacker changes the logic of the program by overwriting a program variable. Here is a simple example of a logic-based buffer overflow. Imagine that a character buffer gets stored on the stack adjacent to a Boolean flag that specifies whether or not the user running the program can view a certain private file. By flipping the bit of the flag, say from 0 to 1, an attacker could gain privileges to view the private files. A famous instance of a logic-based attack occurred in the Morris Worm exploit. The Morris Worm used a logic-based buffer overflow attack that corrupted the name of a file that would get executed by fingerd [23]. Logic-based attacks can involve variables stored on the stack, the heap, the data or the bss regions of memory. Logic-based attacks are not as common as some of the other attacks, but are just as dangerous.

2.3 Attacker Injects Code

An attacker may inject code via the string that is used to overflow a buffer. If this is the case, in stack-based attacks, the attacker usually overwrites the return address with a pointer back to the buffer. It is also possible for an attacker to use two buffers to carry out an attack. An attacker may inject code into one buffer, without overflowing it, and overflow another buffer, overwriting an adjacent code pointer to point to the first buffer. This attack may be used when the unchecked, overflowable buffer is not large enough to contain the attack code. It may also be used in the case when the overflowable buffer does have bound checking, but the bound check is done incorrectly. In other words, it may be possible to overflow the buffer by a few bytes, but not enough to insert the attack

code. The two used buffers may be located in different regions of memory. For example, the buffer containing the attack code may be on the stack, but the overflowed buffer and the code pointer may be stored on the heap. If the buffer containing the code is on the stack, it is possible to prevent this type of an attack by making the stack non-executable, as was done with OpenWall's Linux kernel patch [68]. Finally, an attacker may be able to use an environment variable to store the attack code. If the attacker has access to an environment variable, it may be possible to write the attack code into the environment variable, and then overflow some buffer on the stack, overwriting a function's return address with the address of the environment variable. Alternatively, it may also be possible for an attacker to overwrite some function pointer and point it to the environment variable.

2.4 Attacker Uses Existing Code

An attacker can circumvent a non-executable stack defense by overwriting the return address of the vulnerable function with the address of another function in the program, or even a shared library function. Such attacks do not involve code stored in the overflow string. The attacks that cause a jump into a standard libc function are commonly known as return-into-libc attacks [87, 88]. The basic idea of the classical return-into-libc attack is to overwrite a function's return address with the address of a standard C library routine, such as system(). By also cleverly placing the arguments to this C routine on the stack, it is possible to execute function calls such as system("\bin\sh"), i.e. start a shell. It is even possible to chain several libc function calls together.

Fortunately, Solar Designer has modified OpenWall's Linux kernel patch to prevent return-into-libc attacks by mapping libc into the 0x00... memory range. Since usually the attackers do not know the exact location of the return address on the stack, the way they manage to overwrite the return address is by repeating the desired address many times in the buffer until it hits the location of the return address. Since the address of each libc function now contains a null byte (a buffer terminator character), using a string function to overflow a buffer and overwrite the buffer with the address of the attack code would not work (using the repeated address method). In this case an attacker would be required to know the exact location of the return address. Even if the attacker is able to guess the exact location of the return address, having a null byte in the libc function address would prevent him from writing the desired function arguments into the buffer. It might be possible to cause damage by calling a zero-argument libc function, but the attacker's job has been made much more difficult. A company by the name of Entercept Security Technologies also has developed a patented solution for protecting against return-into-libc attacks [31].

2.5 Heap-based Buffer Overflows

Heap-based buffer overflows are becoming more common because of non-executable stacks. Although it is possible to have non-executable heaps, heaps are more likely to be executable than stacks. Other major deterrents of stack-based buffer overflows are dynamic tools such as StackGuard, StackShield, and ProPolice (See 3.2). Unfortunately, a similar method for protecting heaps is much more complex and does not yet exist. A fairly recent attack involving a heap-based buffer overflow was the infamous Code Red Worm of 2001 [19]. It should be noted that heap-based attacks are usually more complex and are more difficult to mount than stack-based attacks. First, the unlink() technique mentioned earlier requires some understanding of the of the heap memory management routines. Second, in order to mount a successful heap buffer overflow attack, an attacker must find some security critical variable on the heap that can be overflowed, e.g., a function pointer, a filename or a user id. This is usually a pretty difficult task. Two examples of real heap overflow vulnerabilities occurred in Sun Solaris Xsun and Microsoft IIS [90, 42]. An exploitable heap overflow vulnerability also appeared in a section of code in BIND responsible for handling transaction signatures (See 5.3.4).

2.6 Is the Buffer Overflow Exploitable?

As was mentioned in section 1.5, determining whether or not a buffer overflow is exploitable can be a very difficult task. Once a buffer overflow vulnerability is discovered, people usually try to come up with a script or a program, a.k.a. an exploit, that would invoke the vulnerable program with particular inputs, cause a buffer overflow, and lead to a security violation. Sometimes it might be possible to write an exploit under laboratory conditions, but not under any ordinary real-life conditions. Under laboratory conditions one might make certain assumptions that might be hard to justify under real-life conditions. One might assume that the attacker has access to certain environment variables, that certain program options have been enabled (e.g. a debug mode), or that certain flags have been used to compile the program. Vulnerability databases such as ICAT rank the discovered buffer overflow vulnerabilities according to levels of severity. The most severe buffer overflows are those that can be exploited remotely and allow the attacker to execute arbitrary code. Less severe buffer overflows might simply cause a program to crash, resulting in a denial of service, or they might allow a local user to elevate their privileges. It is also possible for a buffer overflow to be absolutely harmless. For instance, imagine an off-by-one bug inside some function that causes a stack buffer to be overflowed, as a result overwriting one byte of an adjacent local

variable. If that local variable has already been used and its value never gets referenced following the buffer overflow, overwriting the variable will not cause any harm. Sometimes however a buffer overflow might overwrite one or more local variables and drastically change the logic of the program. Although this might not allow the attacker to execute arbitrary code, and it might not even crash the program, it still is a serious vulnerability, for it would cause the program to behave incorrectly. As mentioned before, there have been several serious logic-based buffer overflow attacks in real programs.

Chapter 3 Approaches to detecting/preventing buffer overflows

3.0 Types of Software Security Tools

Several approaches have been taken to try to detect or prevent buffer overflow vulnerabilities. These include the following:

1) Dynamic Testing
2) Other Dynamic Analysis Approaches (Mini-simulation + Fault Injection)
3) Compiler-based Dynamic Prevention Tools
4) Language-based Approach
5) Static/Dynamic Hybrids
6) Operating System Approach
7) Static Source Code Analysis

3.0.0 Dynamic Testing Tools

Dynamic testing tools rely on instrumenting the program (source code or binary) to add buffer bounds checks. Dynamic testing tools allow one to discover buffer overflow vulnerabilities in a program by running it on different test cases, for instance with different user inputs or different options set. Most dynamic testing tools that instrument the binary code or the source code slow down the program execution significantly. Also, tools such as Purify consume large amounts of memory. This performance/memory cost might sometimes be acceptable during the testing phase, but it poses a serious problem if the instrumented code is to be used during normal program runs. A good thing about Purify is that it requires no source code. Gdb is a less powerful tool than Purify that allows programmers to step through their program as it executes, allowing them to see the contents of the stack and the heap. BoundsChecker is a dynamic testing tool for Windows that is able to diagnose various errors in the static, stack and heap regions of memory. BoundsChecker also detects things like memory and resource leaks.

Pros:

Dynamic testing tools have the advantage that all program values are known at run-time. It is therefore easy to catch illegal pointer de-referencing and buffer bounds violations. Once an error occurs, finding the cause of the error using dynamic tools is easy, since the state of the heap and all program variables can be clearly mapped out.

Cons:

The dynamic testing approach has some serious drawbacks. First, many buffer overflow vulnerabilities occur in rarely used code paths. Testing such rare code paths using dynamic techniques is very difficult and far more time consuming than using static analysis tools because it requires test cases that cause the paths to be traversed; unless an input that causes the buffer overflow is used, the buffer overflow vulnerability will not be detected via dynamic testing techniques. Second, some program code cannot be tested efficiently with dynamic testing techniques. For instance, a significant portion of operating system code lies in device drivers and cannot be tested without all the devices being installed. This is a serious inconvenience. Probably the biggest drawback of most dynamic analysis tools is the large performance cost. For instance, the dynamic memory checker, Purify [41], slows down program execution by 10x-30x and substantially increases memory usage.

Dynamic Testing Tools	DOMAIN	EFFECT	COMMENTS
Purify [41]	C, C++ (source code not required)	Instruments binaries to check for buffer access errors, memory leaks and many other errors. Purify catches illegal accesses to uninitialized or unallocated memory locations, but does not detect invalid accesses to allocated memory locations.	Good for debugging. Slows down program execution by 10x-30x and increases memory usage substantially.
gdb (GNU source-level debugger) [40]	C, C++ (source code required)	Can step through the program as it executes. Allows one to analyze the contents of the stack and the heap during run-time.	A debugging tool. Not as powerful as Purify.
BoundsChecker[11]	C++, Delphi (source code required)	Detects and diagnoses errors in static, stack and heap memory, and also finds memory and resource leaks. Can find potential buffer overflows at run-time. Checks for proper API usage.	A debugging tool for Visual Studio. Works on Windows platform and Microsoft .NET Framework.

Table 2: Dynamic Detection/Prevention Tools.

3.0.1 Other Dynamic Analysis Approaches

An interesting dynamic approach to testing protocol security has been undertaken by a group of researchers at the University of Oulu, Finland [50]. Their project is known as PROTOS. The researchers working on PROTOS have developed a method of functional testing (black box testing) known as mini-simulation. The focus of PROTOS is on applications that use standard protocols such as HTTP, TCP, SNMP etc. For a

given application, the behavior of the protocol is defined manually using attribute grammars. Source code for the tested application is not required. Once the attribute grammar for the protocol has been created, a large class of anomalous and normal test cases is generated automatically. Test cases include things like malformed packet headers, invalid checksums etc. The attribute grammar specifying the protocol behavior is mutated using the test cases, and then protocol execution gets simulated for each test case. An example simulation might involve a simulated HTTP client sending an invalid request for a file to a real HTTP server. For each test case, the behavior of the HTTP server would be observed, namely, did it crash or hang. Fully automating many test runs is quite difficult for the simple reason that, if the application crashes, it needs to be restarted. The PROTOS researchers tested several applications, including HTTP browsers, an LDAP server, and SNMP agents and management stations. Overall, 49 different product versions were tested and 40 of them were found to be vulnerable to at least a denial-of-service attack (buffer overflow exploits were constructed). Unfortunately, preparing a single test suite took on average 4.8 man months!

Yet another fairly popular dynamic analysis technique is known as software fault injection. This technique forces a system to go into an anomalous state during execution by injecting faults into the source code. Faults are often injected at program interfaces, both internally and externally. For example, when testing a server, one can send too many requests to the server, or send it invalid requests. Also, one can make one or more modules of the program act in unexpected ways, i.e. make a module stop responding or start sending invalid inputs to other modules. By observing how the system behaves under anomalous conditions, it is often possible to discover certain inherent vulnerabilities in the software. A tool by the name of FIST, Fault Injection Security Tool, developed by Ghosh et al shows some promise [37]. Unfortunately, FIST requires users to manually prepare code for testing. This is a serious drawback. Other popular fault injection tools are Fuzz and Ballista [64][52].

Pros:

The PROTOS mini-simulation tool can generate many anomalous test cases automatically, which can be used to find robustness problems in programs. Fault injection tools allow you to simulate unexpected inputs and module failures, which can help identify robustness problems. All of these approaches are language independent and do not require source code.

Cons:

The mini-simulation technique requires a significant amount of preparation time for each program, on the order of several months. Automating the execution of programs using the mini-simulation technique is not easy due to system crashes and hanging. Similarly, fault injection methods require a great deal of user intervention. Users must inspect the output for each failed test case resulting in a system crash and determine where the error might have occurred. The test cases generated by these tools do not exhaustively cover all possible execution paths. Some applications might have security features that prevent them from being tested via fault injection.

TOOL	DOMAIN	EFFECT	COMMENTS
PROTOS Mini-Simulation [50]	Requires a grammar specifying protocol behavior (application src., code not req.)	Automatically generates many classes of normal and anomalous test cases for application robustness testing.	Time required to prepare program for mini-simulation testing (several man months per program). Program execution not easy to automate (due to halts and crashes).
FIST, Fuzz, Ballista [37][64][52]	Different applications. (source not required)	Simulate bad user inputs, module failures, unusual module interactions etc. Can be used to find robustness problems and possible buffer overflow conditions.	Users must prepare code manually for testing.

Table 3. Other dynamic testing tools.

3.0.2 Compiler-based Dynamic Prevention Tools

The goal of compiler-based prevention tools is to dynamically prevent buffer overflow exploits from happening. StackGuard and StackShield are two such tools (see Table 4). These tools compile programs using special compilers, placing in protection mechanisms that detect and stop specific buffer overflow exploits during runtime.

The basic idea of StackGuard is to write a special value, known as a canary (in reference to the canaries used by coal miners to detect the presence of dangerous gases), next to the function's return address on the stack. If the return address gets overwritten, then so does the canary value. Thus, by monitoring the canary value, StackGuard can effectively detect any stack-smashing attack. Two types of canaries can be used, a random 32-bit value calculated at run-time, or a 4-byte "terminator canary" consisting of a null byte, a carriage return, a line-feed, and an "EOF". The random canary works because the attacker is unable to guess the canary value. The terminator canary works because the set of standard C string functions rely on terminator bytes to terminate character buffers, thus making it impossible to write over and past the terminator canary using any of the standard C string functions. Although both of these canary schemes protect against many stack-smashing attacks, they cannot protect against less common attacks such as the one pointed out by Mariusz Woloszyn (a.k.a "Emsi") [14][30]. The Emsi vulnerability involves overwriting a function pointer to point to the return address of a function, and then overwriting the return address through the function pointer. To deal with such attacks, the StackGuard team came up with a new canary scheme, whereby the canary placed on the stack is the XOR of a random 32-bit value and the return address. The random 32-bit value is saved separately in memory. When a function exits, the 32-bit value is fetched from memory and is XORed with the return address on the stack. If the result does not match the stored canary on the stack, execution is halted and a security alert is raised. Unfortunately, the StackGuard team decided to drop support for the random XOR canary [85]. The reasons for doing so were not too apparent. A better tool, known as PointGuard, is currently being developed, and will use the canary method to provide general pointer protection.

StackShield is capable of preventing many stack-based attacks, as well. StackShield implements two security modes, the Global Ret Stack (GRT) and the Ret Change Check (RCC). The GRT is a separate stack for storing function return addresses of functions called during execution. Upon entering a function, the return address is placed on the stack and is simultaneously copied into the GRT, and upon exiting the function, the address stored in the GRT is used to replace the return address stored on the stack. Thus, any attacks that overwrite the return address on the stack get stopped by this method, and execution continues without interruption. The user is not even aware that an attack was attempted. The drawback of this scheme is that local stack variables could still have been corrupted. Another limitation of this scheme is that the GRT stores 256 entries by default, thus limiting the stack depth to 256 frames. The other option, RCC, is simpler and faster. RCC uses a single global variable to store the return address of the invoked function. When the function returns, the global variable's value is checked against the return address stored on the stack. If the two values differ, the program is halted and an alert is raised. Finally, StackShield also attempts to safeguard function pointers. The key idea is that function pointers should only point to the text segment of the process memory. Any attacks that attempt to point a function pointer to injected code will end up pointing to a location in the stack, bss, data or heap regions of process memory. Thus, restricting function pointers to point only to the text segment of the memory should prevent such attacks. StackShield adds code before each function call that uses function pointers. The code declares a global variable in the data segment of memory. This variable is used as a boundary tag. Before dereferencing any function pointer, StackShield makes sure that the function pointer points to a memory location below that of the global variable. If the function pointer points above the global variable, execution is halted.

ProPolice is a gcc patch developed by Etoh and Yoda. It is very similar to StackGuard in its main approach – canaries are used to detect stack-smashing attacks. What is novel about ProPolice is that it tries to protect local variables, i.e. it can prevent many logic-based buffer overflow attacks. It goes about this by allocating all the character buffers at the bottom of the stack, next to the old base pointer. This ensures that when a buffer gets overflowed, none of the local variables can get overwritten. The unfortunate thing about ProPolice is that it was noticed to be quite unstable [85]. According to Wilander and Kamkar, ProPolice was the most effective of the three compiler-based tools discussed. During testing, ProPolice prevented 8 out of 20 buffer overflow attacks without halting the program, and stopped 2 out of 20 by halting the process.

Microsoft's C/C++ compiler from Visual Studio .NET comes with a special /GS option that allows developers to build their applications with a so-called 'buffer security check'. The /GS option is essentially a Win32 port of Crispin Cowan's StackGuard. Unfortunately, the /GS protection mechanism can be bypassed. When the /GS option is enabled, a special user-defined handler can be installed which gets called when a stack-smashing attack is detected. The address of the user handler's code is stored in a global variable called user_handler. Suppose that during a buffer overflow attack the attacker somehow overwrites some pointer, P, and makes it point to the location of the global variable user_handler. Now, suppose later on in the function, the contents of some

buffer, B, get copied to the location pointed to by the local pointer P. If during the buffer overflow the attacker can overwrite B, he can overwrite it with arbitrary data, i.e. address of the attack code stored in the overflowed buffer. Thus, when the stack-smashing attack finally gets detected, the user-defined handler will be called. Unfortunately, the variable storing the address of the user handler code will have been overwritten, so the program will jump to the attacker's code and execute arbitrary instructions (See [62] for an example exploit).

Last but not least, several people have also proposed adding patches to the standard gcc compiler that would add bounds checking code to buffers [12][58][48]. Table 4 lists some of the well known compiler-based tools just mentioned.

Pros:

Compiler-based tools automatically install protection, and programmers do not need to modify source code. These tools can detect and stop specific types of buffer overflow exploits during run-time. For instance, StackGuard and StackShield can prevent stack-smashing attacks. Tools such as PointGuard [23] can detect many potential problems with illegal pointer de-referencing or pointer-overwrite attacks.

Cons:

All source code needs to be recompiled with a special compiler. Although such tools can prevent many buffer overflow exploits, their main drawback is that they usually turn a buffer overflow attack into a denial of service attack by terminating the program. All of the discussed compiler-based tools are not able to prevent buffer overflow attacks on buffers in the heap/BSS/data regions of memory. The problem with compiler patches is that they impose a large performance penalty and an increase in code size. Execution time and code size may experience an increase by a factor of three or more [38].

See Table 4 on the following page.

Compiler-based Tools	DOMAIN	EFFECT	COMMENTS
StackGuard [25]	Protects C source programs	Detects stack-smashing attacks by placing a canary between the function's return address and the old base pointer.	Halts program when attack is detected.
ProPolice[33]	Protects C source programs	Places canaries before the old base pointer. Is able to detect typical stack-smashing attacks. Also protects non-buffer stack allocated variables.	Halts program when attack is detected. ProPolice is somewhat unstable.
StackShield [78]	Protects C source programs	Detects stack-smashing attacks. Can also detect frame pointer and function pointer overwrite attacks.	Two modes: Global Ret Stack, Ret Range Check GRT: prevents stack-smashing attacks, does not terminate program RCC: detects attack and halts program.
Microsoft's /GS option [62]	C/C++ source programs	Very similar to StackGuard. Prevents stack-smashing attacks using canaries.	Halts program when attack is detected. Part of the C/C++ compiler in Visual Studio .NET.
PointGuard [23]	Protects C source programs	Can detect many function pointer overwrite attacks.	Places canaries next to code pointers. (function pointers and longjmp buffers) Halts program when illegal pointer dereference is detected.
gcc compiler patch [12][58]	Protects C source programs	Modifies compiler to add bounds checking for each buffer.	Execution time and program size can increase by 2x.
Jones and Kelly [48]	Protects C source programs	A gcc patch that performs dynamic buffer bounds checking without modifying pointer representation.	Slows down program by 5x-6x.

Table 4: Compiler-based Dynamic Buffer Overflow Prevention Tools.

3.0.3 Language-based Approach

Some researchers have attempted to solve the problem of buffer overflows by implementing safer alternatives of the dangerous C functions (See Table 3). Libsafe [6][8][9] is a dynamically loaded library that intercepts calls to dangerous routines in the shared glibc library (including strcpy(), strcat(), getwd(), gets(), scanf(), realpath(), sprintf()) and replaces them with safer wrapper functions. Each wrapper function computes a maximum size bound for each local buffer. This bound is computed based on the distance between the beginning of each buffer and the beginning of the saved frame pointer. If the source buffer of the string function is within the established size bound, the corresponding string function is executed normally. If however the source buffer exceeds the maximum allowable buffer size, then like StackGuard, Libsafe terminates the program, thus turning a buffer overflow attack into a denial of service attack. Although Libsafe protects against stack smashing attacks involving common string functions, it does not prevent attacks that overflow a stack buffer and overwrite another local variable.

Another limitation of the Libsafe solution is that it does not work on code compiled with the -fomit-frame-pointer switch, which is often used to give the GCC/x86 compiler one more register to allocate. [26].

An enhancement of libsafe known as Libverify has been developed. For each function call, Libverify copies the return address onto a special "canary stack" stored on the heap. Upon exiting the function, the return address stored on the stack is compared against the address stored on the canary stack. If the two addresses match, execution continues normally, but if they differ, the program is terminated, and an alert is issued. Libverify is able to protect against exploitation of a larger class of string functions than Libsafe. A major shortcoming of Libverify is that the integrity of the canary stack itself is not protected.

Microsoft has developed a library known as StrSafe that implements safe string handling functions [79]. To use the StrSafe library one would have to replace every occurrence of an unsafe C string handling function with the safer StrSafe alternative. This would require significant programmer effort, and thus would not be an ideal solution for dealing with legacy code.

Another language-based approach has been to implement safe string modules, such as the C++ string module or Libmib [61]. Using such modules would mean that all source code would need to be rewritten. Also, interfacing with old libraries that rely on unsafe buffer implementations would be extremely difficult [38].

Pros:

Programs that have been made compatible with the safe libraries or the safe string modules would avoid buffer overflows. Also, future programs written using these safer language constructs would eliminate buffer overflows.

Cons:

As mentioned above, libsafe and libverify transform buffer overflow attacks into denial of service attacks. Also, they don't prevent logic-based attacks, where a local stack variable gets overwritten. Libsafe doesn't work on code compiled with the -fomit-frame-pointer switch. Libverify has the weakness that the canary stack isn't protected. In order to work with libraries such as StrSafe or new string modules such as libmib, much of the old code would have to be rewritten. Interfacing with old libraries that rely on unsafe buffer implementations would be difficult.

See Table 5 on the following page.

Language-based Solutions	DOMAIN	EFFECT	COMMENTS
Libsafe [6] [8][9]	glibc library	Dynamically loaded library that replaces calls to unsafe glibc functions with safer variants. Prevents stack smashing attacks.	Terminates the program when an illegal buffer access is detected. Does not work on code compiled with fomit-frame-pointer switch. Does not stop overflows within local stack variables, e.g. overwriting local function pointers.
Libverify[8]	glibc library	Enhancement of Libsafe that alters all functions to copy the return address to a "canary stack" on the heap. This address is compared against the one on the stack before exiting function.	If return address on function stack does not match the address stored on the canary stack, the program is terminated and alert is raised. Integrity of canary stack is not protected.
StrSafe [79]	Protects C programs	A library of safer C string handling functions.	Would require rewriting large parts of the code. Not good for securing legacy code.
Libmib [61]	Protects C programs	Implements a safe string module.	Old code would have to be rewritten. Interfacing with old libraries that use unsafe buffers would be difficult.

Table 5: Language-based Solutions.

3.0.4 Static/Dynamic Hybrids

A tool called CCured attempts to transform C code into a safer dialect with little or no user intervention. CCured uses a compiler that performs static analysis on a C program and attempts to determine which pointer uses are safe and which are not. If it cannot ascertain that a pointer is used safely, the compiler adds efficient checks to the source code that are meant to catch out-of-bounds references at runtime. This selective insertion of run-time checks helps keep the performance overhead fairly low. The instrumented C program usually experiences a slow-down of about a factor of two, which might not be too high a price to pay if good code security is desired. If a buffer overflow occurs during the execution of a CCured-compiled program, the program gets terminated, and an error message is displayed. Thus, CCured does not eliminate the actual buffer overflow from a program; instead, it turns an overflow into a denial of service. It should also be noted that sometimes the CCured compiler is not able to transform the C source

code into safe code automatically and requires that the user make slight modifications to the original source code.

Another group of researchers has developed a safer dialect of C known as Cyclone. The Cyclone compiler uses a hybrid static/dynamic approach, similar to that of the CCured tool. The Cyclone compiler performs static analysis on the Cyclone source code and inserts runtime checks into the compiled output whenever it can't ascertain that the code is safe. Cyclone has several classes of pointers that are represented differently from C. For instance, the so-called "fat pointer" in Cyclone stores not only an address, but also certain bounds information. Using fat pointers allows Cyclone to better ensure that pointers are within bounds. The major disadvantage of Cyclone is the difficulty of porting C code into Cyclone. While CCured takes C code and translates it into safer code with little or no code annotations required by the user, porting C code to Cyclone usually requires programmers to change about 10% of the code by inserting special Cyclone commands. Only after the programmer has changed about 10% of the C program by hand, can the Cyclone compiler do its magic. Cyclone differs from C enough so that veteran C programmers might be reluctant to use it [47]. See table 6.

TOOL	DOMAIN	EFFECT	COMMENTS
CCured [65]	C (source code required)	Hybrid static-dynamic tool that uses a type inference algorithm to eliminate the need for too many checks. Forbids certain pointer-integer conversions and some other C idioms. Representation of certain pointers is changed. Source code is transformed into safer code.	Speed penalty due to runtime checks is 0-150%. Performs fewer checks than Purify and does not consume a lot of extra memory. Because of the modif., pointer representation, interfacing with binary-only libraries is complicated.
Cyclone[47]	Dialect of C	Cyclone compiler does static analysis on Cyclone source code and inserts runtime checks into compiled output in places where it can't ascertain safety. Checks for NULL pointer dereferencing are added.	Cyclone designed to eliminate buffer overflows. Porting from C code to Cyclone is not very easy; about 10% of code needs to be modified. Some Cyclone applications are up to 3 xs slower than the C programs. Using fat pointers causes a large space overhead.

Table 6. Static/Dynamic Hybrid Tools.

