IOSR Journal of Computer Engineering (IOSRJCE)
ISSN : 2278-0661 Volume 1, Issue 1 (May-June 2012), PP 10-23
www.iosrjournals.org

Buffer Overflow Attack

Samanvay Gupta
4th year 1st semester (Computer Science Department)
Visvesvaraya College of Engineering & Technology

Abstract -- Exploits, vulnerabilities, and buffer-overflow techniques have been used by malicious hackers and virus writers for a long time. In order to attack and get the remote root privilege, using buffer overflow and suidprogram has become the commonly used method for hackers. This paper include vast idea and information regarding the buffer overflow as history of Vulnerabilities, buffers, stack, registers, Buffer Overflow Vulnerabilities and Attacks, current buffer over flow, Shell code, Buffer Overflow Issues, the Source of the Problem, prevention/detection of Buffer Overflow attacks and Finally how to react towards Buffer Overflows. The objective of this study is to take one inside the buffer overflow attack and bridge the gap between the "descriptive account" and the "technically intensive account"

Introduction

Buffer overflows have been documented and understood as early as 1972[23]. In computer security and programming, a buffer overflow is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory. This is a special case of violation of memory safety. Buffer overflows can be triggered by inputs that are designed to execute code, or alter the way the program operates. This may result in erratic program behavior, including memory access errors, incorrect results, a crash, or a breach of system security. Thus, they are the basis of many software vulnerabilities and can be maliciously exploited. Programming languages commonly associated with buffer overflows include C and C++, which provide no built-in protection against accessing or overwriting data in any part of memory and do not automatically check that data written to an array (the built-in buffer type) is within the boundaries of that array. Bounds checking can prevent buffer overflows. A buffer overflow occurs when a program writes data outside the bounds of allocated memory. Buffer overflow attacks form a substantial portion of all security attacks simply because buffer overflow vulnerabilities are so common [1] and so easy to exploit [2, 3, 4, and 5].
Example:

One example is listed as follow:

```
Function (char *buf_src)
{
Charbuf_dest [16];
strcpy (buf_dest, buf_src);
}
/* main function */
Main ()
{
inti;
Charstr[256];
For (i=0; i<256; i++) str[i] = 'a';
Function (str);
}
```

We can see obviously from the program that the size of array str(256 bytes) exceeds greatly the length of buf_dest(16 bytes). Buffer overflow occurred. Buffer overflow vulnerabilities are exploited to overwrite values in memory to the advantage of the attacker.

History of Vulnerabilities

In November 1988, the famous Internet Worm managed to bring down an estimated 10 percent of the Internet. One part of this worm exploited buffer overflow vulnerability in the TCP finger service that existed on some VAX computers running versions 4.2 or 4.3 of BSD UNIX. Finger is a simple program. It does nothing more than report information about users on a particular computer, like their name, office location, or phone number. The finger program in turn calls the C library function gets (), a function used to

handle string input data. While gets() had a buffer 512 bytes long, the Internet Worm passed a string 536 bytes long to the finger program that was in turn passed to gets(). This caused the buffer overflow that gave the Internet Worm access to the target computer. More recently, buffer overflow vulnerabilities have been responsible for many high-profile worms, including the Code Red worm of July 2001, the Slapper worm of September 2002, and the Slammer worm of January 2003. The Code Red worm infected over 359,000 computers in less than 14 hours and caused an estimated $2.6 billion in damage3 by exploiting buffer overflow vulnerability in Microsoft Internet Information Services (IIS). Information about this vulnerability was first released on June 12 2001.

Buffers

A buffer is a given quantity of memory reserved to be filled with data. Say a program is reading strings from a file, like a dictionary, it could find the name of the longest word in the English language and set that to be the size of its buffer. A problem arises when the file contains a string larger than our buffer. This could occur legitimately, where a new, very long word is accepted into the dictionary, or when a hacker inserts a string designed to corrupt memory. Figure illustrates these concepts using the strings “Hello” and “Dog”, with some garbage “x” “y”.

Pointers and the Flat Memory Model

A pointer is an address used to refer to an area elsewhere in memory. They are often used to refer to strings on the heap (another area of memory used to store data) or to access multiple pieces of data via a common reference point and offset. The most important pointer for a hacker is one that refers to an execution point, which is an area with machine code to be executed. These types of pointers will be discussed later in this paper.

The flat memory model is employed by most current operating systems. It provides processes with one contiguous (virtual) area of memory, so that the program can refer to any point of itsallocate memory from a single offset. This may not seem significant now, but it makes it significantly easier for hackers to find their buffers and pointers in memory. The implementation of virtual memory allocation has made a large impact on computing. Processes are now allocated a virtual memory range which refers to an area of physical memory by reference. This means it is far more likely buffers will occur in the same memory location time and time again, as they do not have to worry about other processes occupying the memory space their buffer used on a previous run. The best way to illustrate this principle is to open two separate programs in a debugger and note that they both appear to be using the same memory address space.

Stack

There are many memory structures to consider in x86 architecture (and indeed, many other architectures), the one that will be discussed in this paper is the stack. The technical name for this particular stack is the call stack, but for simplicities sake, it will simply be referred to as ‘stack’ in this paper. Every time the program makes a function call, arguments to that function are ‘pushed’ onto the stack, so that they can be referenced, used and manipulated quickly. The way the stack works

H	e	l	l	o	NUL	D	o	g	NUL	x	y

Say the program allows users to give a new greeting message (replacing Hello with something more personal), the buffer is 6 bytes long, 5 for the word and one for the buffer, say "Hello" is replaced with "Heya", the 4 char word will fill the buffer, a NUL terminator will be appended, there will be one garbage value and then the next string will start, as it did before.

H	e	y	a	NUL	r	D	o	g	NUL	x	y

Note that the r char is a garbage value and could represent anything. It is simply the last value that was in that area of memory. A longer string like "DonkeysCat" will overwrite some adjacent area of memory.

D	o	n	k	e	y	C	a	t	NUL	x	y

If the program now tries to reference the string Dog, it will in fact be referencing the latter section of our previous string, "Cat".

Figure1: String in memory

is it has a register that points to the very top of it (called ESP in 32 bit systems, where the SPstands for *'Stack Pointer'*) which gets incremented (by the size of a given buffer or memorypointer, give or take a few bytes of padding) to make room for new data the process wants tostore. Figure 2 illustrates a string being pushed onto the stack, above another string.

