BUFFER OVERFLOW ASSIGNMENT

vulnerable1.c

1. Briefly describe the behavior of the program.
Ans. The program is such that it accepts any number of arguments. But it is understood that in order to gain access to the site of vulnerability i.e. the launch() method, one argument is mandatory. In the launch() method a buffer of size BUFFER_SIZE is defined and the user_argument is copied into it.

2. Identify and describe the vulnerability as well as its implications.
Ans. BUFFER_SIZE value is 200. If the user_argument provided by the user is of length 200 or lesser, the program will run as expected. However, if the length of user_argument is greater than 200, then the user_argument will get stored in the memory. But the problem in this case is that, since strcpy() does not perform input sanitization, the user_argument will overwrite some of the data in the memory that is next in sequence to the memory space reserved for the buffer variable. Due to this behavior, a carefully crafted input from the user can be a threat to the system. Threats can range from data corruption (have important data overwritten with garbage values) to giving shell access to unauthorized users.

3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.
Ans. The exploit implemented is a shell code which will provide the user access to the shell prompt with root(admin) privileges. The main idea is to place this shell code in the memory allocated for buffer and have the return address point back to the shell code, which on execution will open the shell prompt. Since we have a buffer of size 200, the shell code will occupy only a small portion of it. NOP is used to fill the remainder of the buffer till the return address pointer is reached. Reason is we dont want the assembler to do anything, till the return address pointer is read. When the pointer value is read, it is redirected to the buffer containing the shell code, which upon execution will give shell prompt access with admin privileges.

4. Added as an attachment.

5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?
Ans.
a. Performing a input length check when that value is going to be stored in the memory.
b. Imposing constraints on the kind of input that is expected from the user. i.e.Phone number inputs should allow only numbers (0-9), so that it will be difficult for the exploit to be crafted within this constraints.
c. Using a programming language where there is monitoring of memory allocation, or the whole memory allocation/deallocation is automated and the end user has no influence on it. For eg. Java

vulnerable2.c

1. Briefly describe the behavior of the program.
Ans. The program is such that it accepts any number of arguments. But it is understood that in order to gain access to the site of vulnerability i.e. the launch() method, one argument is mandatory. The argument should be of the format <***integer,string***>. Using the strtoul () method, the integer is separated from the string and are assigned to feed_count and cursor respectively. The cursor is then copied into the buffer using the memcpy() method.

2. Identify and describe the vulnerability as well as its implications.
Ans. Memcpy() method accepts only unsigned int as parameter. Whereas the input the user is permitted to enter can be positive or negative. Now, the buffer size is set at 528*20. Since there are no constraints on the size of the cursor or the range of the integers that a user can give as an input, the typecasting of a signed negative integer to an unsigned integer is exposed for exploitation. i.e. -1 in signed int equates to 0xffffffff in unsigned int. That is why all the conditions put in place can be circumvented to reach the memcpy() method, and execute a buffer overflow attack.

3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.
Ans. The exploit implemented is a shell code which will provide the user access to the shell prompt with root(admin) privileges. The main idea is to place this shell code in the memory allocated for buffer and have the return address point back to the shell code, which on execution will open the shell prompt. Since we have a buffer of size 528*20, th shell code will occupy only a small portion of it. NOP is used to fill the remainder of the buffer till the return address pointer is reached. Reason is we dont want the assembler to do anything, till the return address pointer is read. When the pointer value is read, it is redirected to the buffer containing the shell code, which upon execution will give shell prompt access with admin privileges. In order to reach the memcpy method(), we choose to use a signed negative int, which will satisfy all the criterion on the way to reach the memcpy() method.

4. Added as an attachment.

5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?
Ans.
a. Ensuring that all the instances of typecasting variables should not yield results that will bypass the constraints built around it.
b. Performing a input length check when that value is going to be stored in the memory.
c. Using a programming language where there is monitoring of memory allocation, or the whole memory allocation/deallocation is automated and the end user has no influence on it. For eg. Java

vulnerable3.c

1. Briefly describe the behavior of the program.

Ans. The program is such that it accepts any number of arguments. But it is understood that in order to gain access to the site of vulnerability i.e. the for loop in the strcpyn() method, one argument is mandatory. In tthis method, the user input is written into the memory. But a small error in the for loop constraints is creating an opening for exploitation to be made possible.

2. Identify and describe the vulnerability as well as its implications.

Ans. BUFFER_SIZE value is 192. However, if we observe closely the constraints of the for loop in the strcpyn() method, the loop iterates for 193 times. This leaves open room for making direct changes to the last byte of the base pointer of the stack. With this opening, and a carefully crafted input, an exploit can be given in as input to the method and its vulnerability be exploited. Due to this vulnerability, system can be modified to corrupt data or to acquire admin privileges via root access.