Pros:

Writing programs in a safer dialect of C, such as Cyclone, would make them less susceptible to buffer overflows. Porting a C program to Cyclone is more realistic than porting it to Java. CCured and Cyclone stop buffer overflows from causing damage by terminating the program. CCured and Cyclone try to instrument the program with run-time checks efficiently, inserting them in only special cases, rather than blindly instrumenting all buffer accesses.

Cons:

Porting from C to Cyclone is not very easy. Cyclone is significantly different from C. Program slow-down will be noticeable; Cyclone applications can run up to three times slower than C applications, and CCured programs can experience a slow-down of a factor of two. Both CCured and Cyclone turn buffer overflow attacks into denial of service by terminating the program. Because CCured modifies the internal representation of pointers, being able to interface with binary only programs is difficult.

3.0.5 Operating System Solutions

Some researchers have suggested an operating system approach to solving the buffer overflow problem (See Table 7). For instance, Solar Designer has implemented a Linux kernel patch for a non-executable stack [68]. This patch is capable of preventing stack-smashing attacks. In addition, this patch maps the shared libc routines into the 0x00... memory range. This prevents a class of attacks known as return-into-libc (discussed later). There is a drawback to this patch solution, however; some processes actually require code to be executed on the stack. For instance, Linux relies on an executable stack for its signal handlers, and gcc uses an executable stack for function trampolines for nested functions. Solar Designer's patch detects a process that uses trampoline calls and makes the stack executable for that process. The patch tries to accommodate signal handlers by making the stack only temporarily executable for the duration of the signal handler. Thus, these special cases of trampoline calls and signal handlers could potentially be exploited for buffer overflow attacks [24]. Recently, the OpenBSD group announced an upcoming release of its operating system that will attempt to eliminate buffer overflows completely [55]. Their solution is based on three ideas: the use of a randomized memory location for the stack, the use of canaries to detect stack-smashing attacks, and the separation of main memory into two distinct parts, one writable and one executable, but not both. The goal of this last feature is to prevent attackers from writing and then executing their own code.

Pros:

The Linux kernel patch can prevent stack-smashing attacks and return-into-libc attacks. Programs run on the new OpenBSD system would be more resistant to buffer overflow attacks.

Cons:

The Linux kernel patch does not protect against heap-based buffer overflow attacks. The patch does not prevent all stack-based attacks; signal handlers and nested functions are vulnerable to buffer overflow attacks. Code run on other operating systems is not protected by this patch. All operating systems for which the program is intended would need to be patched. Porting legacy code to a new operating system would

probably be difficult.

Operating System Solutions	DOMAIN	EFFECT	COMMENTS
OpenWall [68]	Linux 2.2 kernels	Stack is made non-executable. Prevents stack-smashing attacks. Also prevents most return-into-libc attacks.	Maps shared libc routines into 0x00... memory range. This prevents most return-into-libc attacks. Does not prevent all stack-based buffer overflows. Signal handlers and trampoline function calls are allowed to execute code on the stack, and thus are susceptible to buffer overflow attacks.
OpenBSD [55]	OpenBSD OS	Prevents many buffer overflows.	Uses a randomized stack memory location and canaries to detect stack-smashing attacks. Also, divides main memory into two sections, one writable and one executable.

Table 7: Operating System Solutions.

3.0.6 Static Source Code Analyzers

Static analysis tools attempt to find buffer overflow vulnerabilities in source code (sometimes binary code) without running it. Table 8 lists some of the existing static analysis tools. Most of the tools listed will be evaluated in this thesis. The set of static source code analyzers can be divided into two main categories: lexical analysis tools (or syntactic) and "deep analysis tools" (or semantic) (See Tables 8 and 9 below). The lexical tools are based on pattern matching; they find dangerous looking syntax or the use of dangerous functions, and subsequently raise a flag. Lexical tools do no semantic analysis on the source code. Lexical tools include RATS, FlawFinder, ITS4 and BugScan. BugScan is a commercial tool that differs from the other lexical analysis tools in that it analyzes binary code rather than source code. BugScan is able to disassemble an executable file and map the control and data flow of the program. The other group of static analysis tools attempts to perform some sort of semantic analysis on the source code. These tools are more sophisticated than the lexical analysis tools and often can catch subtle bugs that the lexical tools cannot. The advantage of lexical tools is that they are fast. They are often good for a first pass on the code, and they can help catch some of the straightforward buffer overflow vulnerabilities by pointing out dangerous code areas. The drawback of lexical tools is that they tend to generate a very large number of false alarms. Most security-sensitive legacy code has been hand-inspected and tested with some lexical tools for potential buffer overflows. In order to catch the subtle buffer overflow bugs that may still be lurking there, one must use more sophisticated tools. The deep analysis tools for finding buffer overflows include ARCHER, BOON, PolySpace C Verifier, PREfast, PREfix, Uno, Splint, and PC-Lint/Flexe-Lint.

Pros:

Unlike dynamic testing tools, some static analysis tools are **capable of exhaustively testing all possible execution paths**. Static analysis tools do not depend on being able to run the code. Static analysis tools allow you to **discover the root of the problem**, so that it can subsequently be eliminated from the code.

Cons:

A disadvantage of static code checkers is that the source code is always required. The static analysis tools may have to do complex analysis to be able to detect subtle buffer overflow vulnerabilities. Another problem with static analysis tools is that often imprecision is introduced as a result of the heuristics used for finding buffer overflows. In general, determining whether a program has a buffer overflow is an undecidable problem, so heuristics have to be used. Also, some static analyzers require significant programmer effort to annotate the code. Lastly, tracking down buffer overflow vulnerabilities based on error messages of the static analyzers can be time consuming.

Semantic Static Analysis Tools	DOMAIN	EFFECT	DEPTH OF ANALYSIS
ARCHER [89]	C source	Memory checker capable of detecting buffer overflows. Uses a form of abstract/symbolic program interpretation.	Inter-procedural, flow-sensitive, context-sensitive, fully symbolic.
PREfast* [71]	C, C++ source	A sister tool of PREfix that is able to detect several classes of run-time errors, including buffer overflows, using a form of abstract interpretation.	Works only within procedures. Runs on single file programs.
PolySpace C Verifier* [70]	C source	Detects run-time errors such as buffer overflows, divide by zero, use of uninitialized memory, integer overflows, shared memory violations etc. Uses a sophisticated form of abstract interpretation.	Works across procedures and is able to track relationships among several variables. Highly memory intensive. Deep analysis takes a long time.
BOON [83]	C source	Detects buffer overflows using integer range constraint analysis.	Inter-procedural, flow-insensitive. Catches buffer overflows only in string manipulation routines.
SPLINT [34, 35]	C source	Finds programming errors such as missing arguments, undeclared variables, improper return statements, nested comment symbols. It is also capable of finding buffer overflow and format string vulnerabilities.	Intra-procedural; requires user annotation for inter-procedural analysis.
Uno[43]	C source	Detects Uninitialized variables, dereferencing of Nil-pointers, and Out-of-bounds accesses.	Inter-procedural, control-flow sensitive
PC-Lint /FlexeLint* [67]	C/C+ source	Performs strong type checking and value tracking. Detects certain buffer overflow bugs, uninitialized variables, out-of-bounds array accesses. Checks for correct usage of certain standard C functions.	Similar to Splint. With code annotation it is possible to do inter-procedural analysis.

Table 8: Semantic Static Source Code Analyzers.
* - Commercial tools

Lexical Static Analysis Tools	DOMAIN	EFFECT	DEPTH OF ANALYSIS
ITS4 [81][18]	C/C++ source	A simple lexical tool that scans source code for potentially dangerous functions that could result in buffer overflows. Can also detect some race conditions.	Grep-like tool. Simpler than RATS and FlawFinder. Tries to assess risk level of each warning, provides simple description of problem and suggests a simple fix.
RATS [74]	C, C++, Perl, PHP, Python source	A lexical tool for detecting common buffer overflows and race conditions (such as TOCTTOU). Scans for uses of potentially dangerous C functions.	Grep-like tool. Reporting more condensed than that of FlawFinder. Uses greedy pattern searching. Also, uses some semantic inspection of program.
FlawFinder [84]	C/C++ source	A lexical tool similar to RATS. Uses a database of vulnerable functions to perform simple pattern matching on source code. Can detect some buffer overflows, format string vulnerabilities, race conditions, meta character risks, system code problems and poor random number generation.	Grep-like tool similar to RATS. Does not do any semantic analysis of source code.
BugScan* [13]	Most Windows and Linux x86 binaries	Finds bugs in executables using lexical static analysis. Disassembles the binary and maps the control and data flow. Detects things like overflows, poor random generators, race conditions.	Installed on network and accessed through web interface. Users upload binary programs to the appliance for analysis. Report can be viewed online or as XML file.

Table 9: Lexical Static Source Code Analyzers.
* - Commercial tools

3.1 Why focus on Static Analysis?

So far, different approaches to detecting buffer overflows in software have been presented. Table 10 summarizes the pros and cons of the different approaches.

Approach	PROS	CONS
Dynamic Testing	Program values known at run-time. If an error occurs at run-time, tracking it is easy. Source code is not always required.	Coming up with test cases to exercise all execution paths is very difficult. Not only that, one must come up with test cases that trigger a buffer overflow. Program execution during testing slows down a great deal, and memory usage goes up. Being able to run the program is not always convenient. (testing device driver code requires the device)
Mini-Sim., & Fault Injection	Language independent. Source code not required. Generate many test cases automatically. Good for robustness testing.	Programs require significant preparation time. User must analyze the failed tests to determine what the source of the problem is. Automating the execution of the program is not easy due to system crashes/hanging.
Dynamic Prevention	Can stop specific types of buffer overflows from causing damage during run-time.	Effectively turn buffer overflow attacks into denial of service by terminating the program. Source code needs to be recompiled with special compiler. Compiler patches usually slow down program execution significantly.
Language approach	Stop buffer overflows by using safe libraries or string modules.	Many of these approaches transform buffer overflows into denial of service. Interfacing with old libraries is difficult. Not ideal for securing legacy code.
Static/Dynamic Hybrids	Safer dialects such as Cyclone minimize the chance of buffer overflows. Porting from C to CCured or Cyclone is more realistic than porting to Java. Cyclone and CCured can stop buffer overflow attacks at runtime.	Porting from C to Cyclone is not trivial and requires one to modify ~10% of the code. Cyclone and CCured turn a buffer overflow into a denial of service.
OS-based Solutions	Can prevent many buffer overflow attacks (esp., stack-smashing).	Linux kernel patch does not prevent heap-based overflows. Making the stack non-executable still leaves room for certain stack-based buffer overflows, i.e. in signal handlers. Porting a legacy program to a new operating system might not be trivial.
Static Analysis	All execution paths can be analyzed without running the program. Discover the root of the problem, so that it can be eliminated.	Source code is always required. Imprecision due to analysis heuristics exists. Sometimes there are many false positives. 100% detection rate is theoretically impossible. Sometimes require users to annotate code.

Table 10. Pros and Cons of different approaches to detecting/preventing buffer overflows

The most effective way to combat buffer overflows is to eliminate them from the source code before the software gets released to the public, i.e. get down to the root of the

problem. We argue that the best way to achieve this is through static source code analysis. Let us first summarize the deficiencies of the other approaches.

As was discussed above, dynamic testing tools have a large performance overhead, and it is almost impossible to dynamically test every execution path. Without having a test case that triggers a buffer overflow, a dynamic testing tool such as Purify is useless for finding buffer overflows. The mini-simulation and fault injection approaches have the same drawback - the automatically generated test cases are not guaranteed to exhaustively cover all execution paths. In both the mini-simulation approach and the fault injection approach, a great deal of user-intervention is required: 1) the user must first prepare the program for testing, which can take months, as in the case of the PROTOS mini-simulation tool, 2) the user usually must be present to restart the system whenever it hangs or crashes during testing and 3) following the test runs, the user must inspect the output of the failed test cases and try to determine the source of the problem, which is not a trivial task. Currently, none of the runtime prevention tools can prevent all types of buffer overflows, and more importantly, such tools turn buffer overflow attacks into denial of service attacks. The operating system approach has succeeded in preventing only certain kinds of buffer overflows. This approach also presents a portability issue for legacy source code; the old code would need to run on the new operating system. The language-based approaches of implementing safer string handling functions or safer string modules have their own share of serious drawbacks, as described earlier. The language-based solution may eliminate buffer overflows from future programs, but it is not ideal for securing legacy code. Finally, porting C programs to safer dialects such as Cyclone can be non-trivial. As a programming language, Cyclone is sufficiently different from C, that it would be a challenge convincing veteran C programmers to abandon C altogether and program in Cyclone instead. Both Cyclone and CCured have the same drawback as other dynamic prevention tools; they turn a buffer overflow attack into a denial of service attack.

Static source code analysis tools get down to the root of the problem by trying to detect actual errors in the code. By pinpointing the bug in the program, the programmer can then fix the code and eliminate the possibility of a disaster occurring. Most importantly, static analysis tools have the capability to analyze all execution paths in the program and thus can find insidious bugs that lurk in rarely executed code paths that dynamic testing techniques would probably miss. One might argue that a disadvantage of static source code analysis is that source code is always required, but in the case of analyzing open-source programs, this is not a problem. And if a program is not open-source, it should be the manufacturer's responsibility to ensure that the software is bug-free. The bottom line is that programmers need good development tools. During software development, source code will always be available.

A disadvantage of many static analysis tools is that they are imprecise and often yield many false positives. Imprecision can result from heuristics used in analyzing source code. Research in improving static analysis techniques is currently quite active. Many C programs are so complex that not having any false positives is almost impossible. Also, unfortunately, having a 100% detection rate is theoretically impossible

(See Halting Problem or Rice's Theorem). However, if we can create a tool that detects most bugs, most of the time, then that would be a huge success.

Since there is such a vast amount of open-source, legacy programs written in C, the primary objective right now should be to secure these programs. A secondary objective should be to come up with development tools that will help ensure that future programs that get written are free of buffer overflow vulnerabilities and other bugs. Of all the approaches discussed so far, static source code analysis offers the greatest potential for meeting these two important objectives. This thesis therefore focuses on current state of the art static source code analysis tools.

Chapter 4 Description of the static analysis tools

4.0 Specific tools

The set of static analysis tools can be divided into two broad categories - the set of tools that do only lexical analysis and the set of tools that do some sort of semantic analysis. The three most well known lexical analysis tools are ITS4, Flawfinder and RATS. This thesis will focus on five semantic analysis tools: PolySpace's C Verifier, ARCHER, BOON, Splint and Uno. Some other semantic analysis tools that were not included in this evaluation are PREfast, PREfix, and Flexe-Lint. PREfast is a tool designed for testing Windows software, specifically device driver code, and since the programs we analyzed were all written for Unix-based systems, we decided to exclude PREfast from the evaluation. PREfix is a heavy-duty, inter-procedural tool used internally by Microsoft. Unfortunately, we were not able to get a copy of PREfix for evaluation purposes. Flexe-Lint is a commercial tool similar to Splint.

4.1 Lexical Analysis

Lexical analysis tools use pattern matching, sort of like the Unix grep command, to find dangerous code constructs. Flawfinder, RATS and ITS4, all have a database of names of standard C functions that could potentially lead to buffer overflows if not used correctly. This database includes mostly string manipulations functions such as strcpy (), strcat (), sprintf () etc. The vulnerability database also includes certain other dangerous functions that can introduce "tainted state" into the program, for example gets() or getenv(). Both of these functions take inputs from the user-space – gets () reads in a line of characters from the standard input, and getenv () returns the contents of a specified environment variable. Thus, one should treat the outputs of these functions as tainted, i.e. one should not trust the outputs to be safe. Other possible sources of taintedness are functions that read input from files, such as fgets (), any function that take inputs from the command line, such as scanf (), and functions that read data from network packets. What differentiates the three lexical tools we've mentioned is mainly the size of their databases, the types of vulnerabilities they target, the number of programming languages they can work on, and the type of output they provide. Some of the tools attempt to sort their warnings by risk level. The risk level not only depends on the function being called, but usually the parameters, as well. This is a useful feature for programmers to have, for they can save time and concentrate on the high risk calls only, if time is a priority. The major drawback of lexical analysis tools is that they give a large number of false alarms. Most lexical analysis tools are not capable of distinguishing between safe and unsafe uses of dangerous functions, and so they end up flagging every appearance of a dangerous function call, such as strcpy(). If a program contains a hundred strcpy() calls, going through the list of warnings and weeding out the false positives could be very time consuming. The advantage of lexical analysis tools is that they are usually very fast, and they can often help programmers find simple bugs. Besides buffer overflow vulnerabilities, some of these lexical tools can find other types of bugs such as format string bugs, race conditions and poor random number generation. We will now discuss the specifics of Flawfinder, RATS and ITS4.

For comparison, we will show the output of each of these tools on the following simple program, test.c, involving an unsafe use of strcpy ():
The program is annotated to show line numbers.

test.c:

```
1.      #include <stdio.h>
2.      #include <stdlib.h>
3.      #include <string.h>
4.
5.      int main () {
6.
7.      char buf[5], bufB[] = "Abra Cadabra";
8.
9       /* BAD */
10.     strcpy(buf, bufB);
11.
12.     return 0;
13.}
```

4.1.0 Flawfinder 1.22

Flawfinder was written by David Wheeler in a programming language known as Python. The first version of this tool was released in May 2001. It should be noted that Flawfinder was developed around the same time as RATS of Secure Software Solutions. This tool uses a database of vulnerable functions to perform simple pattern matching on the source code. Flawfinder is able to understand C and C++ source code. The types of vulnerabilities it scans for are buffer overflows, format string vulnerabilities, race conditions, meta character risks, system code problems and poor random number generation. Flawfinder 1.22 has a database, or a ruleset, of 127 vulnerable functions in C/C++ code. Flawfinder sorts its warnings by risk. The risk level depends on the function being called and its parameters. For example, a call involving a constant string, e.g. strcpy(buf, "Alice), would be considered less risky than a call involving variable buffers, e.g. strcpy(buf1, buf2).

A typical warning message generated by Flawfinder gives:

1. The name of the file.
2. The line number on which the suspected error occurs.
3. The character position.
4. The name of the suspect function.
5. Description explaining the cause for alarm.
6. A possible fix.

Flawfinder allows one to specify HTML reporting of warning messages if desired.

Here is the output of Flawfinder 1.22 on test.c:

Examining test.c
test.c:10: [4] (buffer) strcpy:
Does not check for buffer overflows when copying to destination.
Consider using strncpy or strlcpy (warning, strncpy is easily misused).
test.c:7: [2] (buffer) char:
Statically-sized arrays can be overflowed. Perform bounds checking,
use functions that limit length, or ensure that the size is larger than
the maximum possible length.
Number of hits = 2
Number of Lines Analyzed = 13 in 0.69 seconds (67 lines/second)
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!

Notice that Flawfinder tells you the total number of lines analyzed, the number of warnings (hits), and the total time spent analyzing the code.

4.1.1 ITS4

The release of ITS4 was first announced in late 2000 by a group of researchers at Secure Software Solutions, now Cigital. ITS4, or It's the Software Stupid!, is a lexical analysis tool that scans C and C++ source code and builds a token stream of the code. The tokens are matched against dangerous functions in a database. The current version of ITS4 contains 145 functions in its database. These functions involve buffer overflow vulnerabilities and race conditions. The database also includes a few pseudorandom functions that could potentially be used unsafely in security-sensitive code.

A typical error message generated by ITS4 contains:

1. The name of the file.
2. The line number of the error.
3. One of six risk levels: NO_RISK, LOW_RISK, MODERATE_RISK, RISKY, VERY_RISKY, MOST_RISKY.
4. The name of the suspect function
5. Reason why alarm was raised (i.e. function is high risk for buffer overflows)
6. Whether or not the function is tainted (i.e. can retrieve inputs from external sources such as files, command line, or network packets)
7. A possible fix.

The programmer can set the risk level cut-off threshold via a command line argument (0 = no risk, 5 = most-risky). Having a cut-off of 0 would produce the most number of warnings, whereas having a cut-off of 5 would produce the fewest number of warnings.

This flexibility to set the risk threshold can be useful for a programmer who only wants to focus on warnings above a certain risk level.

Here is the output of ITS4 for test.c:

test.c:10:(Very Risky) strcpy
This function is high risk for buffer overflows
Use strncpy instead

4.1.2 RATS 2.1

RATS, or Rough Auditing Tool for Security, is a tool that was developed by Secure Software Solutions around the same time as Flawfinder. RATS has the advantage of being able to scan source code written in several languages: C, C++, Perl, PHP and Python. RATS was designed to detect buffer overflow vulnerabilities and race conditions. The database of C functions that RATS recognizes is much richer than those of Flawfinder and ITS4. Here is a summary of the number of functions recognized for each of the programming languages:

Entries in perl database: 33
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55

RATS uses a greedy pattern matching algorithm to find vulnerable functions in the source code, so a scan for printf will also match sprintf, vsnprintf, and print. This may create many false positives.

A typical RATS warning includes:

1. The name of the file.
2. The line number.
3. Risk level.
4. The name of the function.
5. Description of problem, often specifying what checks need to be made to ensure safety.

RATS has a capability to generate HTML reports. It is also possible to add some XML reporting features.

Here is the output of RATS for test.c:

Analyzing test.c
test.c:7: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated on the stack are used safely. They are prime targets for buffer overflow attacks.

test.c:9: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy more data than can be handled, resulting in a buffer overflow.

Total lines analyzed: 13
Total time 0.001527 seconds
8513 lines per second

Notice that RATS, like Flawfinder, gives you the total number of lines analyzed and the total time elapsed. It also gives the effective speed of scanning (i.e. lines per second).

4.2 Semantic Analysis Tools

4.2.0 What is abstract interpretation?

Researchers have been studying static code analysis for several decades. A powerful technique known as abstract interpretation was pioneered in the mid-70s by Patrick Cousot et al [22]. Abstract interpretation has become the main approach for doing semantic code analysis, and all of the tools that are described in this section, with the exception of Splint, use some form of abstract interpretation/symbolic analysis. There are, however, significant differences between these tools, particularly in the types of algorithms that are used to solve constraints among variables. The underlying principles of abstract interpretation are as follows.

The goal of abstract interpretation is to simulate all paths of a program without ever executing any code. During the simulation of a program, "abstract values" are associated with the variables of the program, instead of concrete values that would be used during actual execution. An abstract value is defined as a set of concrete values. Thus, an abstract value assigned to a variable specifies a range of concrete values that the variable might take on. Probably the simplest example of abstract values would be integer ranges, or intervals, that indicate the range of values an integral valued variable might take on.

The main idea of abstract interpretation is to symbolically step through a program and for each variable keep track of all possible values that it could take on at

each point in the program. Each program construct (e.g. loop, conditional, assignment etc) modifies abstract values of the variables involved in a specific way. This is illustrated below through an example program. Imagine the following code which has been annotated for easy reference:

```
1.  void taint_add(int y){
2.  int x;

3.  scanf("%d", &x);

4.  if (x < 10)
5.  x = x + 1;
6.  else
7.  x = x – 1;

8.  return x + y;
9.  }


10. int main(){

11. int y, z;
12. char buf[] = "Hello World";

13. y = 2;
14. z = taint_add(y);

15. return buf[z];
16. }
```

For the sake of the example, we will choose to track only integer valued variables.

We begin by evaluating the program abstractly inside of main(). Upon entry into main, y and z are undefined and can take on any value between $-\infty$ and ∞, so their abstract values are $[-\infty, \infty]$. Next, on line 13, y gets assigned to 2, changing y's abstract value to [2, 2]. On line 14, z gets assigned taint_add(y). We now step inside taint_add(). Inside taint_add(), x is read in from the standard input and thus can take on any integer value. Thus, an abstract value of $[-\infty, \infty]$ is assigned to x following the scanf() statement. Now, we encounter an if-statement. We evaluate the two branches of the if-statement, and for each branch we update the abstract value for x. If $x < 10$, then upon exiting the if-statement, x will take on the abstract value $[-\infty, 10]$. On the other hand, if $x >= 10$ upon entry into the if-statement, then upon exiting the if-statement, x will take on an abstract value of $[9, \infty]$. Finally, depending on which branch of the if-statement was followed, the return value of taint_add() will take on the abstract value $[-\infty, 10+y]$ or $[9+y, \infty]$. In our case, since the abstract value of y was [2, 2], the two possible return values are $[-\infty, 12]$ and $[11, \infty]$. So, getting back to the main() routine, z can have two abstract values [-

∞, 12] or [11, ∞]. The variable z is used as an index into the array buf which consists of 12 characters, including the null terminator. Thus, the minimum allowable value for z can be 0 and the maximum value can be 11. At this point, our abstract interpreter would raise a flag, since it is possible to index buf with an illegal index, i.e. z >=12 or z<0. This example illustrates the power of abstract interpretation to analyze programs. As was shown, all possible paths of the program were analyzed without running the program on specific inputs. Abstract analysis can be extended to deal with many other programming constructs, such as loops and switch statements. Abstract values can be defined for other types of variables besides integral valued variables.

We now look at the specific approaches taken on by our suite of static semantic analysis tools.

4.2.1 Splint (Secure Programming Lint)

LCLint, the predecessor of Splint, was developed by David Evans et al of University of Virginia. LCLint was enhanced to deal with security related software bugs, and in 2002, this enhanced version of LCLint became known as Splint. Splint finds programming errors such as missing arguments, undeclared variables, allocated memory not being freed, improper return statements, nested comments etc. A large fraction of warnings that Splint issues are not security related at all. However, Splint can be used to find buffer overflows and format string vulnerabilities as well.

Splint, unlike the lexical tools just discussed, works on the semantic level. The creators of Splint describe their approach as a combination of several "lightweight static analysis" techniques, so for lack of a better name we will use this descriptor as well. To be truly useful, Splint requires the programmer to make annotations, or semantic comments in the source code. These annotations specify certain pre-conditions and post-conditions that may apply to function parameters, function return values, as well as global variables and structure fields. For example, one may declare a function parameter to be not null using the annotation /*@notnull@*/. At any call-site in the program where the function parameter might become null, Splint will report a warning. Similarly, it is possible to declare return values and global variables as being not null. Splint has its own version of annotated standard C library function headers. For instance, the Splint library declares gets() with the annotation /*@warn bufferoverflowhigh "Use of gets leads to a buffer overflow vulnerability... "@*/. Hence, whenever calls to gets() are encountered, Splint raises a buffer overflow warning.

To help catch array bounds violations, Splint uses two clauses, "requires" and "ensures", and four buffer attribute annotations, maxSet, maxRead, minSet, and minRead.