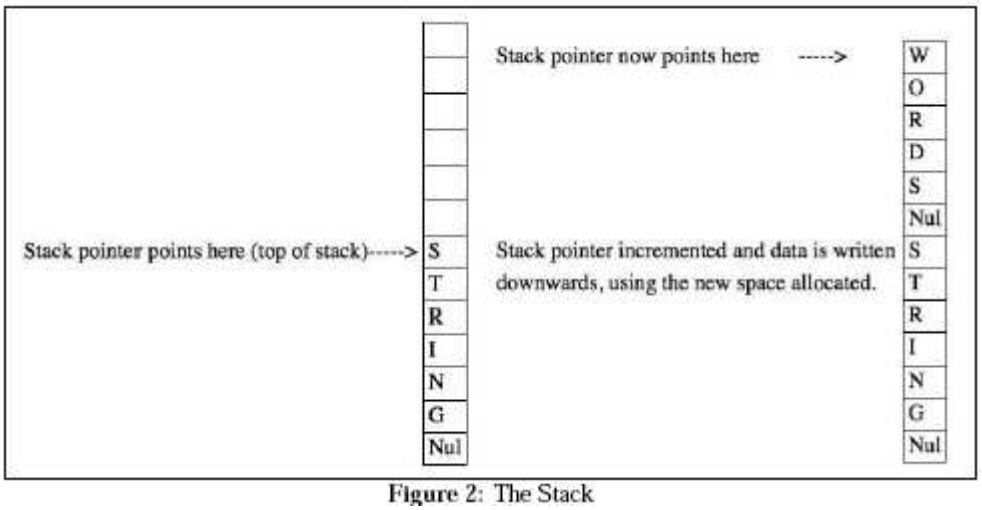


Figure 2: The Stack

The stack is like a tower, we write from the top to the bottom. If ESP provides 50 bytes worth of space, but 60 bytes worth are supplied, the cpu is going to overwrite 10 bytes of information that the it may want to use at a later stage. This diagram does not represent the complexity of the data that resides on the stack. By tactically overwriting the correct areas of memory, some very interesting effect can be observed. To use an analogy, the stack is like the cpu's notepad. When people do things like math's or research, they like to jot down numbers or page references on scrap paper. If a notepad gets too covered in notes, they could end up writing over some of their previous notes and misinterpret them at a later stage.

Registers

Registers are sections of high speed memory that reside on the CPU itself. General purpose registers (nAX, nBX, where n is a character that represents the size of the register) are used for arithmetic, counters, storage of pointers, flags,function arguments and a number of other purposes. As well as general purpose registers, there are some which have more specific purposes. nSPfor instance, points at the lowest address (the topmost point) in the stack, hence the name StackPointer. This register is extremely useful for referencing data on the stack, as the location ofdata in memory can vary greatly, but there is much less variation between data on the stack and the location nSP points to. Another important register to consider in the world of computer security is nIP, the InstructionPointer. This pointer points to the address of the current point of execution. The way this pointer gets its values is of extreme interest to hackers, and will be explained later.

General idea of Buffer Overflow Attack

Target of Buffer Overflow Attacks

Subvert the usual execution flow of a program by redirecting it to anInjected (malicious) code

The attack consists of:

1 Injecting new (malicious) code into some writable memory area,

2 Changing a code pointer (usually the return address) in such a way that it points to the injected malicious code.

General Terms

Buffer overflow is the root cause for most of the cyber-attacks like worms, zombies, botnets and server breaking.

Worms - Self-replicating malware computer program which uses a computer network to send copies of itself to other nodes and it may do so without any user intervention.

Zombies - a computer connected to the Internet that has been compromised by a computer virus or cracker and can be used to perform malicious tasks even in remote directions. **Botnets -** a large number of

compromised computers that are used to create and send viruses or flood a network with messages as a denial of service attack.

Return to libc attacks – a computer security attack usually starting with a buffer overflow in which the return address on the stack is replaced by the address of another instruction. This allows attackers to call malicious code without the need to inject malicious code into a program.

Code Injection

Code can be injected by overflowing a local buffer allocated on the stack. The target of the injected code is usually to launch a shell to the adversary Therefore the injected code is often referred to as **shell code.**

Shell code

A common element in all buffer overflow exploits is the shell code. Shell code is the attacker's code which is triggered by a exploiting a vulnerability. It is typically planted in an input buffer of a vulnerable program which is then tricked into running it. Shell code has to be compiled and assembled before it can be planted. Often, it is also necessary to character encode it. For example, when the target program expects character input from the user, the user would supply characters whose binary encodings (in ASCII) represent the shell code. This presents its own problems since the code must not use certain bytes. For example, end-of-file (^D) or newline characters which would terminate the input. The shell code usually contains instructions to launch a shell or a remote xterm. If the target program is a server daemon running as root then the shell will run as root too. Through a root shell the attacker has unrestricted access to the target machine. These days shell-code is much more than that and what it can do is only limited by a hacker's creativity. Because of this, some experts in the field have suggested the name shell-code is insufficient [24]. As it is It is common practice to write shell-code in assembly and use an assembler such as NASM http://www.nasm.us/ which converts assembly instructions to OPCODEs and does some low level memory management, such as creation of stack frames (unless the user opts to do that themselves). This OPCODE is then modified to run as shell code. Note that the shell-code used in this paper was written by Steve Hanna [25].

Buffer Overflow Vulnerabilities and Attacks

Goal: subvert the function of a privileged program so that the attacker can take control of that program, and if the program is sufficiently privileged, thence control the host. Typically the attacker is attacking a root program, and immediately executes code similar to "exec (sh)" to get a root shell, but not always. To achieve this goal, the attacker must achieve two sub-goals:

1. Arrange for suitable code to be available in the program's address space.
2. Get the program to jump to that code, with suitable parameters loaded into registers & memory.

NUL Terminated Strings0x00

There have been few fundamentals of computer science, operating systems and programming languages as controversial as the implementation of NUL terminated strings. They've been referred to as *'The most expensive one-byte mistake'*[26] (incidentally, it would have been far more than a *'one'*-byte mistake, if it was a mistake at all, but that's a topic for another paper) and they are the reason buffers overflow in the manner they do. When a NUL terminated string is written to the stack (or anywhere, in fact), the program will mindlessly continue writing data until it reaches this NUL terminator. This means it will overwrite other arguments, saved pointers (which are of GREAT importance and will be discussed later), absolutely anything in fact.

