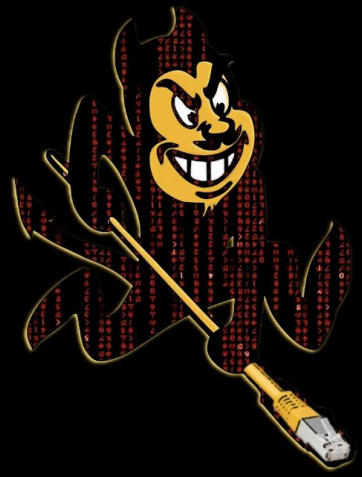
Computer Systems Security

CSE 466
Fall 2018
Yan Shoshitaishvili

http://pwn.college
http://groups.google.com/group/cse-466
https://goo.gl/NuCDTD

Announcements



PWNDEVILS

Who? The pwndevils, ASU's resident hacking club!
When? Tuesday 4:30pm and Thursday 4pm, and CTFs on weekends.
Where? BYENG 420
Web? pwndevils.com

CTF THIS WEEKEND: 5pm Friday in BYENG 209, until 5pm Sunday.

Homework 1 Retrospective

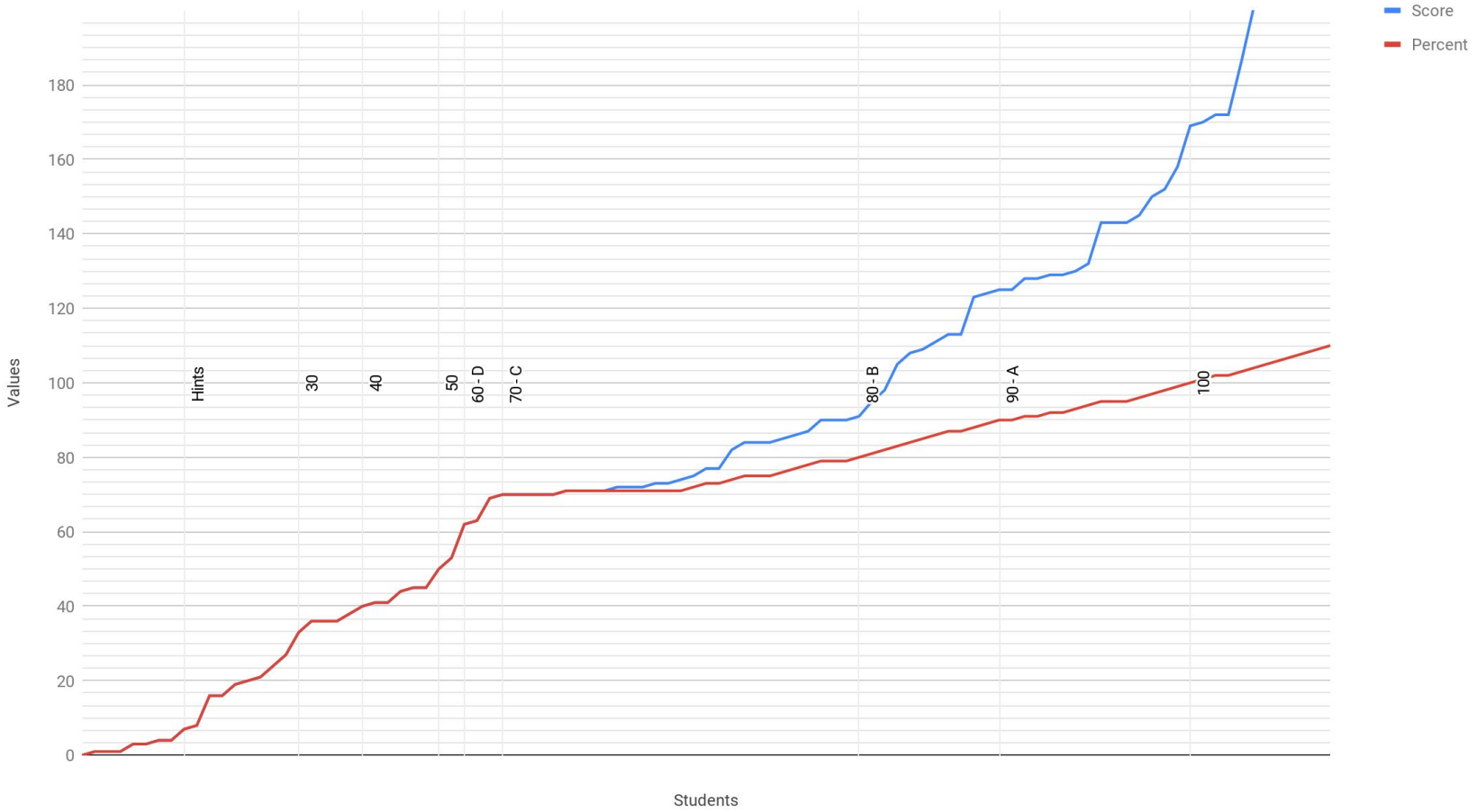
Statistics

97 graded students (1 auditing)

14 A+ (>= 99%)
8 A (>= 90%)
12 B (>= 80%)
31 C (>= 70%)
3 D (>= 60%)
29 E (< 60%)

Curve took effect right at 71 points (61 students made it).

Hw1 Scores and Percents



Top 3

Master hackers:

1. tumblebelly - 1337 points
2. bozaloshtsh - 339 points
3. yan_imitator - 251 points

How did they do it?

Standard progression

1. Panic.
2. Learning.
3. Automation.
4. Mastery.

Step 1: Panic

This is normal.

There are situations in life where the path forward isn't fully clear.

- Hacking is one.
- Research is another.
- Health issues are a third.

Some panic at the beginning of the process is healthy.

> *"It's ok to lose to opponent. It's never okay to lose to fear" - Mr. Miyagi*

Step 2: Learning

Many learning resources:

- man pages!!!!
- program --help
- **google is your friend**
- online writeups for similar things
  - i.e., "The Ultimate A To Z List of Linux Commands"
- slowly develop intuition
  - understand similar groups of programs
  - understand how to control them

"First learn stand, then learn fly. Nature rule Daniel son, not mine" - Mr. Miyagi

Step 3: Automation

Eventually, you'll notice that you repeat a number of things when playing with a program:

- program /flag
- program /flag /asdf; cat /asdf
- program /flag /dev/stdout
- program /dev/stdout /flag
- program -a /flag
- program -b /flag
- etc....

This is automatable!

2nd and **3rd** place automated --- did you?

Step 3: Automation

Props to: bozaloshtsh

Their automation would:

- choose a program
- call `program --help`
- parse the output to get a list of options
- call `program --option /flag` for every option
- look for the flag

Final flag count: 339

Step 3: Automation

Quick tips:

```
r.interactive()
r.readuntil("blah", timeout=5) or fail()
```

Step 4: Mastery

This one is hard to teach.

Props to: tumblebelly

In this assignment: `/proc` filesystem is accessible.

- /proc/$PID and /proc/self point to process resources
  - /proc/self/fd/0 is standard input, /proc/self/fd/1 is standard output
  - /proc/self/cmdline is the commandline that the process was called with
  - **/proc/self/cwd is the working directory**
  - **/proc/self/root is the root directory of the filesystem**
- Using these, you can get a "flag multiplier".

"Only root karate come from Miyagi. Just like bonsai choose own way grow because root strong you choose own way do karate same reason." - Mr. Miyagi

Step 4: Mastery

Multipliers for **`/bin/cat`**:

- `/bin/cat`
- `/proc/self/cwd/bin/cat`
- `/proc/self/root/bin/cat`
- `/proc/1/cwd/bin/cat`
- `/proc/1/root/bin/cat`
- `/proc/thread-self/cwd/bin/cat`
- `/proc/thread-self/root/bin/cat`
- `/proc/N/cwd/bin/cat`
- `/proc/N/root/bin/cat`
- `/proc/N/task/N/bin/cat`
- `additional versions of all the core utils scattered around the system`

tumblebelly got 19 flags using **`/bin/cat`** alone!