Here is a simplified example of how Splint annotates the string function strcpy() inside standard.h:

```
char *strcpy(char *s1, char *s2)
    /*@requires maxSet(s1) >= maxRead(s2) @*/
    /*@ensures maxRead(s1) == maxRead(s2) @*/
```

maxRead(b) gives the value of the highest index i such that b[i] can be read. maxSet(b) gives the value of the highest index i such that b[i] can be set to some value. minRead and minSet are defined similarly. Before each function call, Splint checks that all the pre-conditions are met, i.e. the "requires" clauses are satisfied, and that the post-conditions are ensured. Constraints for variables, functions etc., are generated at the expression level and are stored in the corresponding node in the parse tree. Some constraints get generated automatically. For instance, the declaration "char buf[MAXSIZE]" generates the post-conditions "maxSet(buf) = MAXSIZE – 1" and "minSet(buf) = 0". If Splint later encounters an expression of the form "buf[i] = x", it will generate pre-conditions "maxSet(buf) >= i" and "minSet(buf) <= i" and then try to verify that these pre-conditions hold true. If Splint is unable to verify that i <= MAXSIZE – 1 and i >= 0, it will issue a warning. Splint also uses certain algebraic rules involving pointer manipulations, such as maxSet(ptr + i) = maxSet(ptr) – i.

To some extent, Splint is able to track tainted variables, or variables that might have values obtained from an outside source, such as the standard input, a file, a network packet etc. To track tainted variables, Splint annotates several standard library functions inside tainted.xh. For example, to say that printf() expects an untainted parameter, Splint annotates printf() as "extern int printf (/*@untainted@*/ char *format, ...) ;". Using taint analysis it is possible to detect certain format string vulnerabilities, i.e. where the format string passed to a printf family function comes from an outside environment.

To deal with control flow constructs such as loops (e.g. for-loops and while-loops), Splint uses certain heuristics. Unfortunately, Splint can analyze only the most common loop constructs. David Evans et al assume that the programmer will follow stylized, common loop conventions, which does not seem like a very safe assumption to make. Programmers are likely to deviate from conventional programming styles. One interesting assumption that Evans makes is that any buffer overflow that occurs inside a loop will be apparent either in the first or the last iteration. Thus, to assure loop safety, Splint only needs to verify that all the constraints are satisfied in the first and the last iterations of the loop. If the initial values for the loop variables, such as the loop index, are known and the final values are known or are easily computable, Splint can carry out a safety analysis for the loop. For instance, given a loop of the form "for (init; *buf; buf++)", Splint assumes that the loop will get executed for maxRead(buf) iterations. Now, if some variable inside the loop gets incremented by one during each iteration, Splint will estimate that variable's final value to be init_value + #iterations, or init_value + maxRead(buf).

When analyzing source code, it is suggested that Splint be run iteratively several times, each time either changing the code, i.e. fixing a bug, or making an annotation. Splint is able to check about 1000 loc per second, so running it several times does not impose a huge time penalty. Finally, it is important to note that Splint will not be able to detect any sort of inter-procedural errors, unless annotations are used. The goal of our evaluation is to test tools that can find bugs in legacy software through static analysis without having to modify the source code. Thus, for our evaluation we decided to opt against annotating source code.

4.2.2 BOON

BOON was developed by David Wagner as part of his PhD work and was publicly released in July 2002 (BOON stands for "Buffer Overrun detectiON"). Unfortunately, BOON is only a "working prototype" and not a fully supported tool. It is not very well documented and might contain bugs. The basic assumption that Wagner made was that most buffer overflows occur in string buffers. Consequently, the focus of BOON is on string buffers.

BOON treats a character string, S, as a pair of integers: the size allocated, alloc(S), and the actual number of bytes currently used, len(S). BOON models all string functions in terms of their effects on these two properties of string buffers. For each string declaration or usage of a string function, an integer range constraint is created via pattern matching. As different execution paths are analyzed, BOON generates two ranges, one for alloc(buf) and one for len(buf). Each of the ranges represents a conglomeration of all the possible values that alloc(buf) or len(buf) can ever take on inside the program.

To illustrate this concept, suppose that at the end of the analysis BOON concludes that len(buf) and alloc(buf) take on values only in [a,b] and [c,d], respectively [83]. There are three possibilities at this point:

1) If b<=c, we can safely conclude that buf never gets overflowed.

2) If a>d, we can conclude that buf always gets overflowed.

3) If the two ranges overlap, buf might get overflowed or it might not. BOON conservatively issues a warning at this point.

Here is an example of a bug that BOON would detect:

```
int main(){
  char bufA[10];
  char bufB[4];

  strcpy(bufA, "Hello");
  /* BAD */
  strcpy(bufB, bufA);
  return 0;
}
```

BOON issues the following warning:

```
Almost certainly a buffer overflow in `bufB@main()':
  4..4 bytes allocated, 6..6 bytes used.
  <- siz(bufB@main())
  <- len(bufB@main()) <- len(bufA@main())
```

Unfortunately, the fact that BOON is flow-insensitive (i.e. ignores order of statements and control flow statements such as if-statements, loops, switch statements) causes a high rate of false positives.

Here is an example of a program that would cause BOON to false alarm:

```
#include <stdio.h>
int main(){

  char *buf;
  int i;

  scanf("%d", &i);

  if (i < 0)
    buf = "Hi";
  else
    buf = "Bob";

  printf("buf = %s\n", buf);

  return 0;
}
```

In the above program, if i < 0, len(buf) = 3, whereas if i >= 0, len(buf) = 4. BOON will conglomerate these two values together into a range, namely the closed interval [3, 4].

Similarly, if i<0, alloc(buf) = 3, and if i >= 0, alloc(buf) = 4. BOON will generate the range [3, 4] for alloc(buf). All string variables, S, must satisfy the constraint "len(S) <= alloc(S)". In this particular case, since the two ranges for len(S) and alloc(S) overlap, BOON is unable to assure string safety and issues the following warning:

```
Slight chance of a buffer overflow in `buf@main()':
  3..4 bytes allocated, 3..4 bytes used.
  <- siz(buf@main())
  <- len(buf@main())
```

It should be noted that BOON is capable of doing a limited amount of inter-procedural analysis. On the down side, BOON does not handle function pointers and doubly referenced arrays, such as command line arguments. BOON has the capability to detect very simple pointer aliasing bugs, and it deals with C structures to some extent. All C structures of a compatible type are treated as potentially aliased. These capabilities will be explored in greater depth in the Methodology and Results chapters.

4.2.3 ARCHER (Array CHeckER)

This is a very new tool, actually still in the development stage, created by a group of researchers (Xie, Chou, Engler) at Stanford University. ARCHER is an inter-procedural, flow-sensitive, context-sensitive, fully symbolic tool for detecting memory access errors. It has found many security holes such as buffer overruns. Unlike Splint/LCLint, ARCHER does not require annotations. ARCHER is able to exhaustively analyze all possible execution paths, and at the same time achieve good performance.

ARCHER uses bottom-up inter-procedural analysis. After parsing the source code into abstract syntax trees, ARCHER records caller-callee relationships between functions. This information is later used to construct an approximate callgraph which is used for determining the order for examining functions. It is only an approximate callgraph because ARCHER does not follow function pointers. This approximation leads to some imprecision, i.e. missed errors, but as the authors claim, it does not increase the number of false positives. ARCHER simulates the body of the C function and creates so-called symbolic triggers, or conditions on the function parameters that result in memory access violations (i.e. bad array indexing). These triggers are used to deduce new triggers for the callers, and so on. Once the top-most caller is reached, if any of its triggers are satisfied, a memory violation flag is raised.

The following is an example taken from the original ARCHER paper. It illustrates ARCHER's ability to do inter-procedural analysis.

```
1.  int aa(int flag, int *buf, int i, int j)
2.  {
3.   if (flag)
4.     return buf[i];
5.  if ((flag & 0xf0) > 0)
6.     return buf[j];
7.   return 0;
8.  }
9.
10. int bb(int *buf)
11. {
12.   aa(1, buf, 5, 5);
13. }

14. int cc(void)
15. {
16.   int *buf;
17.
18.   buf = (int *) malloc(sizeof(int) * 6);
19.  bb(buf + 1);
20.  }
```

ARCHER begins in a bottom-up manner by analyzing the code for aa() first. The first buffer access it sees is on line 4. This access only occurs if flag is not equal to 0. ARCHER generates two triggers for line 4, that look something like:

(flag $\neq$ 0) and (f(i) < 0)

(flag $\neq$ 0) and (f(i+1) > length(buf))

where f(i) = offset(buf) + i*sizeof(int).

These two triggers indicate two scenarios in which an out-of-bounds buffer access would occur. If either of these two triggers gets satisfied at some point, an error is reported. To continue our example, no triggers are added for the buffer access on line 6 because ARCHER does not handle non-linear conditions, i.e. flag & 0xf0. Next, ARCHER goes up one level in the callgraph to bb(). Inside bb(), the call to aa() causes aa's two triggers to be invoked and evaluated using the parameters. Thus, since flag = 1, the two triggers get simplified to:

f(5) < 0

f(5+1) > length(buf)

where again, f(i) = offset(buf) + i*sizeof(int).

These two triggers become the triggers for bb(), and ARCHER finally reaches the top of the callgraph inside cc(). Once inside cc(), ARCHER sees the allocation of buf on line 18, and deduces that length(buf) = 6 * sizeof(int), and offset(buf) = 0. Finally, when bb()

is called on line 19, the triggers for bb() are loaded and evaluated with offset(buf) = sizeof(int). The trigger "f(6) > length(buf)" is satisfied, and so ARCHER issues a buffer overflow warning. This example illustrates that ARCHER is able to detect fairly complex inter-procedural bugs.

The creators of ARCHER have made an interesting assumption; if an array is accessed without any bounds checking whatsoever, it is assumed that the programmer has some knowledge of the safety of such an array access. Thus, ARCHER ignores such unchecked array accesses. On the other hand, if some bound checking is attempted, ARCHER will perform analysis to ensure that the array accesses are indeed safe. The latter assumption might cause ARCHER to find complicated bugs, yet miss certain trivial bugs.

4.2.4 PREfix - not included in evaluation

PREfix is a sophisticated inter-procedural static analysis tool, known as a model checker, which was designed to be run on large code bases. As a matter of fact, it is used internally by Microsoft to test its software. Rumor has it that every couple weeks or so Microsoft runs PREfix on the entire Windows code base to detect potential buffer overflows and other bugs. Unfortunately, PREfix is not available to the general public. Although PREfix could not be included in this evaluation, it sounds like a promising static analysis tool and thus, it is briefly discussed here. The technical name for the type of approach employed by PREfix is "heuristic model checking". Let us see exactly what that means...

To begin, a model must be constructed for each function. For each function, different possible execution paths within the function are analyzed (the maximum number of paths to be analyzed can be specified by the user). A path is analyzed by traversing the function's abstract syntax tree and evaluating the relevant expressions and statements in the tree. For each execution path, PREfix uses a virtual memory model to track the values of the variables and the memory use. PREfix is quite good at following conditional control flow constructs (e.g. if, goto, switch). It is also able to prune out impossible paths based on its knowledge of the program state. Following the analysis of the different paths, the results are summarized and are used to build a single model for the behavior of the function.

A model for a function can be described by certain constraints, or pre-conditions, and results, or post-conditions. Similar to ARCHER, PREfix uses a bottom-up approach to generate the constraints for each function. To illustrate this, suppose function a() is called by b(), which is called by c(). PREfix will first generate constraints on a(), then constraints on b(), and finally constraints on c(). The building of a model for a function also relies on the concept of guards. A set of guards is simply a set of conditions that must be true in order for a function to result in a certain outcome. As a simple example, take a look at the function below [15]:

```
int null_ptr_test (int *p)
{
if (p == NULL)
  return NULL;
return *p;
}
```

This particular function null_ptr_test() has two outcomes. The first outcome has the result (post-condition) of returning NULL. The second outcome has the result of returning a value equal to the one in memory pointed to by p. The guard for the first outcome is the condition "p = NULL" and the guard for the second outcome is the condition "p ≠ NULL". For the first outcome, the only precondition is that p is initialized. For the second outcome, there are three preconditions – 1.) p must be initialized, 2.) p must be a valid pointer, 3.) The memory pointed to by p must be initialized. Without guards it would be impossible to separate the NULL case from the valid pointer case.

By creating models for each function, PREfix is able to carry out an effective inter-procedural analysis of the whole program. To analyze the entire program, PREfix begins with the leaf functions of the call graph and proceeds bottom-up from the root. PREfix simulates each function and reports any detected defects. To simulate a function, PREfix first fetches its model. All the guards for the model are evaluated to determine all possible outcomes for the function. In order for an outcome to be possible, all of its guards must be true. Once all possible outcomes have been enumerated, PREfix begins by selecting an eligible outcome. Notice that more than one outcome will be possible when some of the guards are unknown. Once an eligible outcome has been selected, any unknown guards are filled in with assumed values. Following this, all pre-conditions are tested. If any of the pre-conditions cannot be ascertained to be met, an error is reported. All eligible function outcomes are tried. As the analysis proceeds up the call graph, models for functions that have already been simulated are available for simulation of any callers higher up in the call graph.

PREfix is path-sensitive, incomplete (due to the restriction on the number of paths analyzed), and unsound (due to approximations). Unlike its sister tool PREfast, PREfix was designed to be run on large code bases (e.g. millions of lines of code). According to Bush, Pincus, and Sielaff, on the order of 90% of errors (invalid pointer references, bad storage allocation, using uninitialized memory, bad file operations etc.) are caused by interaction of multiple functions. It is exactly for that reason that PREfix places a strong emphasis on inter-procedural simulation. PREfix tries to cut down on false positives by not reporting errors when it is unsure. Sometimes PREfix reports several defects for a single underlying cause. PREfix is good at dealing with very large and complex code bases. To summarize, here is a quote from the creators of PREfix, "Complexity is managed by using adjustable thresholds on path coverage, merging

equivalent outcomes in models of called functions, employing heuristics to choose paths well, and using lazy techniques to trim paths only when necessary"[15].

4.2.5 PREfast - not included in evaluation

PREfast should not be confused with its sister tool PREfix [15]. PREfast comes prepackaged with the Microsoft Windows 2003 Driver Development Kit. PREfast, unlike PREfix, works only within procedures and operates on single file programs. PREfast was meant to be a fast and light weight tool, hence the name. PREfast was designed to be used iteratively during the process of writing the code, rather than being used on finished programs. When trying to run PREfast on a large piece of code, it is recommended that one split up the source code into smaller sections (less than 10MB each) and then run the tool on each section separately.

One of the main goals of PREfast is to detect memory violations such as buffer overflows. For each function in the program, PREfast analyzes all the possible execution paths (unless it can eliminate certain paths using knowledge about some local state). Thus, PREfast is said to be path-sensitive. PREfast can detect paths where a variable is used without being initialized, and it can also detect paths that leave a variable uninitialized. On the down side, PREfast does not keep track of global state; as a result, many false alarms may result from this weakness. For the same reason, PREfast is unable to prune out certain impossible execution paths that depend on global variables.

In addition to buffer overflows, PREfast can catch uses of uninitialized memory, de-referencing of NULL pointers, memory leaks, format string bugs, illegal type casts, ill-defined loops(i.e. infinite loops that should probably be finite), dead code etc.

One thing to note about PREfast is that it has a nice XML-based graphical interface for displaying the results of the test run. The tool allows one to navigate the source code through hyperlinks, and it even traces the whole execution path leading to the error.

4.2.6 PolySpace C Verifier

The C Verifier is a static analysis tool that attempts to do sophisticated abstract interpretation. The company that developed this tool, PolySpace Technologies, was founded by one of the pioneers of abstract interpretation, Patrick Cousot. PolySpace Technologies has targeted its static analysis tool towards the embedded application industry. As a result, the class of errors they focus on is slightly different from the classes of errors that are targeted by the other tools discussed so far. For instance, catching errors such as division by zero, sqrt(negative number), de-rereferencing of null or out-of bounds pointers, access conflicts for shared memory, reading of uninitialized variables, and illegal type casts is a very important part of C Verifier's analysis. In

addition, catching never-ending loops and unreachable, or dead, code is also a part of C Verifier's repertoire. Of course, catching buffer overflows is an important aspect of C Verifier's analysis as well, but it is not the primary focus. Bugs that result simply in the application halting are very dangerous in the embedded application industry, never mind remotely exploitable buffer overflows. Just imagine what would happen if a critical sensor in an airplane stopped working because of a divide by zero bug! The results could be catastrophic!

Perhaps the major difference between PolySpace's C Verifier and ARCHER, PREfast, Uno and BOON is that PolySpace claims to do a lot more tracking of variable relationships during program simulation. Being able to track relationships between two or more variables is necessary in order to detect such bugs as divide by zero. For instance, suppose there exists an assignment inside a function that looks like w = x/(y – z). The question that PolySpace attempts to answer is "Can y – z ever equal 0?" If the relationship between y and z is tracked throughout the program, PolySpace can determine whether or not it is possible for y to equal z. Tracking relationships between many variables is very complicated; the analysis is very memory intensive and takes a significant amount of time. It is possible to run the C Verifier in one of several modes, 0, 1, or 2, depending on the depth of analysis desired.

4.2.7 Uno

Uno is a relatively young static analysis tool developed by Gerard J. Holzmann of Bell Laboratories [43]. The name Uno derives from the three software defects that the tool targets: the use of **U**ninitialized variables, dereferencing **N**il-pointers, and **O**ut-of-bound array indexing. Uno is a model checker, similar to the tool PREfix. In addition to being used to detect the latter defects, Uno can be extended to test software for specific user-defined, application-dependent properties. User-defined properties are written as ANSI-C functions and make use of a small library of primitives that allow access to program's dataflow information. An example of a user-defined extension could be a function that ascertains that the particular program obeys certain locking rules. A simple locking rule might say that if a function calls a lock function (to disable interrupts), it must later call an unlock function (to enable interrupts) within the same function before returning. Failure to properly lock or unlock code during execution might pose a serious security risk! Another example of a security property that a programmer might want to verify in a program is the proper handling of process privileges. For instance, if a program raises the process privileges to that of a super-user, it should drop the super-user privileges when they are no longer needed.

Uno is an extension of the public-domain compiler front-end tool known as *ctree*. Ctree generates a parse tree for each procedure in the program, and Uno converts the parse tree to a control-flow graph for the procedure. The control-flow graphs are used by Uno to perform local intra-procedural analysis. A special dataflow analysis module is used to keep track of the state of all the data objects in the program. The dataflow module marks each data object with a special tag that specifies whether or not the object

has been declared, initialized, invoked as a function pointer, evaluated as a variable, dereferenced, assigned a new value, or if its address has been taken. For the purpose of detecting out-of-bounds errors, the dataflow module collects information about each array, such as its known bounds and information about variables that are used to index the array.

The basic operation of Uno can be described as a two-pass process. During the first pass, Uno analyzes each source file separately and performs local analysis of each function. During this stage, Uno checks the usage of local function variables and statically declared global variables. Uno uses the technique of abstract interpretation to keep track of possible value ranges for each variable at each point in the program. Different execution paths are analyzed and if possible, infeasible paths are pruned. Consider the following simple example. Suppose that during the execution of a particular path the assignment "x=7;" gets encountered and later on an "if (x > 10)" branch is encountered. In this scenario, Uno would recognize that the path "...x=7; ...x>10" is impossible. Put simply, Uno uses the context of the execution path analyzed so far to determine where the path can lead in the future. Uno also uses intermediate files to save any information that it thinks might be useful later on during the global analysis stage. For each function, Uno constructs and saves a list of functions that can be called from that function. This information is used in the second phase to model the inter-procedural behavior of the program.

Following the first pass, each function has a corresponding control flow graph. In the second pass, Uno performs global analysis based on the information it has collected and stored in the intermediate files. In this phase, Uno checks the usage of any non-static global variables, focusing on any pointer dereferencing operations. Also, based on the information obtained in the first pass, Uno can now construct a function call-graph and abstractly simulate the entire program. Uno starts with the main () routine and recursively works its way to all the other functions that can be called, via a depth-first search. If at any point during the simulation an inconsistency is detected, Uno reports an error.

4.2.8 MC (Meta Compiler) – Not included in evaluation

The MC tool was developed by Dawson Engler et al, at Stanford University. This tool allows a user to write lightweight compiler extensions to ensure that specific security rules are followed. For instance, an array should not be indexed without first doing a bounds check on the index. Similarly, a string copy should not be performed without first checking that the source string does not exceed the length of the destination buffer. Thus, the meta-compiler could potentially be a good tool for detecting buffer overflow vulnerabilities. The basic operation of MC can be described as follows.

The user provides a list of "untrusted sources" and "trusted sinks." Untrusted sources include functions that have arguments coming from untrusted environments, i.e., the network, user specified arguments etc (they include routines such as copyin ( ),

copyinstr ( )). They also include system calls. The list of trusted sinks includes array accesses (e.g. a[x]), loops (e.g. while(i < x)), copy routines (e.g. memcpy (p, q, x), bcopy (p, q, x)) that take length arguments that could be tainted, routines that perform user-kernel copy operations (e.g. copyout (k, u, x), copyin (u, k, x)), and memory allocation routines (e.g. malloc (x), kmalloc (x)). The user writes an FSM description for each particular rule in a language known as METAL. The states of the FSM are typically bound to variables or expressions. Any variable coming from an untrusted source will be assigned a tainted state, and the MC will check to see if the variable is ever properly sanitized. If the variable is correctly sanitized, MC will enter a stop state and the variable will no longer be tracked.

Engler et al used MC to implement a so called range-checker, which was geared for detecting out-of-bounds array accesses. The range checker starts by assigning a state called "tainted" to an untrusted variable x, and if it ever sees a check of the form "x < something", it changes x's state to "need_lb", (need lower bound). If the range checker later encounters a check of the form "something < x", x is no longer tracked, and the particular thread tracking x goes into a stop state. Similarly, if the range-checker had first encountered a check of the form "something < x", x would have transitioned from the "tainted" state into the "need_ub" (need upper bound) state. If a "tainted" variable ever reaches a trusted sink, a flag is raised. The user is free to define other patterns to be matched against, besides simple lower bound and upper bound checks.

One difficulty encountered by the designers was being able to differentiate between incoming and outgoing network packets (only incoming packets should be checked). The heuristic used was: if the checker sees memory allocation involving packet structures (e.g. malloc ( )), then it is very likely that the code is building an outgoing packet. Otherwise the packet is assumed to be incoming.

MC uses both intra-procedural and inter-procedural analysis. Using inter-procedural analysis it tries to derive functions that could consume tainted data (i.e. trusted sinks) as a result of being called from known tainted functions, or functions that produce tainted data (untrusted sources) as a result of containing a call to a known tainted function. Also, MC tries to derive any routines whose arguments could reach a trusting sink, in which case, it will track these arguments. If any of these arguments could become tainted, a flag will be raised.

Last but not least, MC follows around function pointers. If a tainted value is passed to a function pointer, MC checks whether the function pointer can possibly point to a function whose arguments can reach a trusting sink.

4.3 Summary of Static Analysis Tools

The table below summarizes the "claimed" analysis capabilities of the discussed static analysis tools.

Tool Name	Availability	Analysis Strategy				Analysis Capabilities			
		Lexical (Grep-like)	Lightwt. Static Analysis with Annot.	Abstract/ Symbolic Interpretation	Lightweight Compiler Extensions	Control Flow	Inter-procedural	Taint	String Functions
Flawfinder	Open-source	*							*
ITS4	Open-source	*							*
RATS	Open-source	*							*
Splint	Open-source		*				* with src. annot.	* with src. annot.	*
MC	Research				*	*	*	*	
BOON	Open-source			*			*		*
Prefix	Internal use at Microsoft			*		*	*		*
PREfast	Commercial			*		*			*
ARCHER	Research/ Proprietary			*		*	*	*	
UNO	Open-source			*		*	*		
C Verifier	PolySpace (commercial)			*		*	*		*

Table 11. Claimed capabilities of static analysis tools.

Chapter 5 Evaluating Static Analysis Tools - Methodology

5.0 Buffer Overflow Classification Scheme

Buffer overflow vulnerabilities in software are like disease-causing microbes. Just like there are many different kinds of microbes, there are many different kinds of bugs that lead to buffer overflows and it would be nice if one could classify them. By being able to uniquely identify each buffer overflow vulnerablity, one can get a better understanding of what the distribution of buffer overflow vulnerabilities is like in real programs. We would like to be able to answer questions like: "What fraction of buffer overflows in open-source programs involves buffers on the stack, the heap, the bss region and the data region? What fraction of buffer overflows involve pointers being passed through several procedure calls? What kinds of buffers tend to get overflowed the most; are they character buffers, integer buffers etc? Are the buffers self-contained or are they members of C structures or unions? And so on." Only once we have an idea of what the true distribution of buffer overflows is like in real programs, can we say which static analysis tools would work best on real programs. For instance, if it turns out that most buffer overflow vulnerabilities involve inter-procedural calls, then tools that only do intra-procedural analysis would not be too useful for detecting such bugs. This section outlines a classification scheme that we developed for classifying the different types of software bugs that lead to buffer overflows. The classification scheme is based on thirteen fields, each field being represented by a single hexadecimal digit. We now describe these fields.

digit 1: WRITE/READ

Is this an illegal write or an illegal read?

0 write	e.g buf[i] = 'A';
1 read	e.g. c = buf[1000];

digit 2: WHICH BOUND

Which bound gets violated, the upper or lower?

0 upper bound	e.g. buf[sizeof(buf)] = 'A';
1 lower bound	e.g. buf[-1] = 'A';

digit 3: TYPE

What type of data is stored in the buffer?

0 char
1 int
2 float
3 wchar
4 pointer (e.g. char *buf[i])
5 unsigned int
6 u_char
7 signed char
8 guint8
9 gchar
A double
...

digit 4: WHERE

Where in memory is the buffer that gets overflowed?

0 on the stack
1 on the heap
2 in data region (declared static, initialized data)
3 in bss data (declared static, uninitialized data)
4 in shared memory

digit 5: SCOPE

Where is the buffer allocated and where is it overrun?

0 allocated in main, overrun in main
1 allocated as global, overrun in main or function
2 allocated in main, overrun in function in same file (inter-procedural)
3 allocated in a function, overrun in same function
4 allocated externally, overrun in different file (inter-file)
5 allocated in one func, overrun in same file, different func (inter-procedural)

digit 6: CONTAINER

Is this buffer inside of a container?

0 no, self contained
1 yes, an array of buffers

2 yes, a struct containing a buffer
3 yes, a union containing a buffer
4 yes, an array of structs containing buffers

digit 7: INDEX/LIMIT COMPUTATION

How is the integer index
... or length provided to str*/mem*/... function)
... or limit for accessing "for" loop,
computed?