Current generation Buffer Overflow Attack

Format String Attacks

Format string vulnerabilities occur due to sloppy coding by software engineers. A variety of C language functions allow printing the characters to files, buffers, and the screen. These functions not only place values on the screen, but can format them as well.The following list contains common ANSI format functions:

• printf _ print formatted output to the standard output stream
• Wprintf _ wide-character version of printf
• Fprintf _ print formatted data to a stream (usually a file)
• Fwprintf _ wide-character version of fprintf
• Sprintf _ writes formatted data to a string
• Swprintf _ wide-character version of sprintf
• Vprintf _ writes formatted output using a pointer to a list of arguments
• Vwprintf _ wide-character version of vprintf
• Vfprintf _ writes formatted output to a stream using a pointer to a list of arguments
• Vwfprintf _ wide-character version of vfprintf

The formatting ability of these functions allows programmers to control how their output is written. For example, a program could print the same value in both decimal and hexadecimal.

```
#include <stdio.h>
Voidmain (void)
{
int foo = 1234;
printf ("Foo is equal to: %d (decimal), %X (hexadecimal)", foo, foo);
}
```

The above program would display: Foo is equal to: 1234 (decimal), 4D2 (hexadecimal)
The percent sign "%" is an escape character signifying that the next character(s) represent the form in which the value should be displayed. Percent-d, (%d) for example, means display the value in decimal format and %X means display the value in hexadecimal with uppercase letters. These are known as format specifiers. See Appendix B for the full list of ANSI C format specifiers. The format function family specification requires the format control, and then an optional number of arguments to be formatted.However, sloppy programmers often do not follow this specification. The below program will print out Hello World! On the screen, but does not strictly follow the specification. The commented line demonstrates the proper way the program should be written.

```
#include <stdio.h>
void main(void)
{
char buffer [13] ="Hello World!"
printf (buffer); // using argument as format control!
// printf ("%s",buffer); this is the proper way
}
```

This type of sloppy programming allows one to potentially control the stack and inject and executearbitrary code. The following program takes in one command line parameter and writes the parameterback to the screen. Notice the printf statement is used incorrectly by using the argument directly insteadof a format control.

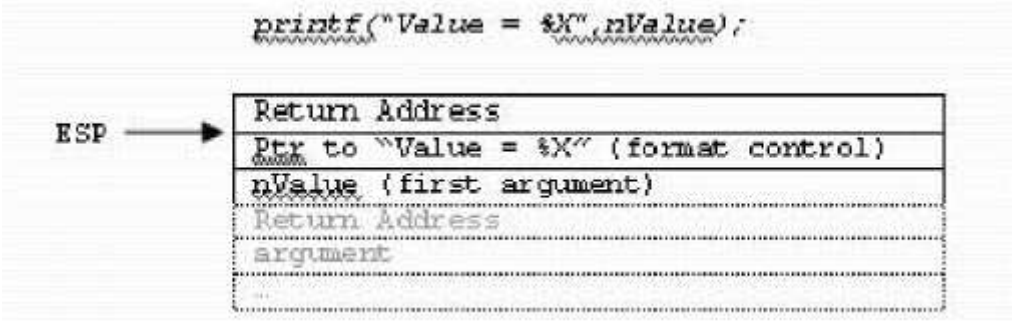
```
intvuln(char buffer[256])
{
intnReturn=0;
printf(buffer); // print out command line
// printf("%s",buffer); // correct-way
return (nReturn);
}
void main(intargc,char *argv[])
{
char buffer [256]=""; // allocate buffer
if (argc == 2)
{
strncpy(buffer,argv[1],255); // copy command line
}
vuln(buffer); // pass buffer to bad function
}
```

This program will copy the first parameter on the command line into a buffer. Then, the buffer will bepassed to the vulnerable function. However, since the buffer is being used as the format control, insteadof feeding in a simple string, one can attempt to feed in a format specifier.
For example, running this program with the argument "%X" will return some value on the stack instead of "%X":

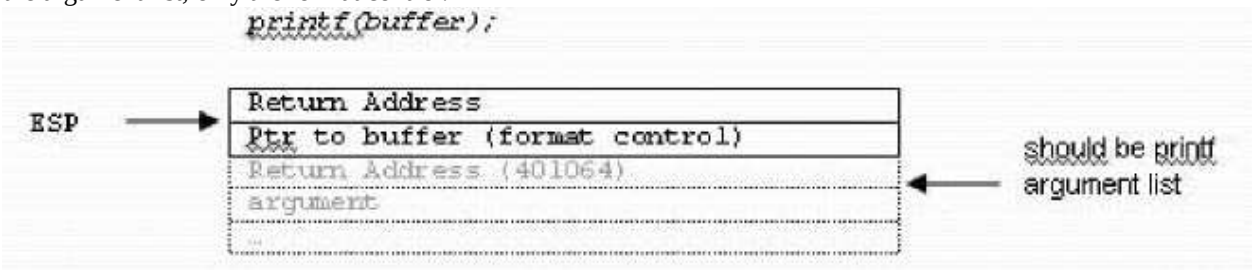
```
C:\>myprog.exe %X
401064
```

The program interprets the input as the format specifier and returns the value on the stack that should be the passed-in argument. To understand why this is the case, one needs to examine the stack just afterprintf is called. Normally, if one uses a format control and the proper number of arguments, the stack

willlook similar to the following:



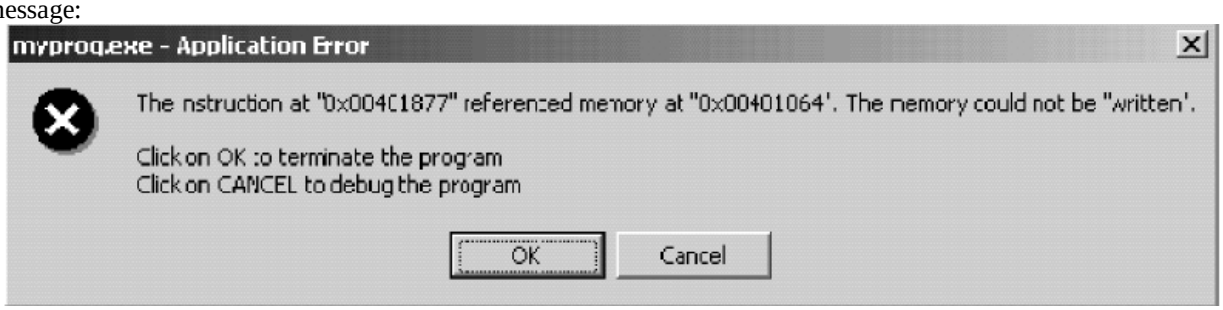
However, by using printf incorrectly, the stack will appear differently since one does not push on the argument list, only the format control.



