Advanced Cybersecurity Topics

19-20

Download Now!

http://bit.ly/ACT19VM

Schedule (Tentative)

- Shellcode Writing
- Format String
- Protection by pass
- RET to lib & ROP
- Heap Exploitation

16/09 -> 02/10

- Flipped Class

07/10 -> 27/11

- Reversing
- Symbolic Execution
- Dom based XSS
- Race Condition
- Serialization
- Unpacking
- Dynamic Malware Analysis

02/12 -> 23/12

Evaluation (Tentative)

A project can be discussed as an alternative if you cannot be physically in class for the lab/flipped parts

How to learn to exploit?

- Hands-on!
- 30-ish minutes of explanation / demo
- You try on your own!
- Ask questions!

What you need? (http://bit.ly/ACT19VM)

- Linux (Last ubuntu LTS highly recommended)
- x86 and x86_64
- GDB (pwngdb, peda, gef, etc…)
- pwntools (python2)
- ghidra (or IDA Pro)
- tmux (screen, terminator)

Binary Challenge Setup

- Challenges (https://training.jinblack.it/)
- Remote Service
  - Running on docker on ubuntu 18.04
- You get the binary
- Read file `“/chall/flag”`

High-level and Machine Code

Developer

```
#include <stdio.h>
#include <stdlib.h>

int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}

int main(int argc, char * argv[]) {
  int avar;
  int bvar;
  int cvar;
  char * str;
  avar = atoi(argv[1]);
  bvar = atoi(argv[2]);
  cvar = foo(avar, bvar);
  gets(str);
  puts(str);
  printf("foo(%d, %d) = %d\n", avar, bvar, cvar);
  return 0;
}
```

≠

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int32_t foo(int32_t a, int32_t b);

// From module:   layout.c
// Address range: 0x80484ac - 0x80484cd
// Line range:    5 - 10
int32_t foo(int32_t a, int32_t b) {
    int32_t c = 14 * (b + a); // 0x80484c4
    return c;
}

// From module:   layout.c
// Address range: 0x80484cf - 0x8048559
// Line range:    13 - 30
int main(int argc, char **argv) {
    int32_t apple = (int32_t)argv; // 0x80484d8
    int32_t str_as_i = atoi((int8_t *)*(int32_t *)(apple + 4));
    int32_t str_as_i2 = atoi((int8_t *)*(int32_t *)(apple + 8));
    int32_t banana = foo(str_as_i, str_as_i2); // 0x804850f
    gets(NULL);
    puts(NULL);
    printf("foo(%d, %d) = %d\n", str_as_i, str_as_i2, banana);
    return 0;
}
```

Compiler

```
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $32, %esp
        movl    12(%ebp), %eax
        addl    $4, %eax
        movl    (%eax), %eax
```

≠

```
push   %ebp
mov    %esp,%ebp
and    $0xfffffff0,%esp
sub    $0x20,%esp
mov    0xc(%ebp),%eax
add    $0x4,%eax
mov    (%eax),%eax
mov    %eax,(%esp)
call   80483b0 <atoi@plt>
mov    %eax,0x1c(%esp)
mov    0xc(%ebp),%eax
```

Decompiler

Assembler

```
0000000: 01111111 01000101 01001100 01000110 00000001 00000001
0000006: 00000001 00000000 00000000 00000000 00000000 00000000
000000c: 00000000 00000000 00000000 00000000 00000010 00000000
0000012: 00000011 00000000 00000001 00000000 00000000 00000000
0000018: 11000000 10000011 00000100 00001000 00110100 00000000
000001e: 00000000 00000000 10110100 00001100 00000000 00000000
0000024: 00000000 00000000 00000000 00000000 00110100 00000000
000002a: 00100000 00000000 00001000 00000000 00101000 00000000
```

Disassembler

Machine

No src! - What do I use?

- Objdump - Disasm
- radare2 - Disasm
- Binary Ninja - Disasm + Primitive Decompiler
- **GHIDRA - Disasm + Decompiler** (https://ghidra-sre.org/)
- rev.ng - Disasm + Decompiler (maybe one day)
- **IDA Pro - Disasm + Decompiler (de facto standard)**

32 bit vs 64 bit

- Registers
- Syscalls (https://w3challs.com/syscalls/):
  - x86 int 80
  - x86_64 syscall
- man is your friend (even google is fine)
  - `man 2 read`

Writing a Shellcode

- execute a shell!
- plan your shellcode:
  - `exec("/bin/sh")`
    - use an assembler

      (https://defuse.ca/online-x86-assembler.htm)

How to Assemble

- GCC (as)
- Nasm
- pwntools
- Online assembler.
  - https://defuse.ca/online-x86-assembler.htm\

Setup the environment

- Most similar env (ubuntu 18.04)
- debug tools (gdb)
- Scripting (pwntools)
- debug while running your script...

Debugging Challenges with GDB

Host the challenge:

```
socat TCP-LISTEN:4000,reuseaddr,fork EXEC:"./shellcode"
```

Connect your script. (NB You script should wait.)

`python x.py` (OR `ncat 127.0.0.1 4000`)

Attach with gdb:

```
ps aux | grep shellcode
```

```
sudo gdb attach 25209
```

Debugging Challenges with GDB the pwntools way

```
1. context.terminal = ['tmux', 'splitw', '-h']
2. # ssh = ssh("jinblack", "192.168.56.102")
3. r = process("./multistage")
4. gdb.attach(r,'''
5. # b * 0x004000b0
6. # b *0x4000DD
7. ''')
8. raw_input("wait")
```

Writing a Shellcode - Multi Stage

- If you do not have space, you make space.
- plan your shellcode:
  - Stage One
    - `read(·,·,·) #second stage`
  - Stage Two:
    - `exec("/bin/sh")`

Writing a Shellcode - Fork Server

- fd 0 or 1 are not always the way.
- plan your shellcode:
  - `dup2(·,·,·)`
  - `exec("/bin/sh")`

Writing a Shellcode open read write

- you may need to read bpf filters (https://github.com/david942j/seccomp-tools)
- plan your shellcode:
  - `open("/flag")`
  - `read(·,·,·)`
  - `write(·,·,·)`

Writing a Shellcode - Reverse Shell

- Connect to remote host.
- plan your shellcode:
  - `socket(·,·,·)`
  - `dup2(·,·,·)`
  - `connect(·,·,·)`
  - `exec(“/bin/sh”)`