It is

0 a constant	e.g. a[2] , *(a+2)
1 a variable	e.g. a[i]
2 a linear expression	e.g. a[5*i+2]
3 a nonlinear expression	e.g. a[i%3], a[(unsigned char) i], a[i*i]
4 the return value of a function	e.g. a[random()]
5 the contents of an integer array	e.g. a[b[i]], a[*b]
6 none	e.g. *p++

digit 8: ACCESS METHOD

How is this buffer read/written?

0 via an index e.g. a[i] = 'a'
1 via a pointer e.g. *p = 'a'
2 via a function e.g. strncpy (a,b,3)
3 mixed, i.e one branch of if-statement overflows buffer
through function and other through pointer
4 via doubly dereferenced pointer e.g. *(*p) = 'a'
5 standard C macro (e.g. PUTSHORT('A', ptr))

digit 9: ALIAS

Is the buffer accessed through an alias or an alias to an alias?

0 no aliasing, access is through the original
1 yes, it is accessed through an alias to the original e.g. ptr = buf; *ptr = 'a';
2 yes, it is accessed through an alias to an alias to the original

digit 10: CONTROL FLOW

Is the overflow in some way obfuscated by program control flow, i.e., does the buffer overflow occur inside an if-statement block, a switch case, conditional statement etc?

0 no
1 yes, through an if statement
2 yes, through a switch statement
3 yes, through a cond statement e.g. (i>4) ? i++; i--;
4 yes, through a goto statement
5 yes, through a setjmp statement
6 yes, through a function pointer
7 yes, through recursion
8 yes, through (shudder) mutual recursion

digit 11: LOOPS

Does the overflow happen within a loop construct?

0 no, no loop
1 yes, in a for loop
2 yes, in a do-while loop
3 yes, in a while loop
4 yes, nested loops (i.e while loop inside a for loop)

digit 12: ASYNCHRONY

Is the buffer overflow obscured by some asynchronous program construct?

0 no
1 yes, via threads API (analysis tool doesn't understand API)
2 yes, via a forked process API
3 yes, via a signal handler API
4 yes, via threads data asynchrony
5 yes, via process data asynchrony
6 yes, via signal handler data asynchrony

digit 13: TAINT

Is the buffer overflow in some way influenceable externally?

0 no
1 yes, through argc/argv
2 yes, through environment variables, getenv(name)
3 yes, by reading from a file
4 yes, through packet data (i.e., read from socket)
5 yes, through calls such as getcwd, pwd etc.

5.1 Simple Test Cases

Once a classification scheme had been established, a set of approximately 50 pairs of test cases was created to try to evaluate the strenghts and the weaknesses of the different semantic analysis tools. The test cases were simple programs, usually not spanning more than twenty lines of code. The test cases were written in pairs – a bad version of the program and a fixed version of the program. The test cases were designed to try to cover all thirteen axes of our classification scheme. However, we did not create an exhaustive list of test cases covering all possible combinations of the thirteen fields; for the most part, each test case included a bug for which most of its fields were the default value and one or two fields deviated from the default. Take a look at the example programs below. These two test programs could be named something like: test-0001001000100-BAD.c and test- 0001001000100-OK.c.

```
/*BAD Version*/
#include <stdio.h>
#include <stdlib.h>

int main(){

 int i;
 char *buf = (char *) malloc(20);

 for(i = 0; i<21; i++){
  /*BAD*/
  buf[i] = 'a';
 }

 return 0;
}
```

```
/*OK Version*/
#include <stdio.h>
#include <stdlib.h>

int main(){

int i;
char *buf = (char *) malloc(20);

for(i = 0; i<20; i++){
/*OK*/
buf[i] = 'a';
}

return 0;
}
```

There were a few exceptions to the single deviation rule, as in the example programs shown below. Here two fields deviate from the default value, namely the location of the buffer and the type of buffer.

"BAD Version"

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

int main(){

 char **buf, *bufA = "Alice";
 int i;

 buf = (char **) malloc(2*sizeof(char *));

 for(i=0; i<3; i++){
  /*BAD*/
    buf[i] = bufA;
 }

 return 0;
}
```

"OK Version"

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

int main(){

char **buf, *bufA = "Alice";
int i;

buf = (char **) malloc(2*sizeof(char *));

for(i=0; i<2; i++){
/*OK*/
buf[i] = bufA;
}

return 0;
}
```

These two test cases could be named test-0041001000100-BAD.c and test-0041001000100-OK.c, respectively. Notice, that the thirteen fields of the fixed program are the same as those of the bad program; we don't care how the program was fixed.

Here is another program that illustrates the improper usage of a string function. What programmers sometimes forget is that strncat() automatically null terminates the destination buffer. Here the programmer has forgotten to leave space for the null byte, resulting in an off-by-one overflow.

Test-000000420000-BAD.c
String Functions, Off-by-one bug in strncat():

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){
  char buf[6] = "Hi";
  char bufB[] = " there";

  /*BAD*/
strncat(buf, bufB, sizeof(buf)-strlen(buf));

return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){
  char buf[6] = "Hi";
  char bufB[] = " there";

   /*OK*/
 strncat(buf, bufB, sizeof(buf)-strlen(buf)-1);

return 0;
}
```

The set of test cases that we created had a pretty good coverage of the different types of buffer overflows. The test cases included all possible locations of the overflowed buffer, i.e., the stack, the heap, the bss and the data regions, different scope scenarios, e.g., inter-procedural vs. intra-procedural, different control flow constructs, e.g. if statements, for loops, while loops, switches, and tainted and untainted cases.

To illustrate the complexity of analysis that static analysis tools need to carry out, we show two fairly simple programs that would give a lot of trouble to most of the semantic analysis tools.

In the following program, a global variable, num, is used as an index to a buffer. A call to change_index(void) has the side effect of changing the global variable. In order to catch the particular bug, a tool would need to be able to do dataflow analysis, control-flow analysis, handle global variables, and perform inter-procedural analysis.

Inter-procedural:

/*inspired by code segment from mutt-1.3.28/sendlib.c*/

```
#include <stdio.h>

static int num;
static char buf[4];

void change_index(){
  num = 4;
}

void write_to_buf (){
  if (num == 2)
    change_index();

  /*BAD*/
  buf[num++] = 'a';  /* illegal write here*/
}

int main() {

   num = 2;
   write_to_buf();

   return 0;
}
```

```
#include <stdio.h>

static int num;
static char buf[4];

void change_index(){
   num = 0;
}

void write_to_buf (){
   if (num == 2)
     change_index();

   /*OK*/
   buf[num++] = 'a';
}

int main() {

   num = 2;
   write_to_buf();

   return 0;
}
```

The following program is interesting because it involves tainting a buffer by reading from the command line, and using a nested control flow statement. An if-statement is used as a test for breaking out of a for-loop.

Tainting and Control Flow:

```
/* based on code from Ethereal 0.9.8: packet-rtsp.c:716 */

#include <stdio.h>
#include <string.h>

int main(){
 int i;
 char buf[8];

 scanf("%7s", buf);
  for(i=0; i<sizeof(buf); i++){
    if (buf[i] == 'N')
      break;
 }
/* BAD */
/* possible out-of-bounds write */
 buf[i] = 'M';

 return 0;

}
```

```
#include <stdio.h>
#include <string.h>

int main(){
  int i;
  char buf[8];

scanf("%7s", buf);
for(i=0; i<sizeof(buf); i++){
  if (buf[i] == 'N')
    break;
}
if (i == sizeof(buf))
  return 1;
/*OK*/
 buf[i] = 'M';

return 0;
}
```

Roughly 50 simple test programs were constructed as part of this thesis work. The test cases are being used by other researchers to analyze the performance of static code analysis tools, and the results will be presented elsewhere. The remainder of this thesis focuses on real buffer overflow vulnerabilities, rather than the simple test cases.

5.2 Real Vulnerabilities

To test the effectiveness of our suite of static semantic analysis tools, we chose to analyze three applications: BIND (Berkeley Internet Domain), WU-FTPD (Washington University FTPD), and Sendmail. BIND is the most popular software used for running DNS servers. WU-FTPD is a very popular FTP daemon, and Sendmail is currently the most popular mail transfer agent (MTA). All three of these applications have had several serious buffer overflow vulnerabilities publicized in the past several years. Fourteen severe buffer overflow vulnerabilities were selected for this study. Most of the vulnerabilities allowed a remote attacker to gain full control of the system running the vulnerable software and execute arbitrary code. The goal of this retrospective analysis was to see if any of the semantic analysis tools would have been able to detect any of these vulnerabilities.

I began by trying to run Splint on a vulnerable version of Sendmail, version 8.12.4. It soon became apparent that getting Splint to run on the entire version of Sendmail was not going to be easy. Splint issued many parse errors regarding specific type definitions such as u_char and u_long. Even though all of the types in question were defined either in Sendmail include files or standard C include files, Splint was not able to continue with analysis without user intervention. I obtained from David Evans, the creator of Splint, a page-long list of definitions that were required in order for Splint to run on Sendmail. Similar problems were encountered when I tried analyzing an old version of Sendmail using ARCHER. ARCHER was able to carry out some analysis of the code, but it terminated with a cryptic "divide by zero error". PolySpace's C Verifier did not fare much better. It took a PolySpace expert to come up with a script that contained all the necessary flags, and to create a list of all the necessary #define statements and include files, in order for C Verifier to be able to analyze the code for Sendmail 8.12.4. Seeing how difficult it was to get all of the tools to analyze large and complex programs like Sendmail (40K+ lines of code (LOC)), we decided that a better strategy would be to try to extract as much code from the real programs as possible, making sure to contain the buffer overflow vulnerabilities, and to create smaller, self-contained model programs.

Every possible attempt was made to try to preserve the general structure of the vulnerable code when creating these model programs. For instance, if the buffer overflow vulnerability involved cross-procedural calls, the model program had to preserve this structure. By extracting as much code as possible, the relative complexity of the model programs would remain close to the complexity of the real program. Any strange code that served to obfuscate the bug in the real program would obfuscate the bug in the smaller model program. Extracting code from BIND, WU-FTPD, and Sendmail was especially difficult when the vulnerability involved several procedure calls. On average five to seven hours were spent in the process of constructing each model program and getting it to compile with the -Wall and -pedantic gcc flags. Being able to compile the test program with these flags is strongly recommended, especially for testing programs with PolySpace's C Verifier. For each of the "BAD" model programs, i.e., programs containing the specific vulnerabilities, an "OK" version of the model program was also constructed, where the problems had been fixed.

Although being able to analyze the original programs would be ideal, the results of analyzing the model programs would be instructive as well; if a tool could detect any of the buffer overflow vulnerabilities in the model programs, there would be high hope that the tool could also detect these vulnerabilities in the original version of the program (after serious hand-holding and expert configuration). Another goal of this experiment was to see if the tools could differentiate between the "BAD" programs and the corresponding "OK" programs without raising many false alarms. Each of the fourteen vulnerabilities was classified using the scheme described in section 5.0. To see one of the model programs involving a recent Sendmail vulnerability refer to Appendix D.

5.3 Vulnerabilities in BIND (See Appendix C)

Vulnerabilities in BIND, the software that powers most DNS servers, are especially grave since the operation of the Internet relies on the proper functioning of DNS servers. By subverting a single DNS server, an attacker can compromise the integrity of the majority of websites on the Internet. Several of the buffer overflow vulnerabilities that we are about to discuss allow a remote attacker to gain full control of the vulnerable DNS server and execute arbitrary code on the machine running the DNS server. One of the vulnerabilities allows a remote attacker to crash the name server, thus resulting in a denial of service.

5.3.0 nslookupComplain vulnerability: CERT Advisory CA-2001-02

On January 29th, 2001, a CERT security advisory was published regarding a vulnerablity in versions of BIND prior to 4.9.8. This vulnerability involves a stack buffer overflow in a function called nslookupComplain().

Classification of buffer overflow: 0000306201004.

When a DNS server running BIND 4 receives a query with a hostname to be resolved into an IP address, the server first tries to resolve the name locally, by checking the local zone files and the local cache. If the name server is unable to resolve the hostname locally, it tries to delegate the job to another name server. BIND retrieves from a local database a list of name server records, or NS records, for name servers that are responsible for the hostname in question. Once the list of NS records has been retrieved, BIND will try to find out the IP address corresponding to each of the NS records in the list by calling nslookup(). The original name server will then use the list of IP addresses to query other name servers in order to try to resolve the original hostname. During the process of creating the list of IP addresses of the name servers, nslookup() checks each of the IP addresses it finds for validity. If the IP address is invalid, e.g. 0.0.0.0 or 255.255.255.255, nslookup() complains by calling a function called nslookupComplain(). nslookupComplain() is responsible for generating an error message to be logged to syslog.

While generating the error message, nslookupComplain() uses sprintf() to store the message into a fixed size buffer that resides on the stack. No size check is performed to see if the buffer is large enough to store the entire complaint. As a result, an attacker may be able to construct a long query that overflows the stack buffer and overwrites the return address of nslookupComplain() with the address of the attacker's shell code.

5.3.1 SIG-BUG: CERT Advisory CA-1999-14

This vulnerability was published in November 10, 1999 and it involves a buffer overflow because of improper handling of SIG records.

Classification of buffer overflow: 0060301212004

Several versions 4.x and 8.x of BIND are susceptible to an attack that causes the DNS server to crash resulting in denial of service. Name servers use SIG records as a means of authenticating themselves to other name servers and vice versa. The SIG records contain cryptographic signatures and key information that allow the recipient name server to verify that the sender is indeed who it claims to be. A function called rrextract() (where rr stands for resource record) is responsible for parsing SIG records, along with other types of records. This function contains a memcpy() call that does not properly validate the size argument.

Inside of rrextract() there is a switch statement that deals with SIG resource records. Inside the switch statement, the resource record that has been received from the network gets modified and gets stored into a data buffer, which gets referred to via a pointer called cp1. For the SIG record, the data that gets written to cp1 includes the signer's name and the signer's signature among other things. The vulnerable code looks something like this:

```
static int rrextract(...){

u_char data[MAXDATA*2];
....

switch(type){                   /* do stuff for each type of resource record */
....
case T_SIG: {
....
cp1 = (u_char *)data;
....
/* Expands the signer's name and puts it into cp1 */
/* n is the length of the compressed signer's name */
n = dn_expand(msg, eom, cp, (char *)cp1, (sizeof data) - 18);

if (n < 0)  {
hp->rcode = FORMERR;
return (-1);
}

cp += n;
cp1 += strlen((char*)cp1)+1;

 /* Finally, we copy over the variable-length signature into cp1.
    Its size is the total length of the received resource record (dlen), minus what
    we have copied so far, NS_SIG_SIGNER + n bytes..
*/

n = dlen - (NS_SIG_SIGNER + n);

if (n > (sizeof data) - (cp1 - (u_char *)data)) {
...
Exit with an error!
}
```

```
...
/* Finally copy the signature into cp1, which points to data[] */
memcpy(cp1, cp, n);
...
}
```

The problem in the above code is that the third argument to final memcpy call, n, can end up being negative, i.e. if dlen < NS_SIG_SIGNER + n. If n comes out negative, then the if-statement before the memcpy will get skipped (unless the compiler casts n to an unsigned integer during the comparison). The third argument of memcpy always gets cast to an unsigned integer, so a negative n will get interpreted as a very large positive number, approximately 4GB on a 32 bit architecture. The memcpy will end up writing close to 4 billion bytes to the data buffer and will cause the DNS server to crash!

5.3.2 NXT-BUG: CERT Advisory CA-1999-14

This vulnerability was published as a CERT advisory on November 10th, 1999, along with the SIG-BUG vulnerability. This vulnerability involves a buffer overflow in the code handling NXT resource records, and can allow a remote attacker to execute arbitrary code running with the privileges of the target DNS server, usually root. BIND version 8.2 is vulnerable.

Classification of buffer overflow:0060301212004

A malformed response from one DNS server to another, containing a NXT record, can trigger a buffer overflow in the buffer designed to store the response. The overflow is similar to the one described in the SIG-BUG discussion. The function for parsing NXT record responses is also rrextract(), and the NXT case is one of the cases of a switch statement, like the SIG case. Here is what the case for NXT records looks like:

```
1. case T_NXT:
2.    n = dn_expand(msg, eom, cp, (char *)data, sizeof data);
3.    if (n < 0) {
4.       hp->rcode = FORMERR;
5.       return (-1);
6.    }
7.    if (!ns_nameok((char *)data, class, NULL, response_trans,
8.           domain_ctx, dname, from.sin_addr)) {
9.       hp->rcode = FORMERR;
10.      return (-1);
11.   }
12.   cp += n;
13.   cp1 = data + strlen((char *)data) + 1;
14.   memcpy(cp1, cp, dlen - n);
15.
16.   cp += (dlen - n);
17.   cp1 += (dlen - n);
18.
19.   /* compute size of data */
20.   n = cp1 - (u_char *)data;
21.   cp1 = (u_char *)data;
```

22. break;

The overflow occurs in the memcpy on line 14. The problem here is that the program never checks to see if dlen – n bytes fit into the data buffer, i.e., the buffer pointed to by cp1. Dlen is the length of the received resource record as specified in the packet header. An attacker can create a fake DNS packet reply that contains a large fake dlen. This would cause an overflow in data[]. This overflow can be used to mount a stack smashing attack resulting in root compromise of the DNS server. If dlen – n is negative, that quantity would get cast to a very large positive number, and data[] buffer would get overflowed with approx. 4GB, resulting in a crash, i.e., denial of service.

Here is how an attacker could carry out an attack. Suppose that there is a master name server that controls some domain Y.com. An attacker first secretly inserts an NS record, N, for a fake subdomain X.Y.com into the master DNS server's database, such that the IP address of the name server in the NS record is the IP address of an attacker-owned machine. Then the attacker queries any desired target DNS server, using a tool such as nslookup, for some fake hostname Z in the fake domain X.Y.com, i.e., Z.X.Y.com. Since Y.com falls under the authority of the hacked master name server, the target name server queries the master DNS server to try to resolve the subdomain X.Y.com. The master DNS server cannot resolve the subdomain locally and retrieves the malicious NS record, N, and the query gets referred to the attacker-owned "DNS server". The attacker-owned "DNS server" is simply a machine that is patiently listening for queries on the well-known DNS port, 53, and upon receiving a query, it automatically delivers an exploit code to the target DNS server in the form of a fake NXT record reply.

5.3.3 IQUERY-BUG CERT Advisory CA-98.05, CVE-1999-0009

This advisory was published on April 8th, 1998. Versions of BIND 4.9 prior to 4.9.7 and BIND 8 releases prior to 8.1.2 were found to be vulnerable to a buffer overflow resulting from improperly bounds checking a memcpy() call when responding to inverse query requests.

Classification of buffer overflow: 0000301200004

A remote attacker can cleverly craft an inverse query request on a TCP stream and either crash a DNS server or execute arbitrary code.

Let's take a look at the code that handles inverse query requests:

```
static enum req_action
req_iquery(HEADER *hp, u_char **cpp, u_char *com, int *buflenp,
           u_char *msg, struct sockaddr_in from)
{
        int dlen, alen, n, type, class, count;
        char dnbuf[MAXDNAME], anbuf[PACKETSZ], *data, *fname;

        /* read in domain name, type, class, ttl...*/
        GETSHORT(dlen, *cpp); /* get dlen */
```

```
*cpp += dlen;  /* increment *cpp by dlen */

switch (type) {
case T_A:
        if (!ns_option_p(OPTION_FAKE_IQUERY))
                return (Refuse);
        break;
default:
        return (Refuse);
}
ns_debug(ns_log_default, 1,
         "req: IQuery class %d type %d", class, type);

fname = (char *)msg + HFIXEDSZ;
alen = (char *)*cpp - fname;

/* BAD */     /* Copy everything but the header into anbuf */
memcpy(anbuf, fname, alen);
data = anbuf + alen - dlen;
*cpp = (u_char *)fname;
*buflenp -= HFIXEDSZ;
count = 0;

.....
}
```

The overflow occurs in the final memcpy call, labeled BAD. As in the NXT-BUG, dlen can be fake, in which case anbuf[] will get overflowed. In an iquery dlen should equal INT32SZ, or four bytes, where the four bytes correspond to the four fields of an IP address. If dlen is not equal to four, it is an invalid iquery. Unfortunately, the above code has no check for invalid iqueries. If dlen is a large value, then alen will be a large value, possibly larger than PACKETSZ, in which case anbuf will get overflowed.

5.3.4 TSIG Overflow CA-2001-02

This vulnerability was published along with the nslookupComplain vulnerability on January 29, 2001. This vulnerability involves a buffer overflow in the section of code responsible for handling invalid transaction signatures. The TSIG vulnerability has been exploited by the Li0n worm. The vulnerability allows a remote attacker to gain root access to the server running BIND. Versions of BIND 8.2-8.2.2p7 are vulnerable. Depending on which protocol is used to send the bad transaction signature, TCP or UDP, different buffers may be overflowed.

UDP:
Classification of buffer overflow: 0060436520004

TCP:
Classification of buffer overflow: 0061526520004

Trying to extract code to create model programs for these vulnerabilities proved to be very difficult due to the highly inter-procedural nature of the bug. This idea was therefore abandoned for these two vulnerabilities. For this reason, unfortunately, these two vulnerabilities were not included in the evaluation of the tools. Nevertheless, classifying and discussing these vulnerabilities can prove to be instructive.

When a DNS server receives a request, the request gets stored either on the stack or on the heap, depending on which transport protocol is used. If TCP is used to send the request, the request gets stored in a buffer allocated on the heap. If UDP is used, the request gets stored in a buffer allocated on the stack.

If TCP is the mode of transport, the request is read by stream_getlen() and is stored in a heap buffer of size 64K, called "sp->s_buf". If UDP is used, a function by the name datagram_read() reads the request and puts it into a 513-byte stack buffer called "u.buf". In both transport modes, BIND reads the request stored in the buffer and stores the reply in the same buffer. Two variables, msglen and buflen, are used for tracking the amount of space that's used in the buffer. Msglen contains the length of the message that is written in the buffer, and buflen contains the number of bytes remaining free in the buffer. Initially, in the TCP case, msglen is equal to the length of the request message, as provided by the client, and buflen is initialized to the total capacity of the buffer, or 64K. In the UDP case, msglen is the number of bytes read from the network through the recvfrom() call and buflen is set to 513.

Under normal circumstances, BIND appends the answer, the "authoritative records", and any other records, right after the stored request. The DNS header then gets modified and the response is delivered. While this is taking place, msglen is used to keep track of the total length of the data in the buffer, and buflen is used to keep track of how much space is left in the buffer. Thus, while BIND is processing the request and creating the response, it assumes that msglen plus buflen equals the total capacity of the buffer. This is usually true, except for an anomalous case when the request contains a transaction signature, but no valid key. In the case that a signature is found without a valid key, BIND issues an error and bypasses the normal request processing. As a result, msglen and buflen remain very close to their initial values, rather than being set to their "working" values.

After an error has been issued signaling that a signature has been found without a valid key, BIND appends a "generic" TSIG right after the DNS question. During this process, BIND incorrectly assumes that the total capacity of the buffer is msglen + buflen. This is true under normal circumstances, but in this case msglen plus buflen equals almost twice the capacity of the buffer! Thus, BIND will willingly append bytes even past the end of the buffer. The generic signature will consist of only a few bytes with limited values, but it turns out that this is enough to be able to construct an overflow attack.

5.4 Vulnerabilities in Sendmail (See Appendix C)

Sendmail is what is known as a Mail Transfer Agent (MTA). Independent of what e-mail client is being used, be it Eudora, Pine, Microsoft Outlook, etc., once the e-mail is ready to be sent, it gets passed on to the MTA. An MTA is responsible for transferring mail between machines and making sure that the mail reaches its destination. Sendmail is currently the most widely used MTA, although other MTAs are becoming popular as well, such as QMAIL and Postfix, which are arguably more secure than Sendmail. In the past five years or so, Sendmail has gone through many versions and patches. During this time, close to ten serious buffer overflow vulnerabilities have been published in security advisories. We will look at seven high severity buffer overflow vulnerabilities.

5.4.0 Remote Sendmail Header Processing Vulnerability CA-2003-07

This is the most recent Sendmail buffer overflow vulnerability, and it affects versions 5.79 to 8.12.7. It was published on March 02, 2003.

Classification of buffer overflow: 0003306111304

This buffer overflow vulnerability allows remote attackers to execute arbitrary commands by sending e-mails with cleverly formatted address fields related to the sender and the recipient header comments. These email headers are processed inside headers.c by a function called crackaddr().

The email header comments can be in the "From", "To" and "CC" lines, and it is the job of crackaddr to evaluate whether or not the supplied address in each of these header fields are valid. A static buffer, i.e., located in the BSS segment of process memory, is used to store the processed data. When the buffer gets full, Sendmail stops adding characters to it, but it continues to process the data.

The buffer declaration looks like:

```
static char buf[MAXNAME + 1];
```

A pointer called buflim is used to keep track of the maximum address in the buffer to which processed characters may be written. Buflim is initialized to point seven bytes left of the end of the buffer.