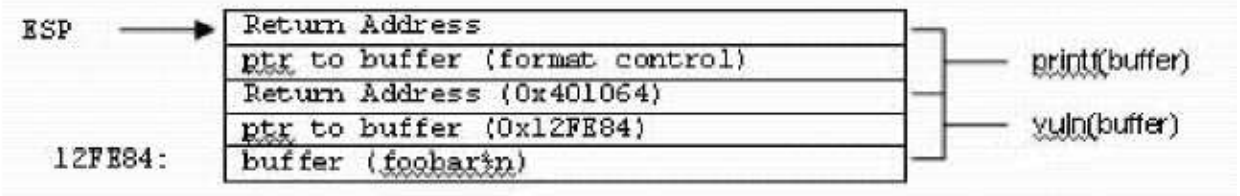
In this case, the program will believe the stack space after the format control is the first argument. Thus, by feeding in "%X" to our sample program, printf displays the return address of the previous function call instead of the expected missing argument. In this example we display arbitrary memory, but the key to exploitation from the point of view of a malicious hacker or virus writer would be the ability to write to memory in order to inject arbitrary code. The format function families allow for the "%n" format specifier. This format specifier will store (write to memory) the total number of bytes written. For example, printf ("foobar%n",&nBytesWritten); will store "6" in nBytesWritten since foobar consists of six characters. Consider the following:

C:\>myprog.exe foobar%n

When executed instead of displaying the value on the stack after the format control, the program will attempt to write to that location. So, instead of displaying 401064 as demonstrated above, the program will attempt to write the number 6 (the total characters in foobar) to the address 401064 and result in an application error message:

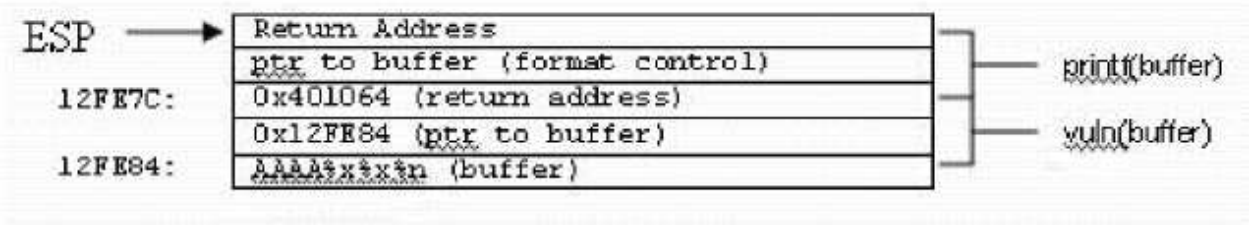


However, this demonstrates that one can write to memory. With this ability, one would actually wish to overwrite a return pointer (as in buffer overflows), redirecting the execution path to injected code. Examining the stack more fully, one determines in the example program, the stack appears as follows:

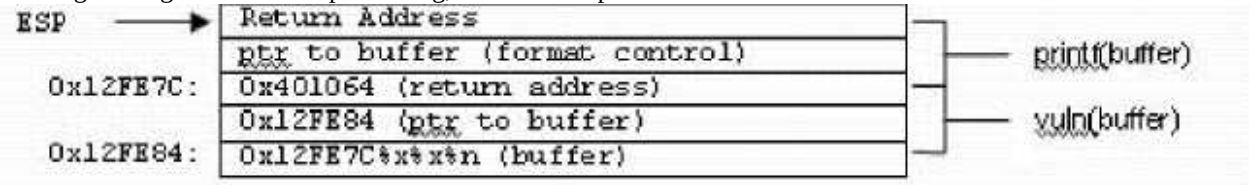


Knowing how the stack appears, consider the following exploit string:

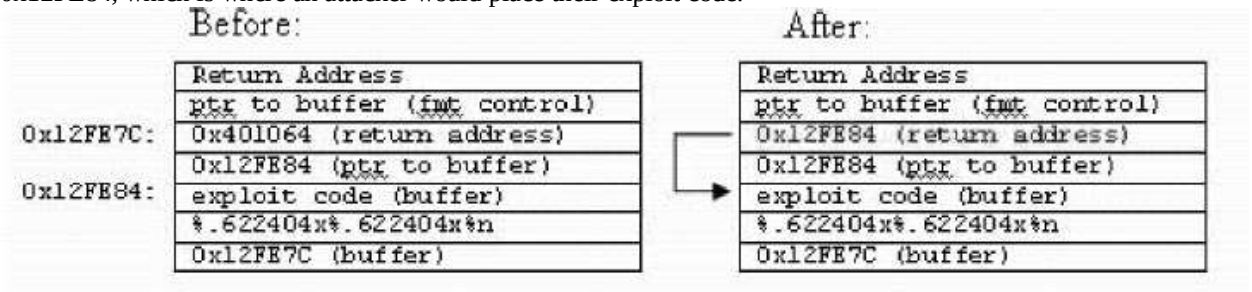
C :> myprog.exe AAAA%x%x%n



The first format specifier, "%x," is considered the Return Address (0x401064), the next format specifier, "%x," is 0x12FE84. Finally, %n will attempt to write to the address specified by the next DWORD on the stack, which is 0x41414141 (AAAA). This allows an attacker to write to arbitrary memory addresses. Instead of writing to address 0x41414141, an exploit would attempt to write to a memory location that contains a saved return address (such as in buffer overflows). In this example, 0x12FE7C is where a return address is stored. By overwriting the address in 0x12FE7C, one can redirect the execution path. So, instead of using a string of as in the exploit string, one would replace them with the value 0x12FE7C.



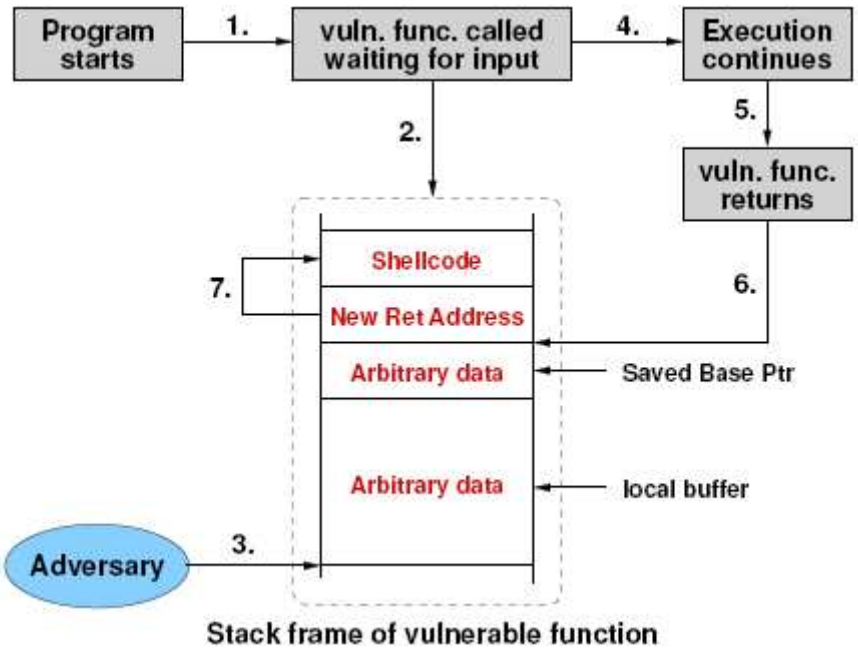
The return address should be overwritten by the address that contains the exploit code. In this case, that would be at 0x12FE84 which is where the input buffer is located. Fortunately, the format specifiers can include how many bytes to write using the syntax %.<bytestowrite>x. Consider the following
Exploit string:
<exploitcode>%x%x%x%x%x%x%x%x%x%.622404x%.622400x%n\x7C\xFEx12
This will cause printf to write the value 0x12FE84 (622404+622400=0x12FE84) to 0x12FE7C if the exploit code is two bytes long. This overwrites a saved return address causing the execution path to proceed to 0x12FE84, which is where an attacker would place their exploit code.



