Lecture 4
Exploiting. Shellcodes

Computer and Network Security

October 21, 2019

Computer Science and Engineering Department

Bugs and Vulnerabilities

- bugs: misbehaving sofware
- vulnerability: misbehaviour that can benefit an attacker
- exploiting: turning a vulnerability into an advantage for the attacker
- auditing: analyzing an application to determine its vulnerabilities

Why Bugs and Vulnerabilities?

- developer carelessness or ignorance
- poor development process
- poor design
- platform (hardware, OS, libraries) issues
- lack of resources

Preventing Vulnerabilities

- development process: defensive programming, code review, code audit
- design with security in mind
- audit systems, penetration testing
- security-centered training
- invest resources

Security Attacks

- eavesdropping, impersonating
- password breaking
- denial of service
- exploiting

Exploiting

- exploiting vulnerabilities
- focus is controlling the system (root account)
- an intermediary step is gaining shell access to user
- privilege escalation

Why Exploiting?

- money
- fame
- challenge
- fun
- political, ideological
- find security holes and fix them (ethical hacking)

Detect/Prevent Exploiting

- monitoring
- update software
- stay connected
- in-depth security
- honeypots
- state of mind: “it will happen”

Types of Exploits

- local exploit
- remote exploit
- user space exploit
- kernel space exploit

Runtime Application Exploiting

- find vulnerability in process runtime: memory, use of resources
- alter normal execution pattern
- aim for: getting a shell, getting access to resources, information leak, crash application, denial of service
- usually tamper with process memory and bad ways of memory management
- special focus on string management functions, input/output, pointers

Runtime Exploit Components

- preparatory phase
- shellcode
- triggering phase

Types of Runtime Application Vulnerabilities

- buffer overflow (on stack or heap)
- integer overflow
- race conditions
- string formatting

Buffer Overflow

- write beyond buffer limits
- stack-based overflow: overwrite variable, return address or function pointer
- heap overflow: corrupt dynamically allocated memory

Shellcode

- sequence of machine level instructions
- stored in memory at a convenient address
- executed when requested by jumping at the start address

Shellcode Objectives

- typically the goal is to create a shell (if possible, with root privilege)
- may be any useful binary code execution, such as starting a client socket, or reading or writing a file, or sending a file over the network

Shellcode Samples

- `http://www.shell-storm.org/shellcode/`
- hexadecimal form for exec-ing a shell process
- also dubbed payload

Shellcode Types

- spawn shell using `execve` syscall
- use `setresuid` to restore root privileges (for setuid-enabled programs)
- port-binding shellcode: create listener socket, accept connections, duplicate file descriptors and spawn shell
- connect-back shellcode: create client socket and connect to remote listener socket (accesible and controled by attacker), duplicate file descriptors and spawn shell

Creating a Shellcode

- may be done in C but it is recommended to do it in assembly
  - allows shorter shellcodes
  - complete control over the end result (binary machine code)
- need to use syscalls for `execve`, `setresuid`, `dup2` and others
- need to place the `/bin/sh` string in memory (or other strings) and pass it as argument to syscall

Using Syscalls in Linux on x86

- eax stores the syscall number
- ebx, ecx, edx, esi, edi store syscall arguments
- use int 0x80 to issue syscall
- syscall numbers in /usr/include/asm/unistd_32.h

setresuid(0, 0, 0) & exit(1)

```
# Fill eax, ebx, ecx and edx with zeros.
xor %eax, %eax
xor %ebx, %ebx
xor %ecx, %ecx
xor %edx, %edx
mov $164, %al                 # Put 164 (setresuid syscall no) in eax.
int $0x80                    # Issue syscall: setresuid(0, 0, 0).
```

```
xor %eax, %eax               # Fill eax with zeros.
xor %ebx, %ebx               # Fill ebx with zeros.
mov $1, %bl                   # Put 1 (EXIT_FAILURE) in ebx (only one byte).
mov $252, %al                 # Put 252 (exit_group syscall no) in eax.
int $0x80                    # Issue syscall.
```