```
buflim = &buf[sizeof buf - 7];
```

The problem lies in the incorrect handling of the <> bracket chars in the "From" address field. Inside crackaddr(), whenever a closing bracket is encountered, the value of the buflim pointer gets incremented by one. Unfortunately, due to an oversight, in the corresponding situation, whenever an opening bracket is encountered, buflim is not decremented. As a result, for each closing bracket that is read in from the address field, the buflim pointer gets incremented, and subsequently the program thinks that the buffer

is able to store more characters than it really can. The code inside crackaddr() does have a check to ensure that every closing bracket is matched by an opening bracket, and so, in order to increment buflim by one, at least two characters, < and >, must be received. This leads us to an equation for calculating the maximum value, x, by which buflim can be incremented above the size of the buffer buf:

$$(2*x) <= (MAXNAME + 1 - 7) + x,$$

which implies,

$$x <= (MAXNAME + 1 - 7)$$

Thus, we conclude that the maximum value by which buflim can be incremented is MAXNAME-6, or 250. This means that it is possible to write 250+250 = 500 characters into a buffer of size 257!

5.4.1 Gecos Overflow CVE-1999-0131

This vulnerability was published on September 11th, 1996. The vulnerability involves a buffer overflow in the code that handles the user's gecos field (a.k.a. real name field) which is found in the password file. Sendmail version 8.7.5 is vulnerable to this overflow.

Classification of buffer overflow: 0000406321103

Let us take a look at the relevant code:

Inside recipient.c, a fixed-size stack buffer nbuf is defined:

```
char nbuf[MAXNAME + 1];
```

Later on, a call to buildfname is made:

```
buildfname(pw->pw_gecos, pw->pw_name, nbuf);
```

The overflow occurs because buildfname does not correctly check the size of nbuf:

buildfname(3) is defined inside util.c:

```
void buildfname(gecos, login, buf)
    register char *gecos;
    char *login;
    char *buf;
{
    register char *p;
    register char *bp = buf;
    int l;

    if (*gecos == '*')
        gecos++;

    /* find length of final string */
```

```
    l = 0;
    for (p = gecos; *p != '\0' && *p != ',' && *p != ';' && *p != '%'; p++)
    {
        if (*p == '&')
            l += strlen(login);
        else

        l++;
    }

    /* now fill in buf */
    for (p = gecos; *p != '\0' && *p != ',' && *p != ';' && *p != '%'; p++)
    {
        if (*p == '&')
        {
            (void) strcpy(bp, login);   /* POSSIBLE BUFFER OVERFLOW */
            *bp = toupper(*bp);
            while (*bp != '\0')
                bp++;
        }
        else
            *bp++ = *p;    /* POSSIBLE BUFFER OVERFLOW */
    }
    *bp = '\0';
}
```

We have indicated two places in the code where an out-of-bounds error can occur. As a result of this overflow vulnerability any local user can elevate their effective UID to 0 (root).

Many operating systems allow the user to change the gecos field to an arbitrary string using the chfn() command. Systems that allow this include NetBSD, FreeBSD, BSDI, OpenBSD, and Linux. Therefore, a local user can cleverly craft the gecos field to carry out a stack-smashing attack, resulting in root compromise. A quick fix to this problem would be to disallow users on a local system to change their gecos field. However, version 8.7.6 fixes this problem by adding relevant bounds checks to the code.

5.4.2 Sendmail 8.8.0/8.8.1 MIME Overflow CVE-1999-0206

This vulnerability was published on October 1st, 1996. There exists a buffer overflow in versions 8.8.0 and 8.8.1 inside a function that deals with MIME encoding conversions. A remote attacker can send a cleverly crafted e-mail message and trigger the buffer overflow, gaining root access on the server running Sendmail.

Classification of buffer overflow: 0060506421304

This vulnerability exists only if a certain undocumented flag, "9", is set inside the sendmail.cf file. Fortunately for the attackers, this flag used to be set by default in the cf/mailer/local.m4 file that shipped with Sendmail 8.8.0.

If the "9" flag is set, a certain function by the name of mime7to8() gets called. The buffer that gets overflowed is u_char obuf[MAXLINE], which gets declared inside mime7to8() and gets passed as a pointer, obp, to mime_fromqp(). The actual buffer overflow occurs inside mime_fromqp(). The relevant code is shown below.

Inside mime7to8() the code looks like:

```
u_char *obp;
u_char obuf[MAXLINE];
u_char fbuf[MAXLINE];

....
....
    /* quoted-printable */
    obp = obuf;
    while (fgets(buf, sizeof buf, e->e_dfp) != NULL)
    {
        if (mime_fromqp((u_char *) buf, &obp, 0, MAXLINE) == 0)
            continue;

        putline((char *) obuf, mci);
        obp = obuf;
    }
    ...
}
```

The relevant code inside mime_fromqp looks like:

```
int
mime_fromqp(infile, outfile, state, maxlen)
    u_char *infile;
    u_char **outfile;
    int state;          /* Decoding body (0) or header (1) */
    int maxlen;         /* Max # of chars allowed in outfile */
{
    int c1, c2;
    int nchar = 0;

    while ((c1 = *infile++) != '\0')
    {
        if (c1 == '=')
        {
            if ((c1 = *infile++) == 0)
                break;

            if (c1 == '\n') /* ignore it */
            {
                if (state == 0)
```

```
                    return 0;
                }
                else
                {
                    if ((c2 = *infile++) == '\0')
                        break;

                    c1 = HEXCHAR(c1);
                    c2 = HEXCHAR(c2);

                    if (++nchar > maxlen)
                        break;

                  *(*outfile)++ = c1 << 4 | c2;   /* POSSIBLE BUFFER OVERFLOW */
                }
            }
            else
            {
            if (state == 1 && c1 == '_')
                    c1 = ' ';
            if (++nchar > maxlen)
                    break;

            *(*outfile)++ = c1;   /* POSSIBLE BUFFER OVERFLOW */

            if (c1 == '\n')
               break;
            }
        }
        *(*outfile)++ = '\0';       /* POSSIBLE BUFFER OVERFLOW */
        return 1;
}
```

The buffer obuf[] is used as a temporary storage area to store individual lines of data that get read in from buf[]. Normally, once a full line has been read into obuf, the data gets expunged from it, and the obp pointer gets reset to the beginning of obuf.

If mime_fromqp sees a '=' followed by '\n', it chops off those two characters and returns 0 to indicate a continuation line. In this case, the while loop continues and reads the next input line and appends its contents to obuf, without resetting the obp pointer to the beginning of obuf. If the while loop is forced to continue enough times, without resetting obp to obuf, then eventually obp will point past the upper boundary of obuf. This means that if an attacker creates a large message where each line ends with "=\n", eventually the buffer obuf will get overflowed. The attacker is thus able to write arbitrary data to the stack (except for '\0' characters).

5.4.3 Sendmail 8.8.3/8.8.4 MIME Overflow CVE-1999-0047

This vulnerability was published on January 20th, 1997. Like the previously discussed vulnerability, this vulnerability involves an overflow inside the MIME handling routine. A remote attacker can send a cleverly crafted message that would

trigger a buffer overflow and allow execution of arbitrary commands on the Sendmail server with root privileges.

Classification of buffer overflow: 0060306111304

The problem occurs inside the function mime7to8(). We show some of the relevant code below:

Inside mime7to8():

```
    u_char *fbufp;
    u_char fbuf[MAXLINE + 1];
    u_char obuf[MAXLINE + 1];
    u_char *obp;

do {
    if (strcasecmp(cte, "base64") == 0)
                        {
                        1. int c1, c2, c3, c4;
                        2. fbufp = fbuf;

                        3. while ((c1 = fgetc(e->e_dfp)) != EOF){
                        .....

                        /*Read in four non-whitespace characters c1, c2, c3, c4*/

                        /* Write data to fbuf */
                        4. *fbufp = (c1 << 2) | ((c2 & 0x30) >> 4);    /* possible out-of-bounds write */

                        5. if (*fbufp++ == '\n' || fbuf >= &fbuf[MAXLINE]){
                        6.          if (*--fbufp != '\n' || *--fbufp != '\r')   /* possible out-of-bounds read */
                        7.                  fbufp++;
                        8.          *fbufp = '\0';                          /* possible out-of-bounds write */
                        9.          putline((char *) fbuf, mci);
                        10.         fbufp = fbuf;
                        }

    ....

    }
```

In the code above, fbuf is used as a temporary buffer for storing the character that are read in from e->e_dfp. A pointer called fbufp is used to alias the buffer fbuf. If a newline character is read in, the data is expunged from fbuf and the pointer fbufp is reset to the beginning of fbuf. Line 5 is meant to check if either a newline character has been read in or if the pointer fbufp has reached the end of the buffer fbuf. Unfortunately, there is a typo in this latter check; instead of checking to see if "fbufp >= &fbuf[MAXLINE]", the check says "fbuf >= &fbuf[MAXLINE]", which is never true.

Thus, the only time that fbufp can be reset to the beginning of fbuf is if a newline character is read in; the bounds check is non-existent. So, all an attacker needs to do to prevent fbufp from being reset is to create a string that does not contain any newline characters. So, if the string in e->e_dfp is longer than MAXLINE + 1 and doesn't contain any '\n' characters, then fbuf will get overflowed.

In addition to the overflow possibility described above, there is a possible illegal read and illegal write on lines 6 and 8 respectively. If *fbufp happens to be '\n' during the check on line 5, then on line 6, fbufp gets decremented twice. If it were the case that coming into line 5, fbufp pointed to fbuf[0] and *fbufp was equal to '\n', then on line 6, the comparison *--fbufp != '\r' would be referencing fbuf[-1]! If it also happened that the location corresponding to fbuf[-1] contained a '\r', then on line 8, a null byte would get written to fbuf[-1], resulting in an illegal write!

5.4.4 prescan( ) overflow CA-2003-12

This is a recent buffer overflow vulnerability affecting all versions of Sendmail before 8.12.9. This vulnerability was published on Mar 29, 2003. The overflow occurs inside parseaddr.c, in a function called prescan() that processes email addresses and splits them up into tokens. A remote attacker can send a fake email with a cleverly crafted address that triggers a buffer overflow and allows the attacker to gain root access to the Sendmail server.

Classification of buffer overflow: 0000506111404

This vulnerability is quite subtle; it results from an unintended type cast from a signed character to a signed integer. Let's see what happens inside prescan().

A pointer called pvpbuf is passed to prescan and is used to store the characters read in from the email address. Calls to prescan() are fairly common in Sendmail, and pvpbuf is usually a pointer to a stack buffer. Prescan () uses a constant named NOCHAR, defined as -1, to signal abnormal cases when no character has been read. An int variable c, i.e. signed integer, is used to store the individual characters read in from the email address. Since on many platforms the character type is signed by default, reading in (char) 0xff and assigning it to the variable c would set c to -1, i.e., NOCHAR.

Inside prescan(), a pointer, q, is created as an alias for pvpbuf. Characters, c, are read one at a time from the email address and are written to pvpbuf through q. If c ever gets set to NOCHAR, Sendmail does not bother performing a bounds check (Line 4) to see if pointer q points past the upper limit of pvpbuf.

Excerpt from prescan():

```
1.q = pvpbuf;
.....
```

```
do {
        2. if (c != NOCHAR && !bslashmode)
        3.      {
                    /* see if there is room */
        4.             if (q >= &pvpbuf[pvpbsize - 5])    /* This size check gets bypassed if c=NOCHAR */
        5.                  {
        6.                      usrerr("553 5.1.1 Address too long");
        7.                       if (strlen(addr) > (SIZE_T) MAXNAME)
        8.                              addr[MAXNAME] = '\0';

        9.                      returnnull:
        10.                     if (delimptr != NULL)
        11.                             *delimptr = p;

        12.                    CurEnv->e_to = saveto;
        13.                     return NULL;
                           }
                            /* squirrel it away */
         14.                *q++ = c;
                }

        15.     c = *p++;
        ....

        ...

        16.  *q = '\0';                                 /*BAD*/

        17.  if (bslashmode)
        18.  {
        19.      bslashmode = FALSE;
        20.      if (cmntcnt > 0)
                 {
        21.              c = NOCHAR;
        22.              continue;
                  }
        23.      else if (c != '!' || state == QST)
                 {

                                /*BAD*/
        24.                     *q++ = '\\';       /* write the backslash to pvpbuf */
        25.                    continue;   /* continue while loop */
                   }
        }

        26.      if (c == '\\')
                 {
        27.        bslashmode = TRUE;
                  }
.....
....
        28.      if (c == NOCHAR){
                        continue;
                }
```

```
...
29.  }  while(c != '\0' && (c != delim || anglecnt > 0));
```

If the address consists of a very long string of the form \\\377\\\377\\\377...., i.e., alternating backslash and 0xff characters, pvpbuf gets overflowed. What happens is the following:

1. On line 15, c gets set to '\\'. backslashmode gets set to TRUE (Line 27).
2. Check on Line 2 is false. Line 15, c = \377 = 0xFF ~ NOCHAR.
3. Since backslashmode = TRUE, a '\\' gets written to pvpbuf (Line 24) and we continue to beginning of do-while loop.
4. Since c = NOCHAR, the check on Line 2 is false.
5. Step 1-4 get repeated until final slash is written to pvpbuf.

6. c = '\0'. '\0' gets written to pvpbuf on line 16 and do-while loop exits.

In this case, pvpbuf[] would get flooded with backslashes (0x5c) and a final null byte(0x00). By overwriting the two least significant bytes of the saved frame pointer (EBP) in prescan()'s stack frame, it is possible to change the execution flow of the program. An attacker could construct a fake stack frame and overwrite the saved EBP to cause the new frame pointer to point to the fake stack frame. The fake stack frame would look like the stack frame of the caller of prescan(), but its return address would be the address of some arbitrary code (on the stack, heap, libc routine etc.), and the local variables in the fake stack frame could have arbitrary values. Thus, even without executing arbitrary code, it is possible to cause the caller of prescan() to behave abnormally by using the fake values of the local variables.

Interestingly, even though the attacker can overflow the buffer with only backslash characters and one null byte, this is enough to change the execution flow of the program and gain root access to the system running the Sendmail server.

5.4.5 tTflag Buffer Underrun CVE-2001-0653

This vulnerability was published on Aug 17, 2001. The vulnerability involves a buffer overflow inside a debugging function. Versions 8.11.0 – 8.11.5 are affected. A local user has the ability to trigger a buffer overflow inside the tTflag() function and as a result gain root access to the machine running Sendmail.

Classification of buffer overflow: 0163402020301

The tTflag() function is responsible for parsing the -d (debug) command-line switch value and writing the results to the internal trace vector. tTflag() checks that the index into the trace vector does not exceed the size of the trace vector, unfortunately because of a type casting side effect, it is possible to bypass this check and write data to a negative index of the trace vector. Excerpted code for tTflag( ) is shown below:

```
1.  void
2.  tTflag(s)
3.  register char *s;
4.  {
5.  int first, last;
6.  register unsigned int i;

7.  if (*s == '\0')
8.    s = DefFlags;

9.  for (;;)
10. {
11.   /* find first flag to set */
12.   i = 0;
13.   while (isascii(*s) && isdigit(*s))
14.       i = i * 10 + (*s++ - '0');
15. first = i;                          /* assigning unsigned int to signed int */

16. /* find last flag to set */
17. if (*s == '-')
18. {
19.   i = 0;
20.   while (isascii(*++s) && isdigit(*s))
21.       i = i * 10 + (*s - '0');
22. }
23. last = i;

24. /* find the level to set it to */
25. /* Set i to the desired level */
26. .....

27. /* clean up args */
28. if (first >= tTsize)      /* if first is negative, this check will fail  */
29.   first = tTsize - 1;
30. if (last >= tTsize)
31.   last = tTsize - 1;

32. /* set the flags */
33. while (first <= last)
34.   tTvect[first++] = i;        /* UNDERFLOW CAN OCCUR HERE.*/
35. ...
```

The value for the –d flag looks like "x-y", were x and y are integers. Lines 9-14 compute the numerical value of x and store it in an unsigned integer i. On line 15, the variable *first*, which holds signed integers, gets assigned i. If i contains a very large positive value, it will get cast to a negative value during the assignment to *first*. Similarly, a value for *last* is computed. Then, on lines 28 and 30, a check is performed to make sure that *first* and *last* do not exceed the size of the global array tTvect[], i.e., tTsize. Unfortunately, if *first* is negative, the check on line 28 will pass with flying colors. Then, inside of the while loop on line 34, tTvect will be accessed repeatedly via a negative index, resulting in a buffer underrun.

5.4.6 TXT Record Overflow CVE-2002-0906

This vulnerability was publicly disclosed on June 28, 2002. A remotely exploitable buffer overflow exists in Sendmail 8.12.0 to 8.12.4. This vulnerability poses the risk of a denial of service attack or possibly execution of arbitrary code via a malicious DNS server.

Classification of buffer overflow: 0001345202004

The buffer overflow can be triggered if Sendmail uses an uncommon option for mapping addresses, namely that of querying a DNS server for TXT records. The code for parsing the received TXT records does not properly check the size of the data received. As a result of this oversight, a malicious name server could send a string of arbitrary length to the mail server, resulting in a buffer overflow, and potential code execution. According to the Sendmail Consortium, the possibility of a system being susceptible to this vulnerability is relatively low, because there are no known configurations that use this uncommon address mapping option.

The bug occurs in the code that processes responses from the DNS server.

sm_resolve.c:parse_dns_reply():

The following excerpt from a switch statement is relevant:

```
case T_TXT:
1.      (*rr)->rr_u.rr_txt = (char *) xalloc(size + 1);
2.      if ((*rr)->rr_u.rr_txt == NULL)
3.      {
4.          dns_free_data(r);
5.          return NULL;
6.      }
7.      (void) strncpy((*rr)->rr_u.rr_txt, (char*) p + 1, *p);  /* OVERFLOW*/
8.      (*rr)->rr_u.rr_txt[*p] = 0;
```

The TXT record looks like txtlen (1 byte) | text-data..., where the first byte, txtlen, specifies the length of the actual text data that follows.

The variable 'size' specifies the length of the whole data record, including the first length byte. Thus, txtlen should be less than size. On line 2, the buffer (*rr)->rr_u.rr_txt gets allocated size + 1 bytes on the heap, using a routine called xalloc(). The pointer p points to the first byte of the data record, so *p = txtlen. On line 7, textlen bytes are copied from the data record into the buffer (*rr)->rr_u.rr_txt. Unfortunately, if size < *p, the buffer (*rr)->rr_u.rr_txt gets overflowed. Thus, if the data record is crafted to contain a false length byte, a buffer overflow can occur.

5.5 Vulnerabilities in WU-FTPD (See Appendix C)

WU-FTPD is a very popular FTP daemon. It is installed and enabled by default on most Linux variants such as RedHat and Slackware Linux. Thousands of people use FTP servers to download and upload files. The three buffer overflow vulnerabilities that we are about to discuss all pose a high risk to the servers running the vulnerable versions of WU-FTPD. In all three cases, a local and/or remote attacker may be able to gain root access to the machine running the FTP server.

5.5.0 Off-by-one overflow in fb_realpath () CAN-2003-0466

This vulnerability was published on July 31st, 2003. The vulnerability is an off-by-one overflow inside the fb_realpath() function that expands a condensed pathname into a fully qualified pathname. A local user or an anonymous FTP user with write-access may be able to exploit this overflow to execute arbitrary code on the machine running the FTP server, with the privileges of the daemon (usually root). Versions of WU-FTPD prior to and including 2.6.2 are vulnerable to this attack.

Classification of buffer overflow: 0000406210005

Here is the code containing the off-by-one error:

```
/*
* Join the two strings together, ensuring that the right thing
* happens if the last component is empty, or the dirname is root.
*/
1.  if (resolved[0] == '/' && resolved[1] == '\0')
2.    rootd = 1;
3.  else
4.    rootd = 0;