Thus, by not following the exact specification, programmers can allow hackers to overwrite values in memory and execute arbitrary code. While the improper usage of the format function family is widespread, finding vulnerable programs is also relatively easy. Therefore, most popular applications have been tested by security researchers for such vulnerabilities. Nevertheless, new applications are developed constantly and unfortunately developers continue to usethe format functions improperly, leaving them vulnerable.

Conventional Buffer Overflow Attack

The main goal of a conventional buffer overflow attack [7] is to subvert the usual execution of a program by redirecting it to a malicious code that was not originally placed by the programmer. Basically, the attack consists of two tasks: (i) injecting new malicious code in some writable memory area and (ii) changing a code pointer in such a way that it points to the injected malicious code. The injected malicious code usually launches a shell to the adversary and is therefore often referred to as shell code. The preferred code pointer to run the attack is the return address on the stack. However, also the saved base pointer can be used as an attack target; in the case the return address cannot be overwritten. This attack is referred to as frame pointer overwriting [6].



Buffer Overflow Issues

1Executable attack code is stored on stack, inside the buffer containing attacker's string

• Stack memory is supposed to contain only data, but…

2Overflow portion of the buffer must contain correct address of attack code in the RET position

• The value in the RET position must point to the beginning of attack assembly code in the buffer

Otherwise application will (probably) crash with segmentation violation

• Attacker must correctly guess in which stack position his/her buffer will be when the function is called.

How to find buffer overflows?

1. Hackers find buffer overflows as follows:

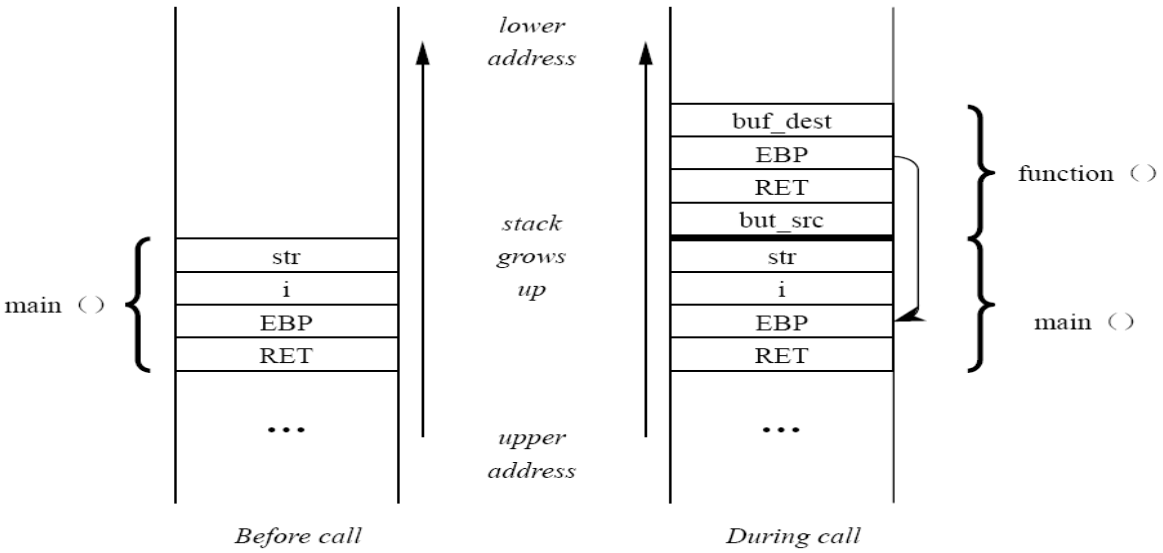
• Run web server on local machine.

• Issue requests with long tags.

All long tags end with "$$$$$".

• If web server crashes, search core dump for "$$$$$" to find overflow location.

2. Some automated tools exist.



Why Is The Buffer Overflow Problem So Important in Computer and Network Security?

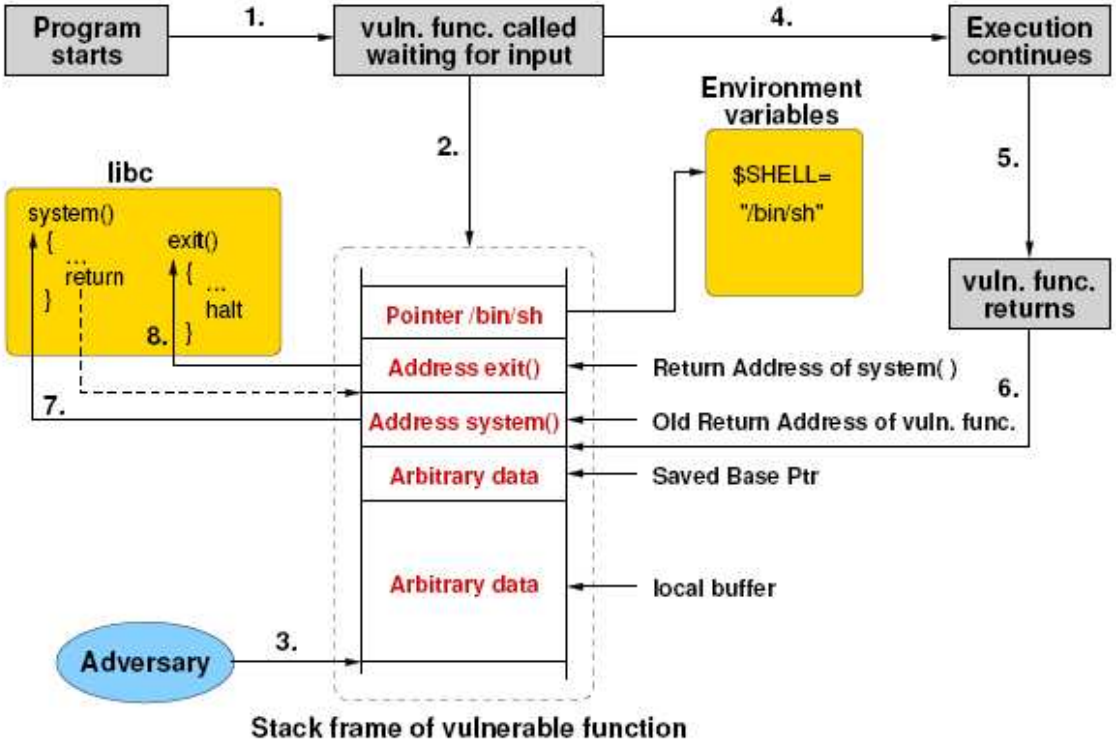
1.Practically every worm that has been unleashed on the Internet has exploited bufferoverflow vulnerability in some networking software
2. The statement made above is just as true today as it was 20 years ago when the Morris worm caused a major disruption of the internet.
3. Yes, it is true that modern compilers can now inject additional code into the executables that at run time checks for the conditions that cause buffer overflow. However, the production versions of the executables may not incorporate such protection for performance reasons. Additional constraints, such as those that apply to small embedded systems, may call for particularly small executables, meaning executables without the protection against buffer overflow.

