Software Security Course

Extra Workshop: Black Box Vulnerability Research

Dimitrios A. Glynos
{ daglyn at unipi.gr }

Department of Informatics
University of Piraeus

June 29th, 2024

Part I

Exploring the binary

File format

- The ’file’ utility

```
$ file foo
foo: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV)
dynamically linked (uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=0x628d57caa90b9cb4d373105a9b17da72aa4bb0d7,
not stripped
$
```

Mapping the application functionality

Execute the application in order to identify

- core functionalities
- application states
- user interaction points
- input / outputs
- assets that would be vulnerable if a bug in the application was exploited
- high level security controls
- file permissions
- required / obtained privileges

Dependencies to dynamic libraries

- Use ’ldd’ to spot dependencies to dynamic libraries

```
$ ldd foo
        linux-gate.so.1 =>  (0x00602000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0x0011000
        /lib/ld-linux.so.2 (0x005c8000)
$
```

Incorporated static libraries

- Look for interesting symbols with ’nm’ or ’objdump’

```
$ objdump -d foo
...
804c315: e8 b6 00 00 00 call 804c3d0 <SSL_library_init>
...
$
```

Execution tracing

- Use 'strace' to trace system calls

```
$ strace -fF ./foo
...
[pid 16155] execve("/bin/mount", ... ) = 0
...
$
```

- Use 'ltrace' to trace library calls

```
$ ltrace ./foo
...
strcpy(0xbfc0343d, 0xbfc0451a) = 0xbfc0343d
...
$
```

Part II

Searching for vulnerabilities

Identify parameters that can be controlled

- Inputs from user
- (Modifiable) files
- Environment variables
- See related material from previous lectures!

Fuzz testing

- Test wild input values on user-controlled parameters
- Use automated fuzzers
  - Protocol fuzzing with ’peach’
  - File fuzzing with ’honggfuzz’

Checking whether a crash due to memory corruption is interesting / exploitable

Use a debugger (like 'gdb') to check if:

- EIP was set to a user controllable value
- the input overwrote control data on the stack (e.g. saved stack frame / EIP)
- the input overwrote control data on the heap (e.g. free list pointers)
- the input overwrote sensitive data on stack / heap (e.g. function pointers)

Identify and exploit functions with known weaknesses

- a *strcpy* buffer overflow
- a *memcpy* buffer overflow
- a *printf* with a format string bug
- ...

Study interesting paths using reverse engineering

- Use a disassembler (like ’objdump’) to see how interesting code is invoked
- Use a debugger (like ’gdb’) to follow interesting code paths

Investigate the type of vulnerability

- See `/proc/{PID}/maps` to figure out if an oveflown buffer belongs to the heap or stack

```
...
09133000-09431000 rw-p 00000000 00:00 0 [heap]
...
bfc44000-bfc65000 rw-p 00000000 00:00 0 [stack]
...
```

Recording a vulnerability

- Record the vulnerable code
- Record the vulnerability type
- Record the trigger
- Keep in mind that multiple weaknesses may come in handy during the exploitation phase (e.g. address leak and memory corruption in pages close to the leaked address)

Part III

Exploiting a vulnerability

Identify the security controls protecting the vulnerability

- Does the OS + binary employ ASLR?
- Is the vulnerable buffer protected by a canary?
- Is the vulnerable code accessible with the privileges available?
- Do we need to exploit another vulnerability to reach the vulnerable code?

ASLR

- Use ’paxtest’ to check for OS ASLR capabilities
- Use ’readelf’ to check if there are sections that will be loaded at fixed memory addresses
- Use `/proc/{PID}/maps` to check if there are memory pages with interesting attributes (e.g. writable + executable)
- Abuse the allocator to jump to a memory address that is likely to have controllable data

Canary checks

- Use a disassembler to look for stack canary checks

```
8048571: 8b 45 f4                   mov    -0xc(%ebp),%eax
8048574: 65 33 05 14 00 00 00       xor    %gs:0x14,%eax
804857b: 74 05                      je     8048582 <myfunc+0x2e
804857d: e8 fe fe ff ff     call   8048480 <__stack_chk_fa
```

- If a known allocator is used for the heap, check for signs of canaries or guard pages

Code requiring special privileges

```
-rwsr-x--- 1 root audit 7335 Apr 26 15:11 foo
```

- Only users of group ’audit’ can execute the vulnerable binary

Stack buffer overflow exploitation

- Fill stack with pattern
- `strcpy(buf, .. ABCDEFGHIJKLMNOPQRSTUVWXYZ .. )`

```
high address  ^   [ Function Parameters ] UVWX
              |   [    Return Address   ] QRST
              |   [ Saved Frame Pointer ] MNOP
              |   [   Local Variables   ] IJKL
              |   [   Local Variables   ] EFGH
 low address  |   [   Local Variables   ] ABCD
```

- Use a debugger (e.g. 'gdb') to identify the portion of the pattern that overwrote the return address

Stack buffer overflow exploitation

```
Program received signal SIGSEGV, Segmentation fault.
0x08048677 in main ()
(gdb) x/i 0x08048677
=> 0x8048677 <main+198>:     ret
(gdb) x/a $esp
0xbf9c496c:        0x54535251
```

- 0x54535251 stands for 'QRST' on x86 CPUs

Stack buffer overflow exploitation

- Replace 'QRST' with address of code to jump to
- For example if `0x08031234` always contains code that spawns a shell:
  - `strcpy(buf, .. ABCDEFGHIJKLMNOP` **`\x34\x12\x03\x08`** `UVWXYZ .. )`
- Avoid memory addresses containing NULL bytes if the vulnerability is caused by a string processing function (strcpy etc.)
- If no useful code is readily available to jump to, implement exploit via ROP chain

Post exploitation

- Is the vulnerable code protected by a sandbox?
- Does the exploited process have enough privileges to gain access to the target asset?
- ...