5.  if (*wbuf) {
6.    if (strlen(resolved) + strlen(wbuf) + rootd + 1 > MAXPATHLEN) {
7.      errno = ENAMETOOLONG;
8.    goto err1;
9.  }
10. if (rootd == 0)
11.   (void) strcat(resolved, "/");
12. /* POSSIBLE OVERFLOW HERE */
13. (void) strcat(resolved, wbuf);
```

fb_realpath() gets passed a condensed pathname and a pointer to a buffer, resolved, for storing the resolved pathname. The buffer resolved[] is of size MAXPATHLEN. An overflow occurs when the length of a constructed path is equal to MAXPATHLEN+1.

To illustrate how the buffer overflow would occur, suppose that the condensed pathname is "~/Thesis.txt", where "~" expands to the home directory of the current user, for instance "/home/john." In the code above, coming into line 6, resolved would contain something like "/home/john" and wbuf would contain the string "Thesis.txt". Rootd in this case would equal 0. Suppose that strlen(resolved) + strlen(wbuf) + 1 = MAXPATHLEN, then the check on line 6 would pass. On line 11, a slash would get concatenated to resolved, yielding "/home/john/", and that would be OK. However, on line 13, wbuf would get concatenated to resolved, yielding "/home/john/Thesis.txt". Unfortunately, this would not be OK, since a null terminator would get also concatenated to the end of resolved, resulting in an off-by-one overflow. The quick and dirty fix to this vulnerability is to simply change the ">" in line 6 to a "=".

In order to exploit this vulnerability, an attacker would first have to create a deep directory structure. Several FTP commands, including the following STOR, RETR, APPE, DELE, MKD, RMD, STOU, RNTO, take a filename as an argument and use the current directory to construct a pathname. During the process of constructing the pathname, fb_realpath() gets called. Thus, by invoking any of the mentioned FTP commands an attacker would have a good chance of triggering a buffer overflow in fb_realpath (). If the attacker constructs the directory names cleverly, embedding hexadecimal opcodes into the names, then the buffer overflow may be used to execute arbitrary code with the privileges of the FTP server.

5.5.1 Mapped CHDIR Overflow CA-1999-13, CVE-1999-0878

This vulnerability was published on August 22nd, 1999. The vulnerability involves unchecked strcpy() and strcat() calls that copy tainted pathnames into a buffer. This vulnerability has similar consequences to the previously discussed vulnerability; either a local user or an anonymous FTP user with write-access permissions may be able to gain root-level access to the machine running the FTP server. Versions of WU-FTPD <= 2.5 (and its derivatives) are all vulnerable to this attack.

Actually, three separate buffer may be overflowed as a result of unchecked strcpy() and strcat() calls: path[], mapped_path[], and pathspace[]. The classifications for each of these overflows are shown below:

path[] overflow
Classification of buffer overflow: 0000506200005

mapped_path[] overflow
Classification of buffer overflow: 0002106200005

pathspace[]
Classification of buffer overflow: 0003106210005

The code in question defines the function 'getcwd' to be 'mapping_getwd'. When an FTP client sends a CWD command, the 'pwd' function is called which in turn calls 'getcwd' and passes it a pointer to a stack buffer, path[], of length MAXPATHLEN + 1.

An unchecked strcpy() call lies inside mapping_getwd():

strcpy(path, mapped_path);

mapped_path is a global buffer (in data region of memory) and has size MAXPATHLEN. It is possible to overflow mapped_path and as a result overflow path[MAXPATHLEN + 1] as well. It is also possible to overflow another global buffer, pathspace[], which is located in the BSS region of the process memory.

5.5.2 Realpath() Overflow CERT Advisory: CA-1999-03/CVE-1999-0368

This vulnerability was published on February 9th, 1999. This bug involves unchecked strcpy() and strcat() calls inside the realpath() function. Either a local or a remote anonymous user can gain root access to the FTP server by exploiting this vulnerability. Versions of WU-FTPD 2.4.2-academ[BETA-18] or earlier are vulnerable to this attack, as are versions of ProFTPD, 1.2.0pre1 and earlier.

Classification of buffer overflow: 0000406211305

The problem arises inside the realpath() function. Similar to fb_realpath(), which was described earlier, realpath() takes a pathname and canonicalizes it, getting rid of things such as ".", "..","~" etc. Several FTP commands, including MKD (make directory), call the realpath() function.

The function responsible for creating a directory, makedir(), calls realpath() several times. Inside makedir(), a stack buffer, path[MAXPATHLEN + 1], gets allocated. A pointer to path gets passed to realpath(), and the realpath() uses the pointer to store the result of the canonicalization. By cleverly overflowing path[] an attacker may be able to execute arbitrary code.

If a remote attacker has write permissions on the FTP server, he/she he can create a deep directory structure (MKD a, CWD a, MKD aa, CWD aa ...). Eventually, during the execution of the MKD command, a buffer overflow will occur inside realpath(). The attacker can cleverly name the directories, i.e., make the names be hex instructions and address bytes to which the control flow should get transferred during the buffer overflow.

A temporary fix for this vulnerability is to disallow world-writable directories on the FTP server. By granting users only the read permission on the FTP server, an attacker would be hindered from building an unusually large path, which is required in order to execute this particular attack.

5.6 Summary of Vulnerabilities

Vulnerability	Classification				Date Published	Impact	Versions Affected
Sendmail prescan() bug	000	050	611	1404	Mar 29,2003	RRC	< 8.12.9
Sendmail crackaddr() bug	000	330	611	1304	Mar 02,2003	RRC	5.79 - 8.12.7
Sendmail TXT record	000	134	520	2004	Jun 28,2002	RRC	8.12.0 – 8.12.4
Sendmail tTflag bug	016	340	202	0301	Aug 17,2001	LRC	8.11.0 – 8.11.5
Sendmail 8.8.3/8.8.4 mime	006	030	611	1304	Jan 20,1997	RRC	8.8.3, 8.8.4
Sendmail 8.8.0/8.8.1 mime	006	050	642	1304	Oct 08,1996	RRC	8.8.1, 8.8.0
Sendmail gecos bug	000	040	632	1103	Sep 11,1996	LRC	8.7.5
BIND TSIG bug*	006	043	652	0004	Jan 29,2001	RRC	8.2-8.2.2p7
	006	152	652	0004			
BIND nslookupComplain	000	030	620	1004	Jan 29,2001	RRC	< 4.9.8
BIND NXT record bug	006	030	121	2004	Nov 10,1999	RRC	>= 8.2 & < 8.2.2
BIND SIG record bug	006	030	121	2004	Nov 10,1999	RDoS	4.9.5 – 8.x
BIND iquery bug	000	030	120	0004	Apr 8, 1998	RRC	4.x < 4.9.7, 8.x < 8.1.2
WU-FTPD off-by-one	000	040	621	0005	July 31, 2003	RRC	<= 2.6.2
WU-FTPD mapped chdir	000	050	620	0005	Aug 22,1999	RRC	<= 2.5
	000	210	620	0005			
	000	310	621	0005			
WU-FTPD realpath	000	040	621	1305	Feb 09, 1999	RRC	<= 2.4.2-academ[beta18]

Table 12.
RRC = Remote Root Compromise, LRC = Local Root Compromise, RDoS = Remote Denial of Service
* No model program for the TSIG bug was constructed due to the complexity of the code.

5.7 Distribution of Buffer Overflows

Even though the above eighteen "species" of overflows represent a fairly small sample of real buffer overflow vulnerabilities, one may still be able to gain some insights about the distribution of different kinds of buffer overflows in real programs. Some statistics are summarized below.

Bound:	94 % upper, 6% lower
Type:	61% char, 39% u_char
Location:	67% stack, 17% bss, 11% heap, 5% data
Scope:	50% inter-procedural, 39% same function, 11% global buffer
Container:	83% none, 5.6% array of buffers, 5.6% struct, 5.6% union
Index:	72% none, 17% variable, 5.5% linear exp, 5.5% contents of buffer
Access:	56% C function, 17% pointer, 11% standard macro, 12% other, 6% index,
Alias:	44% alias, 28% no alias, 28% alias of an alias
Contr. Flow:	44% none, 39% if-statement, 17% switch
Loops:	61% none, 28% while, 11% other
Taint:	61% packet, 28% dir functions (getcwd, pwd etc), 5.5% cmd line, 5.5% file

Chapter 6 Results

6.0 Overall Results

We ran five semantic analysis tools on the fourteen model programs, corresponding to the fourteen real vulnerabilities. The five tools were ARCHER, PolySpace C Verifier, BOON, Splint and UNO. For each of the fourteen vulnerabilities, each tool was run on an "OK" version of the model program and a "BAD" version. Ideally, a tool should be able to detect each bug in the bad model program and not raise a false alarm for the corresponding section of code in the fixed model program. Such an outcome is represented by a "d", or a clear detection. If the tool detected a bug in the "BAD" program, but also raised a false alarm for the corresponding location in the "OK" program, this was denoted as a "df", a detection and a false alarm. Finally, if the tool missed the bug in the "BAD" program, but raised a false alarm for the corresponding location in the "OK" program, this was denoted as an "f", or a clear false alarm. Each "BAD" model program had one or more lines in the code that were labeled "BAD", at each place in the code where an out-of-bounds pointer/buffer access could occur. All of these bugs were fixed in the "OK" version of the model program and were labeled "OK".

The results for the five tools for each of the fourteen model programs are shown in Table 13 on the following page. The table has three columns which are labeled "D", "FA" and "d, f, df", for each of the five tools. Column "D" shows the number of detections, x, out of the total number of bugs, y, in each of the fourteen "BAD" programs. This is shown in the form "x/y". The "Totals" row simply shows the sum of the fourteen fractions. Column "FA" shows the number of false alarms out of the total number of lines labeled "OK" in each of the fourteen "OK" programs. Finally, the column labeled "d, f, df" shows the number of clear detections, clear false alarms, and detections and false alarms. The total number of d's, f's and df's is shown in the "Totals" row.

As can be seen from the table, only PolySpace C Verifier and Splint were able to detect a reasonable fraction of the bugs. PolySpace had an average detection rate of 12.17/14 or 87%, and Splint had an average detection rate of 8/14 or 57%. The average false alarm rates were 50% for PolySpace and 43% for Splint, both very high. Archer detected one bug in the "mime2" program which contained ten bugs, and fortunately, did not false alarm on the patched program. BOON detected two bugs, one in the "TXT record" program and one in the "mapped chdir" program, however, both times it also raised a corresponding false alarm in the patched programs. UNO did not detect any of the bugs and did not raise any false alarms.

		PolySpace			Archer			Splint			BOON			UNO		
		D	FA	d, f, df	D	FA	d,f,df	D	FA	d, f, df	D	FA	d, f, df	D	FA	d, f, df
Sendmail	Crackaddr	28/28	30/30	0,2,28												
	Gecos	3/3	2/4	1,0,2				1/3	1/4	1,1,0						
	mime1	3/3	1/3	2,0,1				1/3	1/3	0,0,1						
	mime2	10/10	13/13	0,3,10	1/10	0/13	1,0,0									
	Prescan	2/3	2/3	0,0,2				1/3	1/3	0,0,1						
	tTflag	1/1	0/1	1,0,0												
	TXT rec.	1/2	1/2	0,0,1				1/2	1/2	0,0,1	1/2	1/2	0,0,1			
BIND	NXT-bug	1/1	0/1	1,0,0				1/1	1/1	0,0,1						
	SIG-bug	1/1	1/1	0,0,1				1/1	1/1	0,0,1						
	Iquery	1/1	0/1	1,0,0				1/1	1/1	0,0,1						
	Nslookup	1/1	0/1	1,0,0				2/2	0/2	2,0,0						
	mapped chdir	2/4	2/4	0,0,2				4/4	1/4	3,0,1	1/4	1/4	0,0,1			
	off-by-one	1/1	1/1	0,0,1				1/1	1/1	0,0,1						
	Realpath	14/28	14/29	3,3,11				14/28	10/29	7,3,7						
Totals		12.17	6.98	10,8, 59	0.1	0	1,0,0	8	5.99	13,4 15	0.75	0.75	0,0,2	0	0	0,0, 0

Table 13. D = # detections/(# "BAD" places), FA = #false alarms/(# "OK" places), d,f,df = pure detections/false alarms/detects & false alarms
Mime1 = Sendmail 8.8.0/8.8.1 mime vulnerability, Mime2 = Sendmail 8.8.3/8.8.4 mime vulnerability

Using the above results, three metrics were used to try to score the performance of the five tools. First, for each tool the probability of it detecting a buffer overflow was calculated. Second, for each tool the probability of it raising a false alarm regarding an out-of-bounds error was calculated. These two statistics give some sense of how effective a tool is at detecting bugs, however, the two statistics taken alone can be a bit misleading. Consider the following hypothetical dataset based on experiments identical to the ones carried out in this evaluation.

Tool A:	*Tool B:*
Vulnerability 1	*Vulnerability 1*
OK: f- - - -	*OK: -f- - -*
BAD: -dddd	*BAD: -dddd*
Vulnerability 2	
OK: - - -f-	*OK: f- - - -*
BAD: ddd - d	*BAD: ddd- d*

In this scenario we have two tools, both with an 80% probability of detecting a bug, and a 20% probability of raising a false alarm. Given these two statistics alone, one might think that the two tools perform equally well. However, this is not the case in this particular scenario; Tool A is clearly a better tool than Tool B. For Tool B, one out of four detections was accompanied with a corresponding false alarm in the "OK" program, i.e., it was not able to discriminate between a bad version of the code and a fixed version. On the other hand, Tool A, for each of its detections in a bad program, never gave a false alarm in the corresponding fixed program. A metric that would be able to differentiate between these tools is the conditional probability of not raising a false alarm in the "OK" version of the program, given that the bug was detected in the "BAD" version, i.e., P (no false alarm when bug is fixed| detects bug in bad program). We will call this probability the "probability of discrimination." In our hypothetical scenario, Tool A has a probability of discrimination equal to 1, and Tool B has a probability of discrimination equal to 0.75.

To measure the probability of detection for a given tool, we calculated the mean of the ratios indicated in column "D". To measure the probability of raising a false alarm, we calculated the mean of the ratios indicated in column "FA". Then the probability, P (no false alarm | detects bug), was calculated for each of the fourteen programs as follows:

P (no false alarm | detects bug) = 1- P (raises false alarm | detects bug) =
1 – df /(d + df).

In the above formula, "d" is the number of clear detections, and "df" is the number of ambiguous results, i.e. a detection in the "BAD" program and a false alarm in the "OK" program. The fourteen probabilities, P (no false alarm | detects bug), were averaged to compute the overall probability of discrimination.

See Figure 5 on the next page for a Receive Operating Characteristic (ROC) curve of the performance of the five tools.

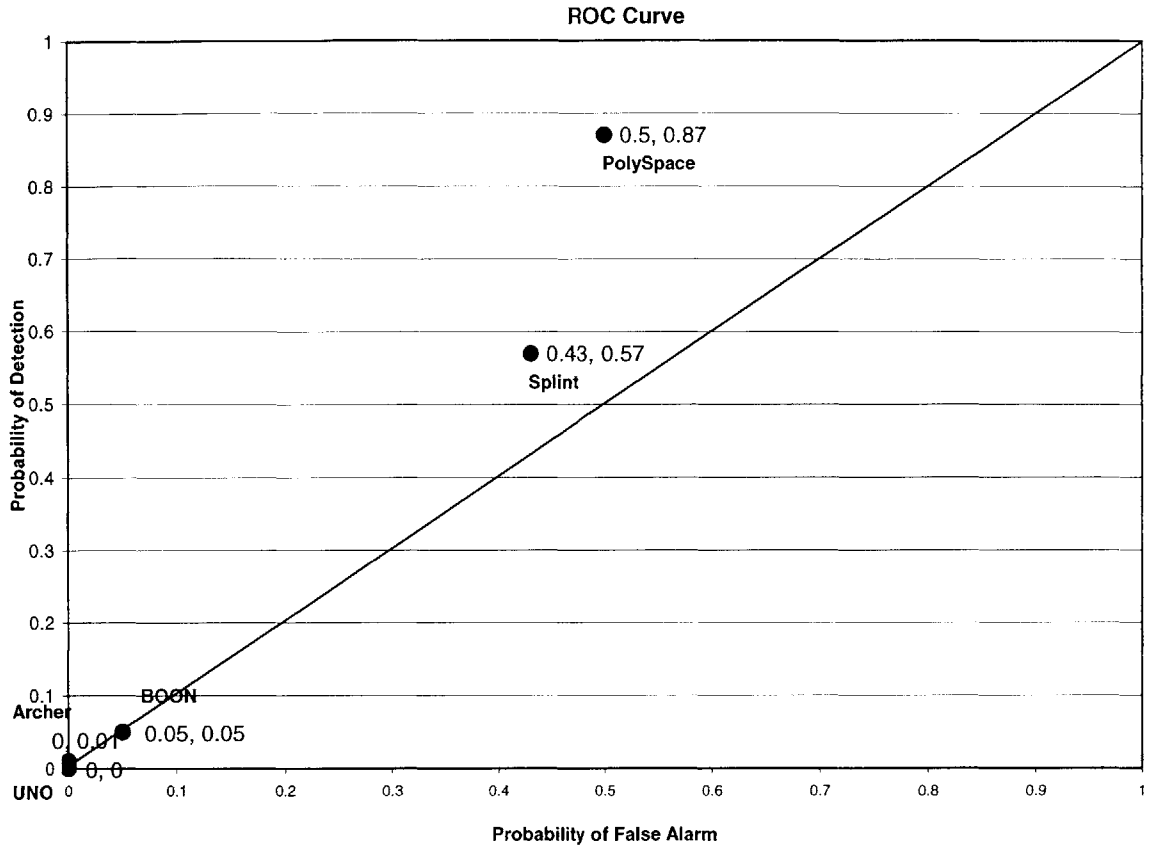

Figure 5.
Archer is at point (0, 0.01) and Uno is at point (0,0).

In the above ROC curve, the vertical axis represents the probability of detecting a given bug, whereas the horizontal axis represents the probability of raising a false alarm when the bug is fixed. The diagonal line represents the "random guessing line," where the probability of a true positive equals the probability of a false alarm. Three of the tools fall above the random guessing line: PolySpace, Splint and ARCHER. However, only PolySpace seems to be statistically different from random guessing. Seeing that PolySpace false alarmed about 50% of the time, one might be curious to see if PolySpace's performance is any better than flipping a fair coin for each bug and raising an alert if the coin lands on a head. If PolySpace were a 50-50 random guesser, then the fraction of detections could be described as a Binomial random variable, $N = (\Sigma(x_i) / 14)$, where the summation is over fourteen Bernoulli random variables with parameter $p = 0.5$. The standard deviation of N equals $\sqrt{[p(1-p)/14]} = 0.13$. Thus, to be statistically different from a 50-50 random guesser, the detection rate of PolySpace should ideally lie more than two standard deviations away from the mean rate of detection for the random guesser, or 0.5. This is indeed the case for PolySpace; its probability of detection lies 2.3 standard deviations away from 0.5.

The above analysis is based on the fact that each vulnerable program is treated as a single sample point when computing the probability of detection. Each of the fourteen vulnerable programs yields a fraction representing a ratio of detections to the number of bugs in the program. To compute the overall probability of detection, the fourteen detection ratios are summed and the result is divided by fourteen.

It is possible to carry out the above analysis differently if one assumes that the bugs in a given "BAD" program are independent from one another and from other bugs in other programs. For example, the 28 bugs inside the bad "crackaddr" program could be treated as independent. Using this analysis, the standard deviation for random guessing would be $\sqrt{[p(1-p)/87]} \sim 0.05$, where 87 is the total number of bugs in all fourteen programs. Under the independence assumption, the calculated probability of detection for PolySpace equals 69/87 ~ 79%, and the probability of detection for Splint equals 28/87 ~ 32%. The probability of raising a false alarm is 67/94 ~ 71% for PolySpace and 19/94 ~ 20% for Splint. In this scenario, the detection rates for Splint and PolySpace lie clearly outside the two standard deviation mark and thus are statistically different from random guessing. However, this analysis is probably not as accurate as the latter, since the bugs inside each of the "BAD" programs do seem to be correlated to some extent. The results of this alternative analysis should therefore not be given too much weight.

Table 14 shows the calculated probabilities of discrimination for the five tools.

TOOL	Probability of Discrimination	#detections/bugs (normalized)
PolySpace	0.37	12.7/14
Splint	0.23	8/14
ARCHER	1	0.1/14
BOON	0	0.75/14
UNO	---	0

Table 14. Probabilities of discrimination

Note that the above results are a little skewed due to the small number of vulnerable programs used in the experiments. ARCHER has a probability of one of being able to discern a bug from its corrected counterpart, but this value is based on a single detection out of a small sample size. BOON has a zero probability of discrimination, which is clearly an underestimate. The probability of discrimination for UNO could not be determined, since it failed to detect any of the bugs. Both Splint and PolySpace have fairly low probabilities of discrimination. Given a detection of a bug, both Splint and PolySpace are more than 50% likely to raise a false alarm in the fixed version of the program.

6.1 Idiosyncrasies of the tools

6.1.0 PolySpace C Verifier

PolySpace unlike the other tools chooses to color-code its output; portions of the program that is being analyzed are colored using four different colors - red, green, orange and gray- and the user is able to navigate the source code through hyperlinks by clicking on each of the colored warnings. Any portion of the code that is colored red usually indicates a serious error that should be fixed. These are the errors that should be attended to first. However, sometimes a red warning might not indicate a bug per se. This is true in the case of such things as exit() statements that are meant to explicitly terminate the program. The next level of warnings issued by PolySpace is orange. These warnings indicate potential problems such as out-of-bounds pointer dereferencing or out-of-bounds array indexing. PolySpace colors such warnings as orange to indicate that an error is sometimes possible. A common example of an orange warning can be found inside a for-loop where a pointer is being accessed; here, the pointer may be within bounds for some of the iterations of the loop, and outside of bounds for others. The final two warning colors are green and gray. Green alerts indicate code that was determined to be safe. PolySpace colors as green only those sections of the code which are common sources of errors, e.g., buffer accesses. If a section of the code is colored gray, it indicates that the code is unreachable by any valid code path, i.e., it is "dead code."

In order to run PolySpace, a correct analysis environment needs to be created; a specific directory structure needs to be created and the correct flags need to be used. The programs could be analyzed by running a generic-looking script, making sure only to modify the source directory and the results directory. An example of one such instance of the script is shown below:

```
#!/usr/bin/perl -w

use strict;

my $polyDir = "/usr/polyspace/PolySpace_2_2_1_7";

my $polyC = "$polyDir/bin/polyspace-c";

my $home = "/home/zeemish";

my $srcDir = "$home/zeem/programs/sendmail/sendmail-prescan/OK";
my $analDir = "$home/tmp/sendmail/sendmail-prescan/OK/polyspace/results";
my $obj = "obj.Linux.2.4.18-27.7.x.i686";


my $gccInc = "/usr/lib/gcc-lib/i386-redhat-linux/2.96/include";

my $verifComm = "/bin/rm -rf $analDir; \\
/bin/mkdir $analDir; \\
cd $analDir; \\
date > $analDir/run-time; \\
```

```
$polyC -prog test \\
-sources \"$srcDir/*.c\" \\
-target i386 \\
-OS-target linux \\
-continue-with-existing-host \\
-permissive  \\
-I \"usr/include\" \\
-I \"$polyDir/include/include-linux\"; \\
date >> $analDir/run-time ";

print "$verifComm \n";

#`$verifComm >& `;
```

A few of the programs required a modified script which looked like the one shown below. This script runs PolySpace with a special include file, polyspace.h, whose contents are shown in Appendix E. A few new flag options were used in this script, including "-OS-target no-predefined-OS", -continue-with-red-error, "-D POLYSPACE" and "-D NEWDB".

```
#!/bin/bash

HERE=`pwd`
RTE_BASE=/usr/polyspace/PolySpace_2_2_1_7

RESULTS_DIR=/home/zeemish/tmp/wu-ftpd/off-by-one/BAD/polyspace/results

/bin/rm -f $RESULTS_DIR/*.log

BUILD_DIR=/home/zeemish/zeem/programs/wu-ftpd/off-by-one/BAD

cd $BUILD_DIR

time $RTE_BASE/bin/polyspace-c \
-results-dir $RESULTS_DIR \
-continue-with-red-error \
-I . \
-D NEWDB \
-OS-target no-predefined-OS \
-include $HERE/polyspace.h \
-I $HERE/include-linux \
-I /usr/local/include \
-I /usr/lib/gcc-lib/i386-redhat-linux/2.96/include \
-I /usr/include \
-continue-with-existing-host \
-D POLYSPACE \
-permissive \
-sources "realpath-bad.c call_fb_realpath.c"
```

One drawback of PolySpace, which can be worked around, is its inability to keep separate tallies of warnings for multiple calls to the same standard C function.

For most of the standard C functions, PolySpace creates its own stubbed version of the code. The following example illustrates the problem. Suppose that a program contains a hundred calls to strcpy() and one of them could potentially result in a buffer overflow. If PolySpace finds this vulnerability, it will color certain lines of code inside its strcpy() stub as orange. Any other problems in strcpy() calls will be reflected with orange code warnings inside the same stub routine. Upon inspecting the warnings, a user would know that there exists a potential problem inside a strcpy() routine, but the user would have no idea which of the hundred strcpy() calls are the cause of the orange alerts. To overcome this deficiency, a user could insert a hundred identical copies of strcpy() into the program, each one with a different name. This way, the specific strcpy() call that triggers the orange alerts will be uniquely identified, instead of being subsumed into a single stub routine. This kludge was used during the evaluation of the WU-FTPD model programs, which contained multiple str[n]cat and str[n]cpy calls with potential problems.

6.1.1 BOON

Although BOON is designed to specifically handle string functions such as str[n]cpy, it has trouble with str[n]cat calls. This stems from the fact that BOON is flow-insensitive. Str[n]cat calls are more difficult to handle since they have an accumulating effect on the size of the destination buffer during the course of the program. If calls to str[n]cat take place several times inside a program and the destination buffer is the same, one needs to keep track of the size of the buffer as it grows. This scenario occurs when str[n]cat calls take place inside loops; BOON assumes that a str[n]cat call inside a loop can be executed any number of times, and for that reason it gives up on trying to do a safety analysis and reports a message of the type "str[n]cat calls were not checked." This leaves the work of verifying the safety of these calls to the programmer.

6.1.2 Splint

Splint had a few quirks of its own too. As was mentioned earlier, sometimes Splint issued long lists of parse errors. Most of these errors were related to type definitions such uid_t, u_char, u_long, u_short, etc. Since the model programs were relatively short, these parse errors could be eliminated by supplying a sufficient number of "-D" flag options (e.g., "-Du_char = unsigned char"). It was sometimes not obvious from the Splint documentation what certain flags were meant to do; perhaps if an expert Splint user were to have conducted these same experiments, the number of warnings could have been diminished to some degree by supplying certain flags.

To analyze the model programs for Sendmail and WU-FTPD, Splint was run with the following command:

```
splint +bounds +posixlib -I/usr/include -I/usr/include/arpa -I/usr/include/sys -I. -
Du_char="unsigned char" -Du_int="unsigned int" -Du_short="unsigned short" -Du_long="unsigned long"
-Du_int16_t="unsigned short int" -Du_int32_t="unsigned long int" *.c
```

To analyze the model programs for BIND, Splint was run with the following command:

```
splint +posixlib +bounds -I. -DARBPTR_T=int* -Du_int8_t=unsigned char -Dint16_t=short
-Du_int16_t=unsigned short -Dint32_t=int -Du_int32_t=unsigned int -Du_int=unsigned int
-D__gnuc_va_list=void* -Du_char=unsigned char -Dlint -Du_long=unsigned long
-Din_addr_t=unsigned int -Du_short=unsigned short -deepbreak *.c
```

6.1.3 UNO

Although UNO did not detect any of the bugs in the model programs, it gave no false alarms either. UNO gave some useful warnings about side effects, such as having assignments inside conditions, e.g., "while ((c = fgetc(f)) != EOF)", or other side effects such as decrementing a pointer inside an if-statement, e.g., "if (*buf-- == 'a')". UNO also gave some warnings about fallthroughs in switch statements, which like the latter side effects can sometimes be difficult to spot and can be a cause for an error.

6.1.4 ARCHER

ARCHER was not too difficult to run, however there were a few problems. When trying to analyze an entire version of Sendmail, ARCHER terminated with a cryptic "divide by zero" bug. In a few other cases, ARCHER ran into problems during its analysis by exhausting its own stack space. On the plus side, ARCHER did not give too many false positives. ARCHER is still a research tool under development, and it has some bugs to be fixed.

6.2 False alarms per lines of code

Several observations were made about the number of false alarms that PolySpace and Splint generated for each of the "OK" programs, specifically the total number of array indexing and out-of-bounds warnings. Table 15 shows the number of lines of C code (not counting include files) for each of the "OK" programs and the number of array indexing errors and out-of-bounds warnings issued by PolySpace and Splint. Sometimes the program sizes differed slightly for PolySpace and Splint. This is because in a few cases it was necessary to make slight modifications to the code in order to get the tools to perform analysis. In three of the cases, namely the WU-FTPD vulnerabilities, the programs analyzed by PolySpace were considerably larger than the ones analyzed by Splint. As was explained earlier, this is because a separate function for each of the str[n]cpy and str[n]cat calls was inserted into the code to be able to distinguish between the different calls.

		PolySpace	Splint
		#warnings/L.O.C	#warnings/L.O.C
Sendmail	crackaddr	62/473	2/473
	gecos	31/374	14/376
	Mime1	16/256	3/253
	Mime2	30/269	2/271
	prescan	18/567	3/559
	tTflag	19/176	1/170
	TXT rec.	72/509	20/510
BIND	NXT bug	36/477	9/476
	SIG bug	58/567	10/544
	iquery	17/134	8/134
	nslookup	20/284	6/284
WU-FTPD	Mapped chdir	27/324	9/199
	Off-by-one	17/635	9/485
	realpath	77/1141	19/575
Average		1/11	1/50

Table 15.

As can be seen from the table above, both PolySpace and Splint generated many false alarms for the "OK" programs. The warnings for PolySpace that were tallied in the above table were orange warnings belonging to two classes: 1) OBAI (Array out-of-bounds warnings) or 2) IDP (Dereferenced pointer errors). The warnings that were tallied for Splint were "possible out-of-bounds store" and "possible out-of-bounds read." Most of the warnings generated can be assumed to be false alarms, however, it is possible that a few "out-of-bounds bugs" were introduced during the construction of the model programs. For each of the programs, the ratio of the number of warnings issued to the number of lines of code is shown. The bottom row shows the average over all fourteen programs. Using these averages as estimates of the frequency of false alarms per lines of code, we see that Splint gives about 1 false alarm for every fifty lines of code, and PolySpace gives about 1 one orange "out-of-bounds" warning per every 11 lines of code. Running PolySpace on an application like Sendmail, which is on the order of 50K lines of code, can potentially yield on the order of 10,000 out-of-bounds warnings, most of which would be false alarms! Similarly, Splint would generate about a 1000 false out-of-bounds warnings!

Chapter 7 Conclusion

Of five tools tested, only PolySpace and Splint were able to detect a significant fraction of the buffer overflows in the fourteen model programs. PolySpace had an average detection rate close to 90%, whereas Splint had an average detection rate around 60%. Both tools, however, reported a large number of false alarms. When analyzing the patched versions of the programs, PolySpace raised an average of one orange, out-of-bounds warning per 10 lines of code, whereas Splint raised an average of one out-of-bounds warning per roughly 50 lines of code.

One thing to realize is that PolySpace is really intended for usage during the development process, rather than for analyzing finished products. Ideally, a programmer would write a hundred or so lines of code and then run them through PolySpace and fix any reported bugs. If PolySpace is used in this manner, the number of false alarms would be manageable. Another thing to keep in mind when looking at our results is that the number of test programs used was relatively small. Had the number of test programs been higher, the results would have been more accurate. Despite the small number of programs analyzed, our results strongly suggest that more work needs to be done in creating tools that are able to analyze large legacy software without overwhelming the user with too many false alarms.

Static code analysis is a complex problem, and we are far from having conquered it. However, as more research is done in the area of static code analysis, the tools will get better and become more "intelligent". Hopefully, as the tools will improve, so will the quality of programs, and the number of buffer overflow related security advisories will diminish.

References

1) Advanced Buffer Overflows.
http://hysteria.sk/arxiv/hack/papers/advancedoverflows/

2) S.J. Ahmed, Securely Programming in C, SANS Information Security Reading Room, Sept. 24, 2002.
http://www.sans.org/rr/code/sec_c.php

3) Aleph One. Smashing The Stack for Fun and Profit. Phrack Magazine. Issue #49, November 1996.
http://destroy.net/machines/security/P49-14-Aleph-One

4) P.Anderson, T. Teitelbaum. Software Inspection Using CodeSurfer, Proceeding of the First Workshop on Inspection in Software Engineering, Paris, July, 2001.
http://www.grammatech.com/research/papers/AndersonTeitelbaum.pdf

5) K. Ashcraft, and D. Engler, Using Programmer-Written Compiler Extensions to Catch Security Holes. In Proceedings of IEEE Security and Privacy. 2002.
http://www.stanford.edu/~engler/sp-ieee-02.ps

6) Avaya Labs Research. Libsafe Project.
http://www.research.avayalabs.com/project/libsafe/

7) Ballista Tool
http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/edrc-ballista/www/

8) A. Baratloo, T. Tsai, and N. Singh. "Transparent Run-Time Defense Against Stack Smashing Attacks." Proceedings of the USENIX Annual Technical Conference, June 2000.

9) A. Baratloo, T. Tsai, and N. Singh. Libsafe: Protecting Critical Elements of Stacks. Bell Labs, Lucent Technologies, 600 Mountain Ave, Murray Hill, NJ 07974 USA
http://community.core-sdi.com/~juliano/libsafe.pdf

10) M. Bishop, and M.Dilger. Checking for Race Conditions in File Accesses. University of California at Davis. Davis, CA. 1996.
http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/ucd-ecs-95-09.pdf

11) BoundsChecker
http://www.compuware.com/products/devpartner/bounds.htm

12) H. t. Brugge. Bounds checking patch for gcc.
http://web.inter.nl.net/hcc/Haj.Ten.Brugge/

13) BugScan Inc.
http://www.bugscaninc.com/product.html

14) Bulba and Kil3r. Bypassing StackGuard and StackShield. Phrack Magazine 56, May 2000.
http://www.phrack.org/phrack/56/p56-0x05

15) W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. Software: Practice and Experience, 30(7):775-802, 2000.
http://osq.cs.berkeley.edu/public/Pincus-StaticAnalyzer.pdf

16) CERT Coordination Center
http://www.cert.org/advisories/

17) H. Chen, D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. 2002.
http://www.cs.berkeley.edu/~daw/mops/

18) Cigital Security. ITS4: Software Security Tool.
http://www.cigital.com/its4/

19) Code Red Worm. Virus Bulletin. Sept. 2001.
http://www.peterszor.com/codered.pdf

20) Computer Security Institute
http://www.gocsi.com/awareness/fbi.html

21) M. Conover, w00w00 on Heap Overflows, Jan. 1999.
http://www.w00w00.org/articles.html.

22) P. Cousot, R. Cousot. Static Determination of Dynamic Properties of Programs. Universite Scientifique ed Medicale de Grenoble. Proceedings of the 2nd international symposium of Programming. Paris, April 13-15 1976.
http://www.di.ens.fr/~cousot/COUSOTpapers/ISOP76.shtml

23) C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. DARPA Information Survivability Conference and Expo (DISCEX), Hilton Head Island SC, January 2000.
http://www.cse.ogi.edu/~crispin/

24) C. Cowan. Solar Designer's non-executable stack patch discussion. Dec. 9. 1997.
http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/usenixsc98_html/node21.html

25) C. Cowan, et al., StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. 7th USENIX Security Conf., Usenix Assoc., 1998, pp. 63-77.
http://www.immunix.org/StackGuard/usenixsc98.pdf

26) C. Cowan. Software Security for Open-Source Systems. In IEEE Security and Privacy. Jan/Feb. 2003. Requires subscription:
http://www.computer.org/security/

27) C++ Reference Page
http://www.cppreference.com/

28) Cyberattacks News Story
http://story.news.yahoo.com/news?tmpl=story&u=/cmp/20030716/tc_cmp/12800622

29) M. E. Donaldson, Inside the Buffer Overflow Attack: Mechanism, Method, & Prevention, SANS Institute Information Security Reading Room, April 3, 2002.
http://www.sans.org/rr/code/inside_buffer.php

30) Emsi vulnerability
http://www.immunix.org/StackGuard/emsi_vuln.html

31) Entercept
http://www.entercept.com/ricochet/bufferoverflows.asp

32) Heap overflow in Microsoft IIS
http://www.kb.cert.org/vuls/id/363715

33) H. Etoh. GCC extension for protecting applications from stack-smashing attacks.
http://www.trl.ibm.com/projects/security/ssp/, June 2000

34) D. Evans, D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. In IEEE SOFTWARE, Jan/Feb 2002, pages 42-51.
http://www.cs.virginia.edu/~evans/pubs/ieeesoftware.pdf
http://www.splint.org/pubs.html

35) D. Evans, D. Larochelle. Statically Detecting Likely Buffer Overflow Vulnerabilities. In 2001 USENIX Security Symposium, Washington, D.C., Aug 13-17, 2001.
http://lclint.cs.virginia.edu/usenix01.pdf

36) P-A. Fayolle, V. Glaume. A Buffer Overflow Study: Attacks and Defenses. ENSEIRB, Networks and Distributed Systems, 2002.
http://www.linux-tech.com/buff_over.html

37) FIST - Fault Injection Tool
A. Ghosh, T. O'Connor and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In Proceedings of the IEEE Symposium on Security and Privacy, pages 104-114, Oakland, CA, May 1998.
http://www.cigitallabs.com/resources/papers/download/ieees_p98_2col.ps

38) N. Frykholm. Countermeasures Against Buffer Overflow Attacks. Nov. 30, 2000.
http://www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.html

39) Bill Gates E-mail on Trustworthy Computing
http://paulboutin.weblogger.com/stories/storyReader$155

40) GDB – The GNU Project Debugger
http://sources.redhat.com/gdb/

41) R. Hastings, B. Joyce. Purify: Fast detection of memory leaks and access errors. In Proceedings of the Winter USENIX Conference, December 1992.
http://www.rational.com/products/purify_nt/index.jsp

42) Heap overflow in Microsoft IIS
http://www.kb.cert.org/vuls/id/363715

43) G. J. Holzmann. Uno: Static Source Code Checking for User-Defined Properties. Bell Laboratories. Murray Hill, New Jersey 07974
http://cm.bell-labs.com/cm/cs/what/uno/

44) M.Howard, D. LeBlanc. Writing Secure Code. Second Edition, Microsoft Press, 2003.

45) ICAT Security Vulnerability Database
http://icat.nist.gov/icat.cfm

46) IDA and gprof
http://www.ida.liu.se/~vaden/cgdi/

47) T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, Y.Wang. Cyclone: A safe dialect of C.
http://www.research.att.com/projects/cyclone/papers/cyclone-safety.pdf

48) R. Jones, P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In Automated and Algorithmic Debugging, pages 13-26, 1997.
http://www.doc.ic.ac.uk/~phjk/Publications/BoundsCheckingForC.ps.gz

49) M. Kaempf (MaXX). Vudo malloc tricks. Phrack magazine. Volume 0x0b, Issue 0x39, 2001.
http://www.phrack.org

50) Kaksonen, Rauli. A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis). Espoo. Technical Research Centre of Finland, VTT Publications 447, 2001.
http://www.inf.vtt.fi/pdf/publications/2001/P448.pdf
PROTOS Project: http://www.ee.oulu.fi/research/ouspg/protos/

51) klog. The Frame Pointer Overwrite.
http://www.phrack.org/show.php?p=55&a=8

52) Ballista Paper
N.P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In the 28th International Symposium on Fault-Tolerant Computing, pages 464-468, 1998.

53) T. La, Secure Software Development and Code Analysis Tools, SANS Information Security Reading Room, Sept. 30, 2002.
http://www.sans.org/rr/code/tools.php

54) D. LeBlanc. Avoiding Buffer Overruns with String Safety.
http://community.core-sdi.com/~juliano/leblanc-nt-avoidbof.html

55) R. Lemos. Open-source team fights buffer overflows. CNET News.com Apr 11, 2003.
http://news.com.com/2100-1002-996584.html

56) D. Litchfield. Non-stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP, March 2002.
http://www.0x36.org/PAPERS/BUFFER/non-stack-bo-windows.pdf

57) L0pht Heavy Industries. SLINT - source code security analyzer.
http://www.l0pht.com/slint.html

58) G. McGary. Bounds checking projects
http://gcc.gnu.org/projects/bp/main.html

59) G. McGraw and J. Viega. Making your software behave: Preventing buffer overflows. Protect your code through defensive programming. Reliable Software Technologies. March 7, 2000
http://community.core-sdi.com/~juliano/gmjv-prevent.pdf

60) G. McGraw and J. Viega. Making your software behave: Learning the basics of buffer overflows. Get reacquainted with the single biggest threat to software security.
http://www-106.ibm.com/developerworks/security/library/s-overflows/

61) Mib Software. Libmib allocated string functions.
http://www.mibsoftware.com/libmib/astring/

62) Microsoft's C/C++ \GS compiler option + exploit to bypass it
http://www.cigital.com/news/index.php?pg=art&artid=70

63) Microsoft's Reliability Group
http://research.microsoft.com/reliability/

64) B. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM, 33(12):32-44, 1990.

65) G.C. Necula, S. McPeak, and W.Weimer. CCured: type-safe retrofitting of legacy code. In Symposium of Principles of Programming Languages, pages 128-139, 2002.
http://raw.cs.berkeley.edu/Papers/ccured_popl02.pdf

66) Off-by-one Buffer Overflow. Sept. 9. 1999.
http://www.phrack.com/phrack/55/P55-08

67) PC-Lint/Flexe-Lint (by Gimpel Software)
http://www.gimpel.com/html/lintinfo.htm

68) OpenWall Project
Linux kernel patch from the Openwall project - http://www.openwall.com/linux/

69) PaX Project
http://pageexec.virtualave.net/

70) PolySpace
http://www.polyspace.com

71) PREfast Whitepaper
http://www.microsoft.com/whdc/hwdev/driver/PREfast.mspx

72) ProPolice
http://www.trl.ibm.com/projects/security/ssp/

73) G. Richarte, Four different tricks to bypass StackShield and StackGuard protection, Core Security Technologies, April 9, 2002 - April 24, 2002
http://ducer.w00nf.org/trash/security/advanced/gerastackguard.pdf

74) Secure Software, RATS scanning tool.
http://www.securesoftware.com/download_form_rats.htm

75) D.Seeley. A Tour of the Worm. Department of Computer Science, University of Utah.
http://world.std.com/~franl/worm.html
http://www.goldinc.com/html/maloy/SECURITY/morris_worm.html

76) U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In Proceedings of the 10th USENIX Security Symposium, 2001.
http://citeseer.nj.nec.com/shankar01detecting.html

77) Solar Designer. /usr/bin/lpr buffer overflow exploit for Linux with non-executable stack. 1997.
http://hysteria.sk/arxiv/hack/papers/advancedoverflows/ret-libc.txt

78) StackShield
http://www.angelfire.com/sk/stackshield/index.html

79) StrSafe
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/resources/strings/usingstrsafefunctions.asp

80) A. Takanen, M. Laakso, J. Eronen, J. Roening. Running Malicious Code by Exploiting Buffer Overflows: A Survey of Publicly Available Exploits. University of Oulu, Department of Electrical Engineering, Finland. 2000.
http://www.ee.oulu.fi/research/ouspg/protos/sota/EICAR2000-overflow-survey/paper.pdf

81) J. Viega, J.T. Bloch, Y. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In Annual Computer Security Applications Conference, 2000.
http://www.rstcorp.com

82) J. Viega, G. McGraw. Building Secure Software: How to Avoid Security Problems the Right Way.
http://www.buildingsecuresoftware.com/

83) D.Wagner, J.Foster, E.Brewer, and A.Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Proceedings of the Year 2000 Network and Distributed System Security Symposium (NDSS), pages 3-17, San Diego, CA, February 2000.
http://citeseer.nj.nec.com/wagner00first.html

84) D. Wheeler, FlawFinder Tool.
http://www.dwheeler.com/flawfinder/

85) J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. Department of Computer and Information Science, Linkoepings universitet.
http://www.ida.liu.se/~johwi

86) J. Wilander and M.Kamkar. A Comparison of Publicly Available Tools for Static Intrusion Prevention. Department of Computer and Information Science, Linkoepings universitet.
http://www.ida.liu.se/~johwi

87) R. Wojtczuk (a.k.a Nergal). The advanced return-into-lib(c) exploits. Phrack Magazine, Vol 11, Issue 58. Dec. 2001.
http://www.phrack.org

88) R. Wojtczuk. Defeating Solar Designer non-executable stack patch. Jan 30. 1998.
http://www.insecure.org/sploits/non-executable.stack.problems.html

89) Y. Xie, A. Chou, and D. Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. Computer Systems Laboratory. Stanford University. 2002
http://www.stanford.edu/~engler/p150-xie.pdf

90) Xsun Heap overflow
http://www.securityfocus.com/archive/1/265370

91) Y2K Bug
http://www.pbs.org/newshour/bb/cyberspace/jan-june98/y2000_6-11.html

APPENDIX A – Common Security Vulnerabilities

A.1 Array Indexing Errors

Indexing an array with an index that exceeds the maximum array index or with a negative index, can lead to serious problems. Whenever one accesses an array through an untrusted index, the bounds should be checked. Accessing a memory location outside the process' memory space will cause the system to crash.

A.2 Buffer Overflows

This is the main class of vulnerabilities that we would like our tools to detect. This vulnerability is discussed in detail in the thesis.

A.3 Dereferencing of Null Pointers or Corrupted Pointers

Often a program can be caused to crash as a result of dereferencing null pointers. Ideally, a tool should detect potential dereferencing of null pointers and should issue an appropriate warning. Also, it is possible to create corrupted pointers by adding offsets to null pointers. Having pointers refer to objects that have been freed using free() or realloc() can also cause serious problems.

A.4 Format String Vulnerabilities [76]

This is one of the more recently discovered types of bugs. These bugs involve idiosyncrasies of printf-type functions, which are common in calls like printf("Hello %s", username). Error logging calls are another common source of format string bugs(e.g., syslog()). If the number of arguments does not correspond to the number of format string parameters (e.g., %s, %d etc), it is possible to overwrite arbitrary locations in memory with supplied values. The underlying mechanism for format string exploits relies on the not-so-common %n option, that allows one to write the number of printed characters into a location specified by an argument.

A.5 Integer Overflows

In C, adding one to the maximum allowed integer, MAX_INT, results in an integer overflow and yields 0.

Consider the following kernel code below [5] :

```
err = copy_from_user(&input, arg, sizeof(input));
...
input.path = kmalloc(input.path_len + 1, GFP_KERNEL);

if(!input.path)
   return -ENOMEM;
```

```
error = copy_from_user(input.path, user_path, input.path_len);
```

If path_len = MAX_INT in the above code, then kmalloc(0, GFP_KERNEL) will return a non-nil pointer to a small amount of kernel memory. The subsequent copy_from_user call would copy MAX_INT (usually around 4 gigabytes) bytes into a small space, corrupting large amounts of kernel memory.

A.6 Integer Underflows

Just like it is possible to overflow an unsigned integer by exceeding MAX_INT, it is possible to cause a wrap-around by subtracting one from an unsigned integer equal to 0.

Consider the following code segment [44, p.624]:

```
void AllocMemory(size_t cbAllocSize)
{
   //We don't accomodate for trailing '\0'
    cbAllocSize--;
   char *szData = malloc(cbAllocSize);
...
}
```

If cbAllocSize is passed in as 0, cbAllocSize-- will cause an underflow, and szData will be equal to malloc(MAX_INT), or a pointer to a chunk of approx., 4 billion bytes!

A.7 Privilege Vulnerabilities

Many security vulnerabilities result if a process runs with root privileges, but fails to drop privileges when necessary. Consider the following chunk of code [17].

```
int main(int argc, char *argv[])
    {  // start with root privilege
m0:   do_something_with_privilege();
m1:   drop_privilege();
m2:   execl("/bin/sh", "/bin/sh", NULL);          //risky syscall
m3: }

void drop_privilege()
    {
      struct passwd *passwd;

d0:   if  ((passwd = getpwuid(getuid())) == NULL)
d1:       return; // but forget to drop privilege!
d2:   fprintf(log, "drop priv for %s", passwd->pw_name);
d3:  seteuid(getuid());       //drop privilege
d4: }
```

The example code above illustrates the problem of forgetting to drop privileges. In the above code, the path [m1d0d2d3d4m2m3] correctly drops privileges, however, the path [m1d0d1m2m3] fails do drop privileges.

To detect this kind of vulnerability, a tool with good interprocedural, path-sensitive analysis would be required.

Another dangerous vulnerability involves a process running with non-root privileges and illegally acquiring root privileges. See the description of truncation errors for an example of such a vulnerability [Appendix B.5].

A.8 Race Conditions

There exists a whole class of security flaws known as Time-Of-Check-To-Time-Of-Use (TOCTTOU) flaws, as described originally by Bishop and Dilger [10]. Basically, a TOCTTOU flaw occurs when a program checks for a particular property of an object, and then takes some action later on that assumes that the property still holds when in fact it does not.

Consider the following code: [10]

```
if (access(filename, W_OK) == 0) {
   if ((fd = open(filename, O_WRONLY)) == NULL) {
      perror(filename);
      return(0);
   }
   /* now write to file */
```

This code segment first checks to see if the file is accessible by the user, then opens it for writing, and then writes to it.

Suppose initially the filename = "/tmp/X", a temporary file accessible by anyone. So, the first access check is passed successfully. Now, imagine that in between the time that the first access check is made and the time that the file descriptor fd is created, an attacker deletes the file "/tmp/X" and creates a hard link named "/tmp/X" that refers to the file "etc/passwd". Now, when the file "/tmp/X" is opened for writing, the password file will be opened for writing, and the attacker will be able to write arbitrary data to the protected password file.

These types of flaws are also commonly known as race conditions, referring to the attacker's race with the program to try to invalidate the program's assumptions about a particular object before the program takes the next action on the object.

APPENDIX B – Dangerous Programming Errors

B.1 ANSI <-> UNICODE BUGS

A serious problem can result when a programmer mixes up the number of elements of a Unicode buffer with the size of the buffer in bytes. A commonly used function on Windows platforms that is vulnerable to this type of a bug is MultiByteToWideChar. This function maps a multi-byte character buffer to a wide character buffer. Here is a chunk of code illustrating incorrect usage of MultiByteToWideChar [44, p. 153]:

```
BOOL GetName(char *szName)
{
   WCHAR wszUserName[256];

   MultiByteToWideChar(CP_ACP, 0, szName, -1, wszUserName,
   sizeof(wszUserName));
... }
```

The problem lies in the last size argument. Since wszUserName is a wide character buffer, each character actually takes up two bytes. The last argument of MultiByteToWideChar should specify the number of elements of wszUserName, and should thus be sizeof(wszUserName) /2, or more generally sizeof(wszUserName)/sizeof(wszUserName[0]) (instead of sizeof(wszUserName)). Due to the above mistake, the function MultiByteToWideChar believes wszUserName to be capable of storing more elements than it is actually allocated to store. Since wszUserName is stored on the stack, a buffer overflow is very likely to occur here.

B.2 Failure to check array bounds

A programmer may be read or write from/to a buffer, but forget to check the bounds. The buffer may be accessed via an index, e.g., buf[i], where the index may exceed the upper bound of the buffer, or it may even be a negative value. The indexing may involve arithmetical expressions as well (e.g., buf[i+j]). The buffer may also be accessed via a pointer expressions (i.e., *++buf, *(buf + i)).

This is a very common programmer error that can lead to serious vulnerabilities. Reading from an illegal memory location usually causes the system to crash. By writing to an illegal memory location an attacker may successfully overwrite program parameters or function pointers. By overwriting a function pointer it might be possible to execute arbitrary code.

B.3 Off-By-One bugs

These are often careless bugs where a programmer forgets to account for a null terminator character. An example of an off-by-one bug is the pair of calls shown below:

```
strncpy(buf, in, sizeof(buf));
buf[sizeof(buf)] = '\0';
```

The correct calls should be:

```
strncpy(buf, in, sizeof(buf) - 1);
buf[sizeof(buf) - 1] = '\0';
```

As a result of the off-by-one bug shown above (if the size of the buffer is divisible by 4), it is possible to overwrite the lower order byte of the saved stack frame pointer with a null byte. This can be used to transfer execution control to the attacker.

Another off-by-one bug occurs in the case of misusing strncat:

The safe way to use strncat is to say:

strncat(buffA, buffB, sizeof(buffA) - strlen(buffA) - 1),

where the -1 is absolutely necessary. The third argument of strncat specifies the maximum number of characters to be concatenated onto buffA. strncat, unlike strncpy, always null terminates the destination buffer. This must be accounted for in the size argument.

If the strncat call had instead been:

strncat(buffA, buffB, sizeof(buffA) - strlen(buffA)),

a null terminator byte would get written into the location adjacent to the destination buffer. Again, if this location is the saved frame pointer, an exploit can be constructed.

Two well known off-by-one bugs appeared in Apache's mod-ssl and wu_ftpd's glob [44, p.138].

B.4 Signed to Unsigned Casts

A buffer overflow can occur as a result of incorrect casting of size fields from signed to unsigned types. Consider the following snippet of code [44, p.620]:

```
int Exampe(char* str, int size)
{
  char buf[80];