Effectiveness of Payload Bypassing

We examined the effectiveness of payload bypassing from three angles: (1) what is the percentage of traffic that is "payload"? (2) How many false positives are eliminated? And (3) how much throughput improvement can be achieved? We tested four protocols, HTTP, FTP, Bit Torrent, and eDonkey on Windows XP, to answer the first two questions. We use Windows applications because most Internet traffic is generated from Windows machines and programs using these protocols are readily available on the Windows platform. Files being downloaded are from the static sample, which has 8068 files with a total size of 1.22 Gbytes. [18] We used Linux machines to answer the third question because we found that Windows XP can only achieve around 200 Mbps TCP throughputs when two Windows XP machines are directly connected, whereas Linux can achieve 500 Mbps with the same set-up. Files used in this test are a random subset of the static sample because we can only afford to use a 500-Mbyte RAM disk for network throughput test. Using physical hard disks in this test is unacceptable because we were using gigabit Ethernet link.

The Source of the Problem

The local variables are allocated on the stack, along with parameters and linkage information's. The accurate content and order of data on the stack depends on the operating system and processing unit architecture. [19] When you use malloc, new, or the same functions to allocate a block of memory or instantiate an object, the memory is allocated on the heap. Every time your program requests the input from a user, there is a potential for the user to enter inappropriate data. For example, they might enter the more data than you have reserved for in memory. If the user enters more data than will fit in the reserved memory space and you do not trim it, then that data will overwrite other data in memory. If the memory overwritten contained data vital to the operation of the program, this overflow will cause a bug that, being irregular, might be very hard to find. If the overwritten data includes the address of other code to be performed and the user has done this intentionally, the user can point to malicious code that your program will then executes. In the case of data saved on the stack, such as a local variable, it is relatively simple for an attacker to overwrite the linkage information in order to execute malicious code. An attacker can also change local data and function parameters on then stack .The data on the heap changes in a no understandable way as a program runs; utilize a buffer overflow on the heap is more difficult. However, many exploits have involved heap overflows. Attacks on the heap might involve overwriting critical data, either to cause the program to crash, or to modify a value that can be exploited later (such as a program temporarily stores a user name and password on the heap and an attacker control to change them). In some cases, the heap contains pointers to executable code, so that by overwriting such a pointer an attacker can execute the malicious code. Although most programming languages check input against storage to prevent buffer overflows, C, and C++ do not. Because many programs link to C libraries, weakness in standard libraries can cause vulnerabilities even in programs written in "safe" languages. For this reason, even if you are confident that your code is free of buffer overflow problems, you should limit the exposure by running with least privileges possible.



Prevention/Detection of Buffer Overflow attacks.

Many high-level programming languages are generally immune to buffer overflows, either because of the way they handle arrays or the way they detect and prevent such vulnerabilities. [21]Only the C and C++ languages are generally vulnerable. On the other hand, some compilers for languages that are otherwise immune give programmers the option to turn off the protection that they provide by default. Since turning off this protection results in better performance, some programmers will end up doing this. Even languages like Java, that run on a virtual machine, can be vulnerable to such weaknesses, since the virtual machines may have been written in either C or C++. Therefore, although using a different programming language sounds like an easy way to provide protection against buffer overflows, the protection can still be incomplete. Following best practices in defensive programming can greatly reduce or eliminate the number of vulnerabilities that will exist in software. Defensive programming may seem like additional effort that will increase development costs and take additional time, but the additional benefits from reduced support costs from more robust and error-free software make the investment worthwhile. There are also tools that programmers can use to build in protection against certain types of buffer overflow attacks. One type of tool adds guard values called canaries, named after the birds that miners would keep with them to warn of the presence of toxic gasses. Canaries are known values that are placed around return addresses, hoping that anything that changes the legitimate return address will also change the canary values. Other types of tools try to protect information that a buffer overflow attack might try to change by moving the information to different places, so that overflows will not overwrite it. Others try to logically separate the memory segment where buffers are created from the memory segment where program instructions are stored, so that data in a buffer cannot become attack code. There are libraries available that programmers can use that replace dangerous library functions with versions that are safer, so that a dangerous function like gets () is replaced by a version that is safer, for example. None of these techniques is perfect, and each of them can be bypassed by a clever attacker, although they certainly make the attacker's job more difficult. Some buffer overflow vulnerabilities can also be discovered in the testing process for software. A good way to do this is to provide an application with input values that are either random or very unusual. Provide long strings when short ones are expected. Provide negative numbers when positive ones are expected. Provide Unicode strings when ASCII strings are expected. In this type of testing, crashes and unexpected errors may indicate the presence of a buffer overflow. Existing prevention/detection techniques of buffer over-flows can be roughly broken down into six classes:

Class A: Finding bugs in source code. Buffer overflows are fundamentally due to programming bugs. Accordingly, various bug-finding tools [8], [9] have been developed. The bug-finding techniques used in these tools, which in general belong to static analysis, include but are not limited to model checking and bugs-as-deviant-behavior. Class A techniques are designed to handle source. SigFree handles machine code embedded in a request (message).

Class B: Compiler extensions. "If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler" [12]. Three such compilers are Stack Guard [10], ProPolice [11], and Return Address Defender (RAD) [13]. DIRA [14] is another compiler that can detect malicious input, and repair the compromised program.

