UOSEC Week 5: Shellcode injection

Frank farana@uoregon.edu
IRC kee

Adam pond2@uoregon.edu
IRC xe0

toin.irc.freenode.net #0x4f
Agenda

- Will Laney UO CISO
- BKP CTF Feb 27th-March 1st
- Symantec Tour Wednesday Feb 11th 11-1
- Participation reminder (CIS 433 students)
- Answer for Stack4 from last meeting
- Quick recap from last meeting
- Shellcode basics
- Write and use your own shellcode
BKP CTF

- Second largest CTF event all year
- Can participate remotely
- Great for first timers
- http://bostonkeyparty.net/
Symantec Tour

● Great networking opportunity
● More people = more leverage
● EMX travel 11am next Wednesday
Participation Reminder

- CIS 433 students only
- Bring your laptop or talk to me
Last time we joined our heros

https://exploit-exercises.com/protostar/stack4/

If you’re a real gangster, get it to print “code flow successfully changed” without a segfault :)

python -c "print 'A'*76 + '\xf4\x83\x04\x08'" | ./stack4

how to figure offset? 64 bytes for char buffer[64] + 8 bytes stack align + 4 byte ebp = 76 bytes junk
Quick recap from last meeting

- You can calculate offsets accurately by disassembling a binary rather than depending on source code.
- The function prologue contains hidden bytes (push ebp, stack alignment) you need to consider when calculating stack offsets.
What is shellcode

- Shellcode is any bytecode used in an exploit example \\xb0\\x01\\x31\\xdb\\xcd\\x80
  shellcode translates to `system("exit");`
What is shellcode

- Shellcode is any bytecode used in an exploit example: `\xb0\x01\x31\xdb\xcd\x80`
  \[\text{shellcode translates to } \text{system("exit")};\]
- Different types of shellcode such as local, remote, download and execute, and staged.
What is shellcode

- Shellcode is any bytecode used in an exploit example `\xb0\x01\x31\xdb\xcd\x80` shellcode translates to `system("exit");`
- Different types of shellcode such as local, remote, download and execute, and staged.
- Different types of encodings alphanumeric, null-free, percent.
Write your own shellcode or dl some

- Two real choices for shellcode you can either write your own or download some from an available shellcode database
Write your own pre-reqs

- In order to write your own shellcode you need basic x86 asm, a little c coding, knowledge of linux syscall, a 32bit linux OS, a disassembler (objdump or gdb), an assembler (nasm), and a text editor (vim)
Write your own pre-reqs

- In order to write your own shellcode you need basic x86 asm, a little c coding, knowledge of linux syscall, a 32bit linux OS, a dissembler (objdump or gdb), an assembler (nasm), and a text editor (vim)
- Big benefit of writing your own sc is that you know exactly what it is doing and allows customization for what you’re exploiting
Transform asm to shellcode

- Disassemble your asm using objdump -d file
Transform asm to shellcode

- Disassemble your asm using objdump -d file
- Create your shellcode by combining the opcodes aka "\xopcode1\xopcode2…\xopcodeN"
Transform asm to shellcode

- Disassemble your asm using objdump -d file
- Create your shellcode by combining the opcodes aka "\xopcode1\xopcode2…\xopcodeN"
- Test out your shellcode in sctest.c
Exercise write some shellcode

- Write your own shellcode that will write the message “uosec club 2015” followed by a new line
- Make sure your shellcode exits cleanly by calling exit at the end
- Test in sctest.c (or in protostar binary if you’re feeling froggy)
Answer shellcode exercise

- uosec.net/week5_answer
A lot of the time you will use shellcode it will be against C programs
The importance of NULL

- A lot of the time you will use shellcode it will be against C programs
- C strings look like

```c
char buf[] = "I’m a C string";
```
The importance of NULL

● A lot of the time you will use shellcode it will be against C programs
● C strings look like
  ```c
  char buf[] = "I'm a C string";
  ```
● What you don’t see is the terminating \0 character which represents a NULL
The importance of NULL

- A lot of the time you will use shellcode it will be against C programs
- C strings look like
  ```c
  char buf[] = "I’m a C string";
  ```
- What you don’t see is the terminating `\0` character which represents a NULL
- Thus a `\x00` byte in a C string ends it
NULL c-string example

sc="\xeb\x12\x5b\x31\xc0\x89\x43\x01\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\xe8\xe9\xff\xff\xff\xf58"

- Lets say we overflow a buffer using `gets()`
- `gets("\xeb\x12\x5b\x31\xc0\x89\x43\x01\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\xe8\xe9\xff\xff\xff\xf58")`
- Since C strings terminate at null bytes our shellcode is truncated and therefore fails
Game over man

GAME OVER MAN! GAME OVER!!
Exercise Stack5

https://exploit-exercises.com/protostar/stack5/