  if(size < sizeof(buf))
  {
```

```
    strcpy(buf, str);
    }
}
```

Here the size of str is supplied as an argument to the function. By default, C's int type describes a signed integer. Suppose, the caller of the Example function managed to pass in a negative size. Since sizeof(buf) always returns an integer of type size_t, equivalent to an unsigned integer, the above check would pass, and str would get copied into buf, potentially causing a buffer overflow. Of course, an attacker could also pass in a fake positive size field that satisfies (size < sizeof(buf)). Buffer sizes passed as function arguments should thus be always closely scrutinized and should not be trusted.

B.5 Truncation Errors

On a 32-bit operating system, values exceeding 32 bits, such as 0x100000000, get truncated to 0x00000000. These types of truncations can be a source of many security problems.

A famous truncation related exploit involves gaining superuser privileges. On a Unix system, the root user (superuser) has a user ID of 0. The network file system daemon would accept a signed integer value as a user ID, check to see if it's non-zero, and then truncate it to a four-byte unsigned integer. Now, suppose an attacker passed in 0x10000 as their user ID. This value would get truncated to 0x0000, effectively granting the attacker root privileges [44, p.147].

Also, because of truncation and as a result of an improper array index, it is possible for an attacker to write to a memory location at an address lower than the base of the array [44, p.146].

The formula for calculating the location of an array element is:
address of array element = base of array + index * sizeof(element).

Suppose base of array = 0x00510048. Suppose the array stores long integers, i.e sizeof(element) = 4 bytes.

Then, array[0x3FF07FCF] is located at 0x00510048 + 0x3FF07FCF*4 = 0x10012FF84.

However, on a 32-bit system, 0x10012FF84 gets truncated to 0x0012FF84. So, it is actually possible to access an address below the base of the array by using an invalid index. Thus, through invalid indexing combined with truncation, an attacker could write to an arbitrary location in memory.

APPENDIX C – Websites for obtaining source code for retrospective analysis + links to vulnerabilities

The following list contains websites where one can obtain source code for old versions of BIND, WU-FTPD, and Sendmail.

C.1 BIND

Latest Version: 9.2.2, Released Mar 3, 2003 - http://www.isc.org/products/BIND/
Versions 4.8-4.9.8, 8.1-8.2.2-P7 can be found at the website below:
ftp://ftp.isc.org/isc/bind/src/DEPRECATED/

Vulnerabilities:

nslookupComplain overflow & TSIG overflow
http://www.cert.org/advisories/CA-2001-02.html

SIG Bug & NXT Bug
http://www.cert.org/advisories/CA-1999-14.html

Iquery Bug
http://www.cert.org/advisories/CA-98.05.bind_problems.html
http://icat.nist.gov/icat.cfm?cvename=cve-1999-0009

C.2 WU_FTPD

Versions 2.0-2.7.0 can be found here:
ftp://ftp.wu-ftpd.org/pub/wu-ftpd-attic/

Vulnerabilities:

off-by-one overflow
http://www.kb.cert.org/vuls/id/743092

mapped chdir overflow
http://www.cert.org/advisories/CA-1999-13.html
http://icat.nist.gov/icat.cfm?cvename=cve-1999-0878

realpath overflow
http://www.cert.org/advisories/CA-1999-03.html
http://icat.nist.gov/icat.cfm?cvename=cve-1999-0368

C.3 Sendmail

Current Release 8.12.9
Many old versions can be found here:
http://www.sendmail.org/ftp/
http://www.sendmail.org/ftp/past-releases/

Version 8.7.5 can be obtained here:
http://www.mit.edu/afs/net/project/attic/sendmail8/

Vulnerabilities:

Remote Header Processing Vulnerability
http://www.cert.org/advisories/CA-2003-07.html

Gecos Overflow
http://icat.nist.gov/icat.cfm?cvename=cve-1999-0131

MIME 8.8.0/8.8.1 overflow
http://icat.nist.gov/icat.cfm?cvename=cve-1999-0206

MIME 8.8.3/8.8.4 overflow
http://icat.nist.gov/icat.cfm?cvename=cve-1999-0047

prescan overflow
http://www.cert.org/advisories/CA-2003-12.html

tTflag overflow
http://icat.nist.gov/icat.cfm?cvename=cve-2001-0653

TXT record overflow
http://xforce.iss.net/xforce/xfdb/9443

APPENDIX D - Model Program (Sendmail remote header vulnerability)

PRESCAN-BAD.c

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* states and character types */
#define OPR         0     /* operator */
#define ATM         1     /* atom */
#define QST         2     /* in quoted string */
#define SPC         3     /* chewing up spaces */
#define ONE         4     /* pick up one character */
#define ILL         5     /* illegal character */

#define NSTATES 6     /* number of states */

#define MAXNAME       40          /* max length of a name */
#define MAXATOM       10          /* max atoms per address */
#define PSBUFSIZE     (MAXNAME + MAXATOM)

#define MAXMACROID     0377         /* max macro id number */

#define TYPE          017    /* mask to select state type */

/* meta bits for table */
#define M           020    /* meta character; don't pass through */
#define B           040    /* cause a break */
#define MB          M|B    /* meta-break */

#define TRUE 1
#define FALSE 0

#define tTd(flag, level)      (tTdvect[flag] >= (u_char)level)
#define tTdlevel(flag)        (tTdvect[flag])

/* variables */
extern u_char  tTdvect[100];  /* trace vector */

typedef int bool;

#ifndef SIZE_T
# define SIZE_T       size_t
#endif /* ! SIZE_T */

typedef struct envelope ENVELOPE;

ENVELOPE   *CurEnv;    /* envelope currently being processed */

/* This is a very simplified representation of the envelope structure of Sendmai
l */
struct envelope
{
     char          *e_to;        /* the target person */
     ENVELOPE      *e_parent;    /* the message this one encloses */
     char          *e_macro[MAXMACROID + 1]; /* macro definitions */
};

/* Simplified address struct */
struct address
{
     char          *q_paddr;     /* the printname for the address */
     char          *q_user;      /* user name */
     char          *q_ruser;     /* real user name, or NULL if q_user */
     char          *q_host;      /* host name */
```

```
    char        *q_home;      /* home dir (local mailer only) */
    char        *q_fullname;  /* full name if known */
    struct address  *q_next;      /* chain */
    struct address  *q_alias;     /* address this results from */
    char        *q_owner;     /* owner of q_alias */
    struct address  *q_tchain;    /* temporary use chain */
    char        *q_orcpt;     /* ORCPT parameter from RCPT TO: line */
    char        *q_status;    /* status code */
        char        *q_rstatus;   /* remote status message for DSNs */
    char        *q_statmta;   /* MTA generating q_rstatus */
    short       q_state;      /* address state, see below */
    short       q_specificity; /* how "specific" this address is */
};

typedef struct address ADDRESS;


static short StateTab[NSTATES][NSTATES] =
{
  /*  oldst  chtype> OPR    ATM    QST    SPC    ONE    ILL    */
    /*OPR*/ {     OPR|B,  ATM|B,  QST|B,  SPC|MB, ONE|B,  ILL|MB  },
    /*ATM*/ {     OPR|B,  ATM,    QST|B,  SPC|MB, ONE|B,  ILL|MB  },
    /*QST*/ {     QST,    QST,    OPR,    QST,    QST,    QST     },
    /*SPC*/ {     OPR,    ATM,    QST,    SPC|M,  ONE,    ILL|MB  },
    /*ONE*/ {     OPR,    OPR,    OPR,    OPR,    OPR,    ILL|MB  },
    /*ILL*/ {     OPR|B,  ATM|B,  QST|B,  SPC|MB, ONE|B,  ILL|M   },
};

/* token type table -- it gets modified with $o characters */
static u_char   TokTypeTab[256] =
{
  /*  nul soh stx etx eot enq ack bel  bs  ht  nl  vt  np  cr  so  si   */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,SPC,SPC,SPC,SPC,SPC,ATM,ATM,
  /*  dle dc1 dc2 dc3 dc4 nak syn etb  can em  sub esc fs  gs  rs  us   */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  sp  !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /    */
     SPC,ATM,QST,ATM,ATM,ATM,ATM,ATM, SPC,SPC,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~  del   */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,

  /*  nul soh stx etx eot enq ack bel  bs  ht  nl  vt  np  cr  so  si   */
     OPR,OPR,ONE,OPR,OPR,OPR,OPR,OPR, OPR,OPR,OPR,OPR,OPR,OPR,OPR,OPR,
  /*  dle dc1 dc2 dc3 dc4 nak syn etb  can em  sub esc fs  gs  rs  us   */
     OPR,OPR,OPR,ONE,ONE,ONE,OPR,OPR, OPR,OPR,OPR,OPR,OPR,OPR,OPR,OPR,
  /*  sp  !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~  del   */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
};

/* token type table: don't strip comments */
u_char  TokTypeNoC[256] =
{
  /*  nul soh stx etx eot enq ack bel  bs  ht  nl  vt  np  cr  so  si   */
```

```
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,SPC,SPC,SPC,SPC,SPC,ATM,ATM,
  /* dle dc1 dc2 dc3 dc4 nak syn etb  can em  sub esc fs  gs  rs  us  */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /* sp  !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /  */
    SPC,ATM,QST,ATM,ATM,ATM,ATM,ATM, OPR,OPR,ATM,ATM,ATM,ATM,ATM,ATM,
  /* 0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?  */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /* @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /* P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _  */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /* `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /* p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~  del */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,

  /* nul soh stx etx eot enq ack bel  bs  ht  nl  vt  np  cr  so  si  */
    OPR,OPR,ONE,OPR,OPR,OPR,OPR,OPR, OPR,OPR,OPR,OPR,OPR,OPR,OPR,OPR,
  /* dle dc1 dc2 dc3 dc4 nak syn etb  can em  sub esc fs  gs  rs  us  */
    OPR,OPR,OPR,ONE,ONE,ONE,OPR,OPR, OPR,OPR,OPR,OPR,OPR,OPR,OPR,OPR,
  /* sp  !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /  */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /* 0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?  */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /* @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /* P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _  */
    \ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /* `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /* p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~  del */
    ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM.ATM,
};

#define NOCHAR      -1    /* signal nothing in lookahead token */

#define DELIMCHARS    "()<>,;\r\n"   /* default word delimiters */

char   *OperatorChars; /* operators (old $o macro) */
int    ConfigLevel;   /* config file level */


/*
**  PRESCAN -- Prescan name and make it canonical
**
**    Scans a name and turns it into a set of tokens.  This process
**    deletes blanks and comments (in parentheses) (if the token type
**    for left paren is SPC).
**
**    This routine knows about quoted strings and angle brackets.
**
**    There are certain subtleties to this routine.  The one that
**    comes to mind now is that backslashes on the ends of names
**    are silently stripped off; this is intentional.  The problem
**    is that some versions of sndmsg (like at LBL) set the kill
**    character to something other than @ when reading addresses;
**    so people type "csvax.eric\@berkeley" -- which screws up the
**    berknet mailer.
**
**    Parameters:
**        addr -- the name to chomp.
**        delim -- the delimiter for the address, normally
**            '\0' or ','; \0 is accepted in any case.
**            If '\t' then we are reading the .cf file.
**        pvpbuf -- place to put the saved text -- note that
**            the pointers are static.
**        pvpbsize -- size of pvpbuf.
**        delimptr -- if non-NULL, set to the location of the
**            terminating delimiter.
**        toktab -- if set, a token table to use for parsing.
```

```
**                If NULL, use the default table.
**
**     Returns:
**          A pointer to a vector of tokens.
**          NULL on error.
*/

char **
prescan(addr, delim, pvpbuf, pvpbsize, delimptr, toktab)
     char *addr;
     int delim;
     char pvpbuf[];
     int pvpbsize;
     char **delimptr;
     u_char *toktab;
{
     register char *p;
     register char *q;
     register int c;

     bool bslashmode;
     bool route_syntax;
     int cmntcnt;
     int anglecnt;
     char *tok;
     int state;
     int newstate;
     char *saveto = CurEnv->e_to;
     /*static char *av[MAXATOM + 1]; */
     /*static char firsttime = FALSE; */
     int errno;


     printf("Inside prescan!!\n");
     printf("Max storage of pvpbuf = %d\n", PSBUFSIZE);


     if (toktab == NULL)
          toktab = TokTypeTab;

     /* make sure error messages don't have garbage on them */
     errno = 0;

     q = pvpbuf;
     bslashmode = FALSE;
     route_syntax = FALSE;
     cmntcnt = 0;
     anglecnt = 0;

     /* avp = av; */

     state = ATM;
     c = NOCHAR;
     p = addr;
     CurEnv->e_to = p;

     do
     {
          /* read a token */
          tok = q;
          for (;;)
          {
               /* store away any old lookahead character */
               if (c != NOCHAR && !bslashmode)
               {
                    /* see if there is room */

                    if (q >= &pvpbuf[pvpbsize - 5])
                    {
                     printf("553 5.1.1 Address too long\n");
```

```
        if (strlen(addr) > (SIZE_T) MAXNAME)
          {
           printf("strlen(addr) > %d\n", MAXNAME);
           addr[MAXNAME] = '\0';
          }
          returnnull:
            if (delimptr != NULL)
              *delimptr = p;
            CurEnv->e_to = saveto;
            return NULL;
      }

      /* squirrel it away */
          printf("Writing %c to q!\n", c);
      *q++ = c;
}

/* read a new input character */
c = *p++;
if (c == '\0')
{
      /* diagnose and patch up bad syntax */
      if (state == QST)
      {
            printf("653 Unbalanced '\"'");
            c = '"';
      }
      else if (cmntcnt > 0)
      {
            printf("653 Unbalanced '('");
            c = ')';
      }
      else if (anglecnt > 0)
      {
            c = '>';
            printf("653 Unbalanced '<'");
      }
      else
            break;

      p--;
}
else if (c == delim && cmntcnt <= 0 && state != QST)
{
      if (anglecnt <= 0)
            break;

      /* special case for better error management */
      if (delim == ',' && !route_syntax)
          {
            printf("653 Unbalanced '<'");
            c = '>';
            p--;
      }
}

/* chew up special characters */

/*BAD*/
*q = '\0';

if (bslashmode)
{

 /*printf("bslashmode = TRUE!!!\n");*/

      bslashmode = FALSE;
```

```
        /* kludge \! for naive users */
        if (cmntcnt > 0)
        {
            c = NOCHAR;
            continue;
        }
        else if (c != '!' || state == QST)
        {

         printf("Writing slash to q!!!!\n");
         /*BAD*/
         *q++ = '\\';
         continue;  /* continue while loop */
        }
    }

    if (c == '\\')
    {
        bslashmode = TRUE;
    }
    else if (state == QST)
    {
        /* EMPTY */
        /* do nothing, just avoid next clauses */
    }
    else if (c == '(' && toktab['('] == SPC)
    {
        cmntcnt++;
        c = NOCHAR;
    }
    else if (c == ')' && toktab['('] == SPC)
    {
        if (cmntcnt <= 0)
        {
            printf("653 Unbalanced ')'");
            c = NOCHAR;
        }
        else
            cmntcnt--;
    }
    else if (cmntcnt > 0)
    {
        c = NOCHAR;
    }
    else if (c == '<')
    {
        char *ptr = p;

        anglecnt++;
        while (isascii((int)*ptr) && isspace((int)*ptr))
            ptr++;
        if (*ptr == '@')
            route_syntax = TRUE;
}
    else if (c == '>')
    {
        if (anglecnt <= 0)
        {
            printf("653 Unbalanced '>'");
            c = NOCHAR;
        }
        else{
          anglecnt--;
        }
        route_syntax = FALSE;
    }
    else if (dclim == ' ' && isascii(c) && isspace(c))
        c = ' ';
```

```
                if (c == NOCHAR){
                  printf("c = NOCHAR.... continuing....!\n");
                      continue;
                }

                /* see if this is end of input */
                if (c == delim && anglecnt <= 0 && state != QST)
                  {
                    printf("breaking from for loop!\n");
                  break;
                  }

                 newstate = StateTab[state][toktab[c & 0xff]];

                state = newstate & TYPE;
                if (state == ILL)
                {
                      if (isascii(c) && isprint(c))
                            printf("653 Illegal character %c", c);
                      else
                           printf("653 Illegal character 0x%02x", c);
                }
                /* if (bitset(M, newstate)) */
                if (newstate & M)               /* replacement for bit set */ {
                  c = NOCHAR;
                }
                /* if (bitset(B, newstate)) */
                if (newstate & B)
                  {
                    break;
                  }
          }

          if (tok != q)
          {
                printf("writing null byte\n");
                /*BAD*/
                *q++ = '\0';

                if (q - tok > MAXNAME)
                {
                      printf("553 5.1.0 prescan: token too long");
                      goto returnnull;
                }
          }

     } while (c != '\0' && (c != delim || anglecnt > 0));

     printf("Exiting while loop!\n");
            p--;
     if (delimptr != NULL)
          *delimptr = p;

     CurEnv->e_to = saveto;

     return NULL;
}

char **
parseaddr(addr, delim, delimptr)
     char *addr;
     int delim;
     char **delimptr;
{

  register char **pvp;
  char pvpbuf[PSBUFSIZE];
  static char *delimptrbuf;
```

```
 /*** Initialize and prescan address. */


 if (delimptr == NULL)
   delimptr = &delimptrbuf;

 pvp = prescan(addr, delim, pvpbuf, sizeof pvpbuf, delimptr, NULL);

 return pvp;

}

int main(){

 char *addr;
 int delim;

 static char **delimptr;
 char special_char = '\377';  /* same char as 0xff.  this char will get interpreted as NOCHAR */
 int i = 0;

 addr = (char *) malloc(sizeof(char) * 500);
 /* This address is valid */
 /* strcpy(addr, "Misha Zitser <misha@mit.edu>"); */


 /* This address causes a buffer overflow and results in a seg fault */
 /* create malicious address... */

 for(i=0; i<300; i=i+2){
   addr[i] = '\\';
   addr[i+1] = special_char;
 }

 delim = '\0';
 delimptr = NULL;

 OperatorChars = NULL;

 ConfigLevel = 5;

 CurEnv = (ENVELOPE *) malloc(sizeof(struct envelope));
 addr = (char *) malloc(sizeof(char) * 500);
 /* This address is valid */
 /* strcpy(addr, "Misha Zitser <misha@mit.edu>"); */


 for(i=0; i<300; i=i+2){
   addr[i] = '\\';
   addr[i+1] = special_char;
 }

 delim = '\0';
 delimptr = NULL;

 OperatorChars = NULL;

 ConfigLevel = 5;

 CurEnv = (ENVELOPE *) malloc(sizeof(struct envelope));
 CurEnv->e_to = (char *) malloc(strlen(addr) * sizeof(char) + 1);

 strcpy(CurEnv->e_to, addr);

 parseaddr(addr, delim, delimptr);


 return 0;
}
```

PRESCAN-OK.c

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* states and character types */
#define OPR         0     /* operator */
#define ATM         1     /* atom */
#define QST         2     /* in quoted string */
#define SPC         3     /* chewing up spaces */
#define ONE         4     /* pick up one character */
#define ILL         5     /* illegal character */

#define NSTATES 6     /* number of states */

#define MAXNAME       40          /* max length of a name */
#define MAXATOM       10          /* max atoms per address */
#define PSBUFSIZE     (MAXNAME + MAXATOM)

#define MAXMACROID      0377          /* max macro id number */

#define TYPE          017     /* mask to select state type */
/* meta bits for table */
#define M           020     /* meta character; don't pass through */
#define B           040     /* cause a break */
#define MB          M|B     /* meta-break */

#define TRUE 1
#define FALSE 0

#define tTd(flag, level)      (tTdvect[flag] >= (u_char)level)
#define tTdlevel(flag)        (tTdvect[flag])

/* variables */
extern u_char   tTdvect[100];  /* trace vector */

typedef int bool;

#ifndef SIZE_T
# define SIZE_T       size_t
#endif /* ! SIZE_T */

typedef struct envelope ENVELOPE;
ENVELOPE   *CurEnv;    /* envelope currently being processed */

/* This is a very simplified representation of the envelope structure of Sendmai
l */
struct envelope
{
     char          *e_to;         /* the target person */
     ENVELOPE      *e_parent;     /* the message this one encloses */
     char          *e_macro[MAXMACROID + 1]; /* macro definitions */
};

/* Simplified address struct */
struct address
{
     char          *q_paddr;      /* the printname for the address */
     char          *q_user;       /* user name */
     char          *q_ruser;      /* real user name, or NULL if q_user */
     char          *q_host;       /* host name */
     char          *q_home;       /* home dir (local mailer only) */
     char          *q_fullname;   /* full name if known */
```

```
    struct address  *q_next;      /* chain */
    struct address  *q_alias;     /* address this results from */
    char        *q_owner;     /* owner of q_alias */
    struct address  *q_tchain;    /* temporary use chain */
    char        *q_orcpt;     /* ORCPT parameter from RCPT TO: line */
    char        *q_status;    /* status code for DSNs */
    char        *q_rstatus;   /* remote status message for DSNs */
    char        *q_statmta;   /* MTA generating q_rstatus */
    short       q_state;      /* address state, see below */
    short       q_specificity; /* how "specific" this address is */
};

typedef struct address ADDRESS;


static short StateTab[NSTATES][NSTATES] =
{
  /*  oldst  chtype> OPR    ATM    QST    SPC    ONE    ILL    */
     /*OPR*/ {     OPR|B,  ATM|B,  QST|B,  SPC|MB, ONE|B,  ILL|MB  },
     /*ATM*/ {     OPR|B,  ATM,    QST|B,  SPC|MB, ONE|B,  ILL|MB  },
     /*QST*/ {     QST,    QST,    OPR,    QST,    QST,    QST     },
     /*SPC*/ {     OPR,    ATM,    QST,    SPC|M,  ONE,    ILL|MB  },
     /*ONE*/ {     OPR,    OPR,    OPR,    OPR,    OPR,    ILL|MB  },
     /*ILL*/ {     OPR|B,  ATM|B,  QST|B,  SPC|MB, ONE|B,  ILL|M   },
};

/* token type table -- it gets modified with $o characters */
static u_char   TokTypeTab[256] =
{
  /*  nul soh stx etx eot enq ack bel  bs  ht  nl  vt  np  cr  so  si   */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,SPC,SPC,SPC,SPC,SPC,ATM,ATM,
  /*  dle dc1 dc2 dc3 dc4 nak syn etb  can em  sub esc fs  gs  rs  us   */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  sp  !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /    */
     SPC,ATM,QST,ATM,ATM,ATM,ATM,ATM, SPC,SPC,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~   del  */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,

  /*  nul soh stx etx eot enq ack bel  bs  ht  nl  vt  np  cr  so  si   */
     OPR,OPR,ONE,OPR,OPR,OPR,OPR,OPR, OPR,OPR,OPR,OPR,OPR,OPR,OPR,OPR,
  /*  dle dc1 dc2 dc3 dc4 nak syn etb  can em  sub esc fs  gs  rs  us   */
  OPR,OPR,OPR,ONE,ONE,ONE,OPR,OPR, OPR,OPR,OPR,OPR,OPR,OPR,OPR,OPR,
  /*  sp  !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o    */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
  /*  p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~   del  */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
};

/* token type table: don't strip comments */
u_char  TokTypeNoC[256] =
{
  /*  nul soh stx etx eot enq ack bel  bs  ht  nl  vt  np  cr  so  si   */
     ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,SPC,SPC,SPC,SPC,SPC,ATM,ATM,
```

```
	/* dle dc1 dc2 dc3 dc4 nak syn etb  can em  sub esc fs  gs  rs  us  */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
	/* sp  !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /  */
	   SPC,ATM,QST,ATM,ATM,ATM,ATM,ATM, OPR,OPR,ATM,ATM,ATM,ATM,ATM,ATM,
	/* 0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?  */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
	/* @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
	/* P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _  */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
	/* `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
	/* p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~  del */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,

	/* nul soh stx etx eot enq ack bel  bs  ht  nl  vt  np  cr  so  si  */
	   OPR,OPR,ONE,OPR,OPR,OPR,OPR,OPR, OPR,OPR,OPR,OPR,OPR,OPR,OPR,OPR,
	/* dle dc1 dc2 dc3 dc4 nak syn etb  can em  sub esc fs  gs  rs  us  */
	   OPR,OPR,OPR,ONE,ONE,ONE,OPR,OPR, OPR,OPR,OPR,OPR,OPR,OPR,OPR,OPR,
	/* sp  !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /  */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
	/* 0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?  */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
	/* @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
	/* P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _  */
	  ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
	/* `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
	/* p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~  del */
	   ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM, ATM,ATM,ATM,ATM,ATM,ATM,ATM,ATM,
};

#define NOCHAR		-1	/* signal nothing in lookahead token */

#define DELIMCHARS	"()<>,;\r\n"	/* default word delimiters */

char	*OperatorChars; /* operators (old $o macro) */
int	ConfigLevel;	/* config file level */


/*
**  PRESCAN -- Prescan name and make it canonical
**
**	Scans a name and turns it into a set of tokens.  This process
**	deletes blanks and comments (in parentheses) (if the token type
**	for left paren is SPC).
**
**	This routine knows about quoted strings and angle brackets.
**
**	There are certain subtleties to this routine.  The one that
**	comes to mind now is that backslashes on the ends of names
**	are silently stripped off; this is intentional.  The problem
**	is that some versions of sndmsg (like at LBL) set the kill
**	character to something other than @ when reading addresses;
**	so people type "csvax.eric\@berkeley" -- which screws up the
**	berknet mailer.
**
**	Parameters:
**		addr -- the name to chomp.
**		delim -- the delimiter for the address, normally
**			'\0' or ','; \0 is accepted in any case.
**			If '\t' then we are reading the .cf file.
**		pvpbuf -- place to put the saved text -- note that
**			the pointers are static.
**		pvpbsize -- size of pvpbuf.
**		delimptr -- if non-NULL, set to the location of the
**			terminating delimiter.
**		toktab -- if set, a token table to use for parsing.
**			If NULL, use the default table.
```

```
**
**    Returns:
**          A pointer to a vector of tokens.
**          NULL on error.
*/

char **
prescan(addr, delim, pvpbuf, pvpbsize, delimptr, toktab)
      char *addr;
      int delim;
      char pvpbuf[];
      int pvpbsize;
      char **delimptr;
      u_char *toktab;
{
      register char *p;
      register char *q;
      register int c;

      bool bslashmode;
      bool route_syntax;
      int cmntcnt;
      int anglecnt;
      char *tok;
      int state;
      int newstate;
      char *saveto = CurEnv->e_to;
      /*static char *av[MAXATOM + 1]; */
      /*static char firsttime = FALSE; */
      int errno;


      printf("Inside prescan!!\n");
      printf("Max storage of pvpbuf = %d\n", PSBUFSIZE);


      if (toktab == NULL)
            toktab = TokTypeTab;

      /* make sure error messages don't have garbage on them */
      errno = 0;

      q = pvpbuf;
      bslashmode = FALSE;
      route_syntax = FALSE;
      cmntcnt = 0;
      anglecnt = 0;
     /* avp = av; */

      state = ATM;
      c = NOCHAR;
      p = addr;
      CurEnv->e_to = p;

      do
      {
            /* read a token */
            tok = q;
            for (;;)
            {
                  /* store away any old lookahead character */
                  if (c != NOCHAR && !bslashmode)
                  {
                        /* see if there is room */

                        if (q >= &pvpbuf[pvpbsize - 5])
                        {
                          printf("553 5.1.1 Address too long\n");

                          if (strlen(addr) > (SIZE_T) MAXNAME)
```

```
  {
        printf("strlen(addr) > %d\n", MAXNAME);
        addr[MAXNAME] = '\0';
      }
      returnnull:
        if (delimptr != NULL)
          *delimptr = p;
        CurEnv->e_to = saveto;
        return NULL;
    }

    /* squirrel it away */
    printf("Writing %c to q!\n", c);
    *q++ = c;
}

    /* read a new input character */
    c = (*p++) & 0x00ff;          /* The bad program says c = *p++, in which case if *p =0xff,
                                     then c = -1 = NOCHAR.  We would like c to equal 0xff if *p =
                                     0xff.  This is accomplished by anding *p with 0x00ff.  */

    if (c == '\0')
    {
    /* diagnose and patch up bad syntax */
    if (state == QST)
    {
       printf("653 Unbalanced '\"'");
        c = '"';
    }
    else if (cmntcnt > 0)
    {
        printf("653 Unbalanced '('");
        c = ')';
    }
    else if (anglecnt > 0)
    {
        c = '>';
        printf("653 Unbalanced '<'");
    }
    else
        break;

    p--;
}
else if (c == delim && cmntcnt <= 0 && state != QST)
{
    if (anglecnt <= 0)
        break;
    /* special case for better error management */
    if (delim == ',' && !route_syntax)
    {
        printf("653 Unbalanced '<'");
        c = '>';
        p--;
    }
}

/* chew up special characters */


if (q >= &pvpbuf[pvpbsize - 5]){
  return NULL;
}
else
  /*OK*/
  *q = '\0';

if (bslashmode)
{
```

```
    /*printf("bslashmode = TRUE!!!\n");*/

        bslashmode = FALSE;

        /* kludge \! for naive users */
        if (cmntcnt > 0)
        {
            c = NOCHAR;
            continue;
        }
        else if (c != '!' || state == QST)
        {

         printf("Writing slash to q!!!!\n");

         if (q >= &pvpbuf[pvpbsize - 5]){
          return NULL;
         }
         else
         /*OK*/
          *q++ = '\\';

         continue;  /* continue while loop */
        }
}

  if (c == '\\')
  {
        bslashmode = TRUE;
  }
  else if (state == QST)
  {
        /* EMPTY */
        /* do nothing, just avoid next clauses */
  }
  else if (c == '(' && toktab['('] == SPC)
  {
        cmntcnt++;
        c = NOCHAR;
  }
  else if (c == ')' && toktab['('] == SPC)
  {
        if (cmntcnt <= 0)
        {
            printf("653 Unbalanced ')'");
            c = NOCHAR;
        }
   else
            cmntcnt--;
  }
  else if (cmntcnt > 0)
  {
        c = NOCHAR;
  }
  else if (c == '<')
  {
        char *ptr = p;

        anglecnt++;
        while (isascii((int)*ptr) && isspace((int)*ptr))
            ptr++;
        if (*ptr == '@')
            route_syntax = TRUE;
  }
  else if (c == '>')
  {
        if (anglecnt <= 0)
        {
```

```
                        printf("653 Unbalanced '>'");
                        c = NOCHAR;
                      }
                    else{
                      anglecnt--;
                    }
                    route_syntax = FALSE;
                }
                else if (delim == ' ' && isascii(c) && isspace(c))
                    c = ' ';

                if (c == NOCHAR){
                  printf("c = NOCHAR.... continuing....!\n");
                    continue;
                }

                /* see if this is end of input */
                if (c == delim && anglecnt <= 0 && state != QST)
                  {
                    printf("breaking from for loop!\n");
                  break;
                  }
                newstate = StateTab[state][toktab[c & 0xff]];
                            state = newstate & TYPE;
                if (state == ILL)
                {
                    if (isascii(c) && isprint(c))
                        printf("653 Illegal character %c", c);
                    else
                        printf("653 Illegal character 0x%02x", c
                 );
                }
                /* if (bitset(M, newstate)) */
                if (newstate & M)               /* replacement for bit set */ {
                  c = NOCHAR;
                }
                /* if (bitset(B, newstate)) */
                if (newstate & B)
                  {
                    break;
                  }
            }
             if (tok != q)
            {
                printf("writing null byte\n");
                if (q >= &pvpbuf[pvpbsize - 5]){
                  return NULL;
                }
                else
                  /*OK*/
                  *q++ = '\0';

                if (q - tok > MAXNAME)
                {
                    printf("553 5.1.0 prescan: token too long");
                    goto returnnull;
                }
            }

        } while (c != '\0' && (c != delim || anglecnt > 0));

        printf("Exiting while loop!\n");
    p--;
        if (delimptr != NULL)
            *delimptr = p;

        CurEnv->e_to = saveto;

        return NULL;
```

```
}

char **
parseaddr(addr, delim, delimptr)
        char *addr;
        int delim;
        char **delimptr;
{

  register char **pvp;
  char pvpbuf[PSBUFSIZE];
  static char *delimptrbuf;

  /*
  **  Initialize and prescan address.
*/


  if (delimptr == NULL)
    delimptr = &delimptrbuf;

  pvp = prescan(addr, delim, pvpbuf, sizeof pvpbuf, delimptr, NULL);

  return pvp;


}

int main(){

  char *addr;
  int delim;

  static char **delimptr;
  char special_char = '\377';  /* same char as 0xff.  This char will get interpreted as NOCHAR */
  int i = 0;
  addr = (char *) malloc(sizeof(char) * 500);

  for(i=0; i<300; i=i+2){
    addr[i] = '\\';
    addr[i+1] = special_char;
  }

  delim = '\0';
  delimptr = NULL;

  OperatorChars = NULL;
  for(i=0; i<300; i=i+2){
    addr[i] = '\\';
    addr[i+1] = special_char;
  }

  delim = '\0';
  delimptr = NULL;

  OperatorChars = NULL;

  ConfigLevel = 5;

  CurEnv = (ENVELOPE *) malloc(sizeof(struct envelope));
  CurEnv->e_to = (char *) malloc(strlen(addr) * sizeof(char) + 1);

  strcpy(CurEnv->e_to, addr);

  parseaddr(addr, delim, delimptr);


  return 0; }
```

APPENDIX E

The following is an include file for PolySpace C Verifier that was used during program analysis in several instances.

polyspace.h:

```
#undef __STRICT_ANSI__

#define unix 1
#define i386 1
#define __NO_INLINE__ 1
#define linux 1
#define __GNUC__2
#define __linux__ 1
#define __unix__ 1
#define __ELF__ 1
#define __i386__ 1
#define __linux 1

#define __GNUC_MINOR__ 96
#define __unix 1
define __i386 1
#define unix 1

#define __WINT_TYPE__ unsigned int

/* Disable unsupported GNU extensions */
#define __extension__
#define __const const
#define __attribute__(x)
#define __restrict
```