Class C: OS modifications. Modifying some aspects of the operating system may prevent buffer overflows such as Class C techniques need to modify the OS. In contrast, SigFree does not need any modification of the OS.

Class D: Hardware modifications. A main idea of hard-ware modification is to store all return addresses on the processor. In this way, no

Class E: Defense-side obfuscation. Address Space Layout Randomization (ASLR) is a main component of PaX [15]. Address-space randomization, in its general form [11], can detect exploitation of all memory errors. Instruction set randomization [11], [13] can detect all code-injection attacks, whereas SigFree cannot guarantee detecting all injected code. Nevertheless, when these approaches detect an attack, the victim process is typically terminated. "Repeated attacks will require repeated and expensive application restarts, effectively rendering the service unavailable" [16].

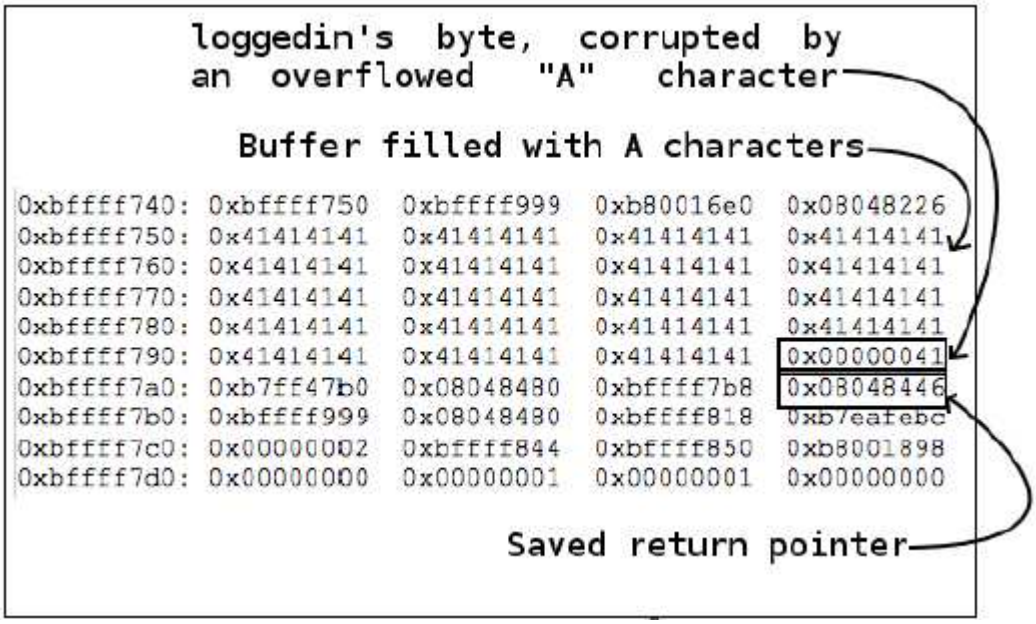
Class F: Capturing code running symptoms of buffer overflow attacks. Fundamentally, buffer overflows are a code running symptom. If such unique symptoms can be precisely captured, all buffer overflows can be detected. Class B, Class C, and Class E techniques can capture some but not all of the running symptoms of buffer overflows. For example, accessing no executable stack segments can be captured by OS modifications; compiler modifications can detect return address rewriting; and process crash is a symptom captured by defense-side obfuscation. To achieve 100 percent coverage in capturing buffer overflow symptoms, dynamic data flow/taint analysis/program shepherding techniques were proposed in Vigilante [15], Taint Check [14] They can detect buffer overflows during runtime. However, it may cause significant runtime overhead (e.g., 1,000 percent). To reduce such overhead, another type of Class F techniques, namely post-crash symptom diagnosis, has been developed in Covers [16] and [17]. Post-crash symptom diagnosis extracts the "signature" after a buffer overflow attack is detected. A more recent system called ARBOR can generate vulnerability-oriented signatures by identifying characteristic features of attacks and using program context. Moreover, ARBOR automatically invokes the recovery actions. Class 1F techniques can block both the attack requests that contain code and the attack requests that do not contain any code, but they need the signatures, Moreover, they either suffer from significant runtime overhead or need special auditing or diagnosis facilities, which are not commonly available in commercial services. In contrast, although SigFree could not block the attack requests that do not contain any code, SigFree is signature free and does not need any changes to real-world services. We will investigate the integration of SigFree with Class F techniques in our future work..The crash log might provide some clues that the root of the crash was a buffer overflow attack

```
Exception:  EXC_BAD_ACCESS (0x0001)
Codes:      KERN_INVALID_ADDRESS (0x0001) at 0x41414140

Thread 0 Crashed:

Thread 0 crashed with PPC Thread State 64:
  srr0: 0x0000000041414140  srr1: 0x000000004200f030                    vrsave: 0x0000000000000000
    cr: 0x48004242           xer: 0x0000000020000007   lr: 0x0000000041414141  ctr: 0x000000009077401c
    r0: 0x0000000041414141    r1: 0x00000000bfffe660   r2: 0x0000000000000000   r3: 0000000000000001
    r4: 0x0000000000000041    r5: 0x00000000bfffdd50   r6: 0x0000000000000052   r7: 0x00000000bfffe638
    r8: 0x0000000090774028    r9: 0x00000000bfffddd8  r10: 0x00000000bfffe380  r11: 0x0000000024004248
   r12: 0x000000009077401c   r13: 0x00000000a365c7c0  r14: 0x0000000000000100  r15: 0x0000000000000000
   r16: 0x00000000a364c75c   r17: 0x00000000a365c75c  r18: 0x00000000a365c75c  r19: 0x00000000a366c75c
   r20: 0x0000000000000000   r21: 0x0000000000000000  r22: 0x00000000a365c75c  r23: 0x000000000034f5b0
   r24: 0x00000000a3662aa4   r25: 0x000000000054c840  r26: 0x00000000a3662aa4  r27: 0x0000000000002f44
   r28: 0x000000000034c840   r29: 0x0000000041414141  r30: 0x0000000041414141  r31: 0x0000000041414141
```

Stack Based Buffer Overflow



1. Main problem:
• strcpy(), strcat(), sprintf() have no range checking.
• "Safe" versions strncpy(), strncat() are misleading
– strncpy() may leave buffer unterminated.
– strncpy(), strncat() encourage off by 1 bugs.
2. Defenses:
• Type safe languages (Java, ML). Legacy code?
• Mark stack as non-execute. Random stack location.
• Static source code analysis.
• Run time checking: StackGuard, Libsafe, SafeC, (Purify).
• Black box testing (e.g. eEye Retina, ISIC).