Wrapper for Creating/Testing a Shellcode

Assembly Wrapper

```
.globl main

main:
        # Prepare registers an syscall arguments.
        # int $0x80     # Do syscall.
```

Assembly Shellcode Sample

```
.globl main

main:
        xor %eax, %eax   # Fill eax with zeros.
        xor %ebx, %ebx   # Fill ebx with zeros.
        mov $1, %bl       # Put 1 (EXIT_FAILURE) in ebx (only one byte).
        mov $252, %al     # Put exit_group syscall no in eax.
        int $0x80         # Issue syscall.
```

Building a Shellcode Wrapper

Makefile

```make
ASFLAGS = -march=i386 --32
CFLAGS = -Wall -m32
LDFLAGS = -m32

.PHONY: all clean

all: shellcode-wrapper-exit

shellcode-wrapper-exit: shellcode-wrapper-exit.o

shellcode-wrapper-exit.o: shellcode-wrapper-exit.s

clean:
	-rm -f shellcode-wrapper-exit shellcode-wrapper-exit.o *~
```

Extracting Hex Data

- actual shellcode is the machine code instruction
- use `objdump` on the object file and process the result
- use `echo -en` above to print in binary form

Using `objdump` to extract hex data

```
for i in $(objdump -d <module-name>.o | tr ’\t’ ’ ’ | tr ’ ’ ’\n’

    | egrep ’ˆ[0-9a-f]2$’) ; do echo -n "\x$i" ; done
```

- the reverse is achievable (getting the assembly mnemonics from hex)

Using `objdump` to extract hex data

```
echo -en "hexadecimal data" > shellcode
objdump -b binary -m i386 -D shellcode
```

Shellcode Constraints

- due to input data filtering
- small code
- null-free
- position-independent
- alphanumeric (not always)
- more on the next lecture

Null-free

- required when dealing with null-terminated strings
- BAD: `mov $1, %eax`
  - uses null bytes
  - `\xb8\x01\x00\x00\x00`
- GOOD: `xor %eax, %eax` + `inc %eax`
  - doesn't use null bytes
  - `\x31\xc0\x40`
- BAD: `mov $100, %eax`
  - uses null bytes
  - `\xb8\x64\x00\x00\x00`
- GOOD: `xor %eax, %eax` + `mov $100, %al`
  - doesn't use null bytes
  - `\x31\xc0\xb0\x64`

Using the Local Stack

- place shellcode in local buffer on stack
- rewrite return address to point to beginning of the buffer on the stack
- may need NOPs if exact address is not known
- unable to be done if stack is non-executable

Using an Environment Variable

- initialize an environment variable with the shellcode string
- environment variable is placed on the stack of main
- may be large enough to store large shellcodes
- unable to be done if stack is non-executable
- more on the next lecture

Using the Heap

- place the shellcode on the heap
- requires a heap buffer overflow
- made difficult by ASLR and non-executable flags

Triggering Shellcodes

- stack buffer overflow
  - overwrite return address and point to address on stack or environment variable
  - overwrite local pointer and point to address on stack or environment variable
- heap buffer overflow
  - overwrites metadata pointers for heap allocated data

Keywords

- bugs
- vulnerabilities
- exploit
- shellcode
- shellcode construction
- shellcode triggering
- shellcode placing
- syscall
- null
- stack buffer overflow
- heap buffer overflow
- pwntools

Useful Links

- http://www.blackhatlibrary.net/Category:Shellcode
- http://www.shell-storm.org/shellcode/
- http://www.metasploit.com/
- https://github.com/Gallopsled/pwntools
- https://docs.pwntools.com/en/stable/

References

- The Ethical Hacker's Handbook, 3rd Edition
  - Chapter 13 & 14
- A Guide to Kernel Exploitation
  - Chapter 1: From User-Land to Kernel-Land Attacks
- The Art of Exploitation, 2nd Edition
  - Chapter 0x500. Shellcode
- Hacking Exposed. Malware and Rootkits
  - Part II: Rootkits