3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.

Ans. The exploit implemented is a shell code which will provide the user access to the shell prompt with root(admin) privileges. The main idea is to place this shell code in the memory allocated for buffer, have the last byte of the base pointer address changed to make it point to any of the NOPs in the buffer and change the word next to the aforementioned NOP to direct to the shell code. Since we have a buffer of size 192, the 193rd byte of our input needs to be calculated properly so that the link mentioned in the previous sentence gets realized . The reason for changing the word next to the aforementioned NOP is that the compiler after reading the base pointer reads the return address. Thus, with this weakmess now pretty obvious, the user input can be used as a means to get access to the buffer containing the shell code, which upon execution will give shell prompt access with admin privileges.

4. Added as an attachment.

5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

Ans. The vulnerability is caused by poor standards of coding. The for loop constraints need to be rigid, so that there is no flexibility involved to make it easy for exploits to the engineered.

Mystery Shellcode Analysis

00000000 xor ecx,ecx   resetting ecx to 0
00000002 mov ecx,0x21100f0e   setting ecx to 0x21100f0e
00000007 xor ecx,0x21212121   xor 0x21100f0e with 0x21212121 which gives 0x00312e2f
0000000d push ecx   pushing 0x00312e2f in the stack
0000000e xor ecx,ecx   resetting ecx to 0
00000010 mov ecx,0x514c550e   setting ecx to 0x514c550e
00000015 xor ecx,0x21212121   xor 0x514c550e with 0x21212121 which gives 0x7061742f
0000001b push ecx   pushing 0x7061742f in the stack
0000001c mov ebx,esp   setting ebx value to 0x7061742f
0000001e xor eax,eax   resetting eax to 0
00000020 xor ecx,ecx   resetting ecx to 0
00000022 xor edx,edx   resetting edx to 0
00000024 mov al,0x5   store 0x5 in al
00000026 mov cl,0x41   store 0x41 in cl
00000028 mov dh,0x1   store 0x1 in dh
0000002a mov dl,0xc0   store 0xc0 in dl
0000002c int 0x80   system call is made to the linux kernel by the program.
0000002e mov ebx,eax   set ebx to eax's value i.e. 0
00000030 xor ecx,ecx   resetting ecx to 0
00000032 mov ecx,0x2b010f44   setting ecx to 0x2b010f44
00000037 xor ecx,0x21212121   xor 0x2b010f44 with 0x21212121 which gives 0x0a202e65
0000003d push ecx   pushing 0x0a202e65 in the stack
0000003e xor ecx,ecx   resetting ecx to 0
00000040 mov ecx,0x4e49524c   setting ecx to 0x4e49524c
00000045 xor ecx,0x21212121   xor 0x4e49524c with 0x21212121 which gives 0x6f68736d
0000004b push ecx   pushing 0x6f68736d in the stack
0000004c xor ecx,ecx   resetting ecx to 0
0000004e mov ecx,0x5446010d   setting ecx to 0x5446010d
00000053 xor ecx,0x21212121   xor 0x5446010d with 0x21212121 which gives 0x7567202c
00000059 push ecx   pushing 0x7567202c in the stack

0000005a xor ecx,ecx   resetting ecx to 0
0000005c mov ecx,0x434e4b01   setting ecx to 0x434e4b01
00000061 xor ecx,0x21212121   xor 0x434e4b01 with 0x21212121 which gives 0x626f6a20
00000067 push ecx   pushing 0x626f6a20 in the stack
00000068 xor ecx,ecx   resetting ecx to 0
0000006a mov ecx,0x4442484f   setting ecx to 0x4442484f
0000006f xor ecx,0x21212121   xor 0x4442484f with 0x21212121 which gives 0x6563696e
00000075 push ecx   pushing 0x6563696e in the stack
00000076 mov ecx,esp   setting ecx to 0x6563696e
00000078 xor eax,eax   resetting eax to 0

0000007a mov al,0x4	store 0x4 in al
0000007c xor edx,edx	resetting edx to 0
0000007e mov dl,0x14	store 0x14 to dl
00000080 int 0x80	system call is made to the linux kernel
by the program	
00000082 xor eax,eax	resetting eax to 0
00000084 mov al,0x6	store 0x6 in al
00000086 xor ebx,ebx	resetting ebx to 0
00000088 xor eax,eax	resetting eax to 0
0000008a mov al,0x1	store 0x1 in al
0000008c int 0x80	system call is made to the linux kernel
by the program	

Final stack contents

0x6563696e <---top of the stack
0x626f6a20
0x7567202c
0x6f68736d
0x0a202e65
0x7061742f
0x00312e2f