Reacting to Buffer Overflows

Buffer overflows are fundamentally a problem with the way a program is written, so they can be fixed only by changing the program. Thus your best strategy for dealing with buffer overflow vulnerabilities is to monitor the web sites of the vendors of the software that you use in your organization. Routinely check vulnerability databases like the National Vulnerability Database [19] also. Note that in the cases of the Code Red, Slapper, and Slammer worms, the vulnerabilities were discovered and patches were available from the software vendors well before the worms were released. So if the users of the vulnerable software had had current patches installed, each of these worms would have been unable to affect the patched systems. Staying up to date on available patches can eliminate much vulnerability before they can be exploited. If you find yourself vulnerable to a buffer overflow attack and the vendor of the vulnerable software has not yet issued a patch, then you need to balance the business need for the vulnerable application against the risk involved in keeping it available to users. In the simplest case, you might find that the vulnerable application is not mission-critical, so that temporarily removing it from your system is a viable option. If the vulnerable application is mission-critical, then another option is to limit the users of the application to a minimal number of essential users until a patch is available. In the worst case, you may not be able to restrict access to the vulnerable application. This may be the case for many web-based applications, for example. In this case, you may have to accept the risk that comes with running the vulnerable software until you can patch it.

Conclusion

Secure development practices should include regular testing to detect and fix buffer overflows. The most reliable way to avoid or prevent buffer overflows is to use automatic protection at the language level. Another fix is "bounds checking" enforced at run-time, which prevents buffer overrun by automatically checking that data written to a buffer is within acceptable boundaries.
The art of exploitation can be summarized in four major steps:
• Vulnerability Identification

– The act of finding an exploit, through white-box or black-box testing.
• Offset Discovery and stabilization
– The act of discovering the relative offset of valuable memory locations (the saved return pointer in this instance) and building a stable exploit using techniques such as the NOP sled and memory address repeating (following it's alignment to the stack of course).
• Payload construction
– Discovery of bad characters and delivery of an appropriate payload; of an appropriate size and fit for purpose.
• Exploitation
– The act of feeding the specially crafted 'malformed' input to the program, observing its affects and making use of whatever new supplied code was executed.
Buffer overflows are one of the most severe computer security threats facing developers and consumers today (and have been for the last 40 years), it is important that developers take this fact into consideration when they are writing code.

References

[1] Michele Crabb. Curmudgeon's Executive Summary. In Michele Crabb, editor, The SANS Network SecurityDigest. SANS, 1997. Contributing Editors: Matt Bishop, Gene Spafford, Steve Bellovin, Gene Schultz, Rob Kolstad, Marcus Ranum, Dorothy Denning, Dan Geer, Peter Neumann, Peter Galvin, David Harley, JeanChouanard.

[2] "Aleph One". Smashing The Stack For Fun And Profit. Phrack, 7(49), November 1996.

[3] "Mudge". How to Write Buffer Overflows. http:// l0pht.com/advisories/bufero.html,1997.

[4] Nathan P. Smith. Stack Smashing vulnerabilities in the UNIX Operating System. http://millcomm.com/ nate/machines/security/stack-smashing/natebuffer.ps, 1997.

[5] "DilDog". The Tao of Windows Buffer Overflow.http://www.cultdeadcow.com/cDc_files/cDc-351/, April 1998.

[6] klog. The frame pointer overwrite. Phrack Magazine, 55(9), 1999.

[7] Aleph One. Smashing the stack for fun and pro_t. Phrack Magazine, 49(14), 1996.

[8] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke,S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard:Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," Proc. Seventh USENIX Security Symp.(Security '98), Jan. 1998

[9] T. ckerChiueh and F.-H. Hsu, "Rad: A Compile-Time Solution toBuffer Overflow Attacks," Proc. 21st Int'l Conf. DistributedComputing Systems (ICDCS), 2001.

[10] R. Chinchani and E.V.D. Berg, "A Fast Static Analysis Approach to Detect Exploit Code inside Network Flows," Proc. Eighth Int'l Symp.Recent Advances in Intrusion Detection (RAID), 2005.

[11] G. Kc, A. Keromytis, and V. Prevelakis, "Countering Code-

Injection Attacks with Instruction-Set Randomization," Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03), Oct. 2003.

[12] B.A. Kuperman, C.E. Brodley, H. Ozdoganoglu, T.N. Vijaykumar, and A. Jalote, "Detecting and Prevention of Stack Buffer Overflow Attacks," Comm. ACM, vol. 48, no. 11, 2005.

[13] E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic, and D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03), Oct. 2003.

[14] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," Proc. 12th Ann. Network and Distributed System Security Symp.(NDSS), 2005.

[15] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-End Containment of Internet Worms," Proc. 20thACMSymp. Operating Systems Principles (SOSP),2005.

[16] Z. Liang and R. Sekar, "Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers," Proc.12th ACM Conf. Computer and Comm. Security (CCS), 2005.

[17] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic

Diagnosis and Response to Memory Corruption Vulnerabilities,"Proc. 12th ACM Conf. Computer and Comm. Security (CCS), 2005.

[18] A free, downloadable book by David Wheeler, Secure Programming for Linux and Unix HOWTO — Creating Secure Software, is available at http://www.dwheeler.com/secureprograms/.This book covers many aspects of secureprogramming, including buffer overflows and how to avoid them.

[19] Flaw finder is available at http://www.dwheeler.com/flawfinder/

[20] Buffer Overflow Attacks on Linux Principles Analyzing and Protection ZhiminGuJiandong Yao Jun Qin Department of Computer Science, Beijing Institute of Technology (Beijing 100081)

[21] Z. Liang, R. Sekar, and D.DuVarney. Automatic synthesis of filters to discard buffer overflow attacks: A step towards realizing self healing systems. In USENIXAnnual Technical Conference, 2005.

[22] XinRan Wang, ChiChun Pan, Peng Liu, and Sencun Zhu. Sig free: A signature Free Buffer Overflow Attack Blocker. In 15 th Use nix Security Symposium, July 2006.

[23] James P. Anderson. Computer Security Technology Planning Study. page 61, 1972.

[24] Mike Price James C. Foster. Sockets, Shellcode, Porting, & Coding. Elsevier Science &

Technology Books, April 2005.

[25] Steve Hanna. Shellcodingfor Linux and Windows Tutorial.http://www.vividmachines.com/shellcode/shellcode.html, 2004. [Online; accessed 20/11/2011].

[26] Poul-Henning Kamp. The Most Expensive One-byte Mistake.http://queue.acm.org/detail.cfm?id=2010365, July 2011. [Online; accessed 18/11/2011].