Lab 2 || Deliverables

Files:
Lab
Code

OS: VM running Fedora 21.
NOTES:
• disable address space randomization:
kernel.randomize_va_space = 0

• This version does not use Exec Shield.
Z shell was installed via #yum install -y zsh

Linked zsh program to /bin/sh
COMPILE program with gcc **-fno-stack-protector** example.c

STEPS:
load shell code into memory so that vulnerable program can jump to it. Launch shell code stored in a buffer.

NOTE: file was missing string.h library.

[s@localhost]~/Documents/CMPE220_LAB2/LAB2% gcc -fno-stack-protector call_shellcode.c

2) compile vulnerable code (stack.c) Objective: Exploit vulnerability. make it set-root-uid as follows:

The program is vulnerable because: the **bof** function declares a local array of size 12, allocated on the stack. The size of the input string is 517 characters long and contains malicious code. Functions string copy (strcpy) does not guard against this, in fact, when using it, the programmer knows that to avoid overflow attacks, the size of the destination array should be of size long enough to hold the entire contents of the source string. This can override contents on the stack (e.g., erase saved state, etc.) and hence, corrupt the stack.

[s@localhost]~/Documents/CMPE220_LAB2/LAB2% su root
Mot de passe :
[root@localhost LAB2]# gcc -o stack -fno-stack-protector stack.c
[root@localhost LAB2]# chmod 4755 stack
[root@localhost LAB2]# exit

This creates a buffer overflow.

3) Objective: code the contents of “badfile" which will be loaded into string that generates buffer overflow such that, when loaded, this code will spawn a root shell! :)

NOTE : code given was not compatible. The program to launch shell was created in a C program shellcode.c, which was then compiled, the object code was generated, and the bytes were taken from object code to create array of code. This required changing the stack and exploit files.

% gcc -o shellcode shellcode.c
% objdump -d shellcode > dissamebly_shellcode.txt

Deliverables:

Complete Task 1,2 and 3 mentioned in Lab

- Answer questions asked in tasks
- Make report containing just the answers to questions
- **Main deliverable is to make video where you would be showing how you solved all 3 tasks**
- Please find 3 files needed for your lab as attachments

Lab explanation

3 files are given to you

-> "Shellcode" file to launch a shell and show you as how to launch shell for further lab programs

- "Vulnerable" file which has buffer overflow vulnerability and using which a shell can be launched
- "Exploit" file (Note: On Windows, this file can be reported by Antivirus as exploit due to its characteristics), which needs to be completed by you to generate "badfile" which would be read by "Vulnerable" file and help launch shell

Lab expectations: Solve and launch shell by 3 different methods:

Method 1: Simple attack and launch shell.

a) exploit.c writes exploit code to badfile, then it is compiled. This launches the shell code.
b) stack.c program loads badfile into buffer and then generates a buffer overflow to execute the code in the badfile.

* In this version, address randomization is OFF :
kernel.randomize_va_space = 0

* In this version, exec-shield is OFF (in fact, it is not included in Fedora 21 version).

* In this version, GCC Stack Guard in OFF, meaning code is compiled with option :
gcc -fno-stack-protector

command lines :
step1 : compile vulnerbale program first
% gcc -o stack -fno-stack-protector stack.c -z execstack
step 2 : compile exploit.c which loads contetns of badfile
% gcc -o exploit -fno-stack-protector exploit.c -z exectack
step3 : execute exploit file
% ./exploit
step4 : execute vulnerable program
% ./stack

Method 2: Same process again but with "Address Randomization turned ON"

Q&A :

Task 2: Address Randomization Turned ON.

* Can you get a shell? If not, what is the problem?

No. It did not work. The program returned a segmentation fault.

* How does address randomization make your attacks difficult? You should describe your observation and explanation.

To implement the attack, the code of the attack and pointer to the code is needed. Generating this address requires knowledge of the exact address of the string. This

address can be obtained by guessing using an offset or by other methods like gdb, getting objects dumps and examining the start address of a particular string, such as « AAAAAAA ». In our case, turning on address randomization makes guessing more difficult because every time the vulnerable program is executed, the position of the stack will vary.

However, by running the program in a loop. A shell was eventually spawned.

 # /sbin/sysctl -w kernel.randomize_va_space=2

$ sh -c "while [1]; do ./stack; done;"

Method 3: Same process again (As method 1) but with "Stack Guard ON"
OBSERVATIONS :

Task 3: Stack Guard

The same set-up as in Task 1 was conducted except that the vulnerable program and the exploit were compiled without the -fno-stack-protector' option. I expected to get an error, such as a segmentation fault, however, the shell with root privileges was spawned successfully!!