Samira C. Oliva Madrigal

OS: Fedora 24 (32-bit)

Lab 2 || Deliverables (Report)

- **Code: in zip file**
- **Videos: here**
- **Report**

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

/***/
Task 1: Simple attack and launch shell.

Prior settings were conducted:
- installing zsh shell and linking the zsh shell
- entering su mode to change virtual address space randomization via sysctl commands

Code explanation:
a) exploit.c fills buffer with NOP sled, shellcode, and address to the start of the buffer (over writes return address and makes PC point to return address of buffer).

b) stack.c reads badfile into buffer and then generates a buffer overflow to execute the code in the badfile.

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

* In this version, exec-shield is disabled using execstack is with -z.

* 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**
% **chmod 4755 stack**
% **exit**
step 2 : compile exploit.c
% **gcc -o exploit -fno-stack-protector exploit.c -z exectack**
step3 : execute exploit file
% **./exploit**
step4 : execute vulnerable program
% **./stack**
step5: root shell is spawned
#

VIDEO URL: https://youtu.be/y1LBpnthlAs

/***/
Task 2: Same process again but with "Address Randomization turned ON"

Q1: Can you get a shell? If not, what is the problem?

A: No. It did not work the first time. The program returned with a segmentation fault.

Q2: How does the address randomization make your attacks difficult? You should describe your observation and explanation.

A: 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 buffer to use in the exploit. This address can be obtained by guessing using (offset + stack pointer address) or by other methods like gdb, getting objects dumps and examinig the exact start address of the exploit buffer, using strings such as « AAAAAAA » which in the memory would be 41 41. and we would look for that in the object dump using **x/200xb $esp**. Using the given code in this assignment, turing address randomization ON 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, the shell is eventually spawned (duration can vary from a minutes to several minutes).

command lines:
% sysctl -w kernel.randomize_va_space=2
% sh -c "while true; do eval ./stack; done;"

At first, it took very long (you may say over 10 minutes, or 30 minutes and a shell was not spawned). When testing again on November 6, 2016, the shell was successfully spawned within several minutes and when tested again, within about 3 minutes. Lastly, a subsequent test to record the video, took close to 10-12 minutes to spawn the shell.

A different approach, such as using gdb could be used to get the exact start address of the buffer and start of the memory which would then just be inserted into the exploit program (no wait time).

VIDEO URL: https://youtu.be/aGhcHVRwcS0

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

command lines:
step1 : compile vulnerbale program first
% sysctl -w kernel.randomize_va_space=0
% gcc -o stack stack.c -z execstack
% chmod 4755 stack
% exit
step 2 : compile exploit.c which loads contetns of badfile
% gcc -o exploit exploit.c -z exectack
step3 : execute exploit file
% ./exploit
step4 : execute vulnerable program
% ./stack
step5: a shell is spawned

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

References for embedded assembly in C for GCC compiler:

Use any of the general purpose registers to store the stack pointer address: **EAX, EBX, ECX, EDX**
https://www.cs.dartmouth.edu/~sergey/cs258/tiny-guide-to-x86-assembly.pdf

Use inline assembly specific for gcc compiler: __asm__("instructions");
https://www.cs.uaf.edu/courses/cs301/2014-fall/notes/inline-assembly/

Must use extended asm: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

Quick review of instructions x86: https://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html

Stack: Computer Systems: A Programmer's Perspective (3nd Ed 2015), Randal E. Bryant & David R. O'Halloran, Pearson, ISBN 9780134092669

NASM, stack, and embedded assembly in C (reference 152 compiler design lectures)

Print the call stack…. :
https://www.gnu.org/software/libc/manual/html_node/Backtraces.html

```
//view machine code files
gcc -Og -o func.c //creates object file
objdump -d func.o //invokes dissembler output

gcc -Og -o test_exe test.c//creates exe file with all code to start and end program….
./test_exe //runs
objdump -d test_exe
```

VIDEO URL: https://youtu.be/yiUk7MawtTU

REFERENCES:

[1] https://www.youtube.com/watch?v=TuI2HyG8-iI

[2] https://www.youtube.com/watch?v=hJ8IwyhqzD4

[3] https://www.youtube.com/watch?v=j0CgtTX1Szw

[4] http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1.html

[5] https://dhavalkapil.com/blogs/Shellcode-Injection/

[6] https://www.youtube.com/watch?v=TuI2HyG8-iI