Step 4: Mastery

Other cool hacks:

- program @/flag
- #including the flag in various language formats (C, C++, etc).
- abusing /dev/stdout

Anyone has other cool hacks?

Bug bounties!

Two bug bounties submitted.

Bug bounty #1: we noobed up ssh - 50 percentage points

- when you'd ssh in, we would run the homework via hw1's .bash_profile
- this was bypassable simply by specifying an alternate command to ssh

```
ssh cse466.pwn.college ls /
```

- noob mistake on our part (next assignment: no ssh!)

Bug bounty #2: readline - 30 percentage points

- we used readline to make interacting with the terminal more comfortable
- readline supports auto-complete requests (on MacOS, option-tab)
- the default autocomplete behavior is to autocomplete file paths *in the context of the out-of-docker wrapper script*, allowing users to list directories outside of the docker.

What's our take-away here?

CSE 466 Week 2

Linux in Depth

This Week

We're going in-depth through the lifecycle of a Linux program:

```
cat /flag
```

cat /flag

1. A process is created.
2. Cat is loaded.
    a. binary
    b. shared libraries
3. Cat is initialized.
    a. shared library initializers
    b. binary initializers
4. Cat is launched.
    a. __libc_start_main()
    b. main()
5. Cat reads its arguments and environment.
6. Cat does its thing:
    a. opens /flag
    b. reads the data
    c. writes the data to stdout
7. Cat terminates.

cat /flag

1. **A process is created.**
2. Cat is loaded.
    a. binary
    b. shared libraries
3. Cat is initialized.
    a. shared library initializers
    b. binary initializers
4. Cat is launched.
    a. __libc_start_main()
    b. main()
5. Cat reads its arguments and environment.
6. Cat does its thing:
    a. opens /flag
    b. reads the data
    c. writes the data to stdout
7. Cat terminates.

A process is created

Every Linux process has:

- state (running, waiting, stopped, zombie)
- priority (and other scheduling information)
- parent, siblings, children
- shared resources (files, pipes, sockets)
- virtual memory space
- security context
  - effective uid and gid
  - saved uid and gid
  - capabilities

Every process is created by another process, and provided a process identity containing the above items.

A process is created

Every Linux process has:

- state (running, waiting, stopped, zombie)
- priority (and other scheduling information)
- parent, siblings, children
- shared resources (files, pipes, sockets)
- virtual memory space
- security context
  - effective uid and gid
  - saved uid and gid
  - **capabilities**

Every process is created by another process, and provided a process identity containing the above items.

Capabilities

Capabilities allow processes (privileged) access to system resources. root, by default, has **all capabilities**, such as:

CAP_DAC_OVERRIDE and **CAP_DAC_READ_SEARCH** - override file permission checks
CAP_KILL - bypass permission checks for sending signals to processes
CAP_NET_ADMIN - administer network settings (ip addresses, firewall, routing)
CAP_NET_RAW - raw network access (spoof packets, etc)
CAP_SETUID and **CAP_SETGID** - arbitrary manipulation of process UID and GID
CAP_SYS_ADMIN - tons of administrative functionality
CAP_SYS_BOOT - reboot the system
CAP_SYS_MODULE - load drivers
... and many others (check out `man capabilities` for the full set)

Normal users have no capabilities by default. However, you can set capabilities on a *per-program* basis using setcap and getcap. This acts similar to SUID and SGID bits.

Remember: *processes* are what have permissions and capabilities, files just have metadata. Demo: capabilities, docker capability reductions.

cat /flag

1. A process is created.
2. **Cat is loaded.**
    a. binary
    b. shared libraries
3. Cat is initialized.
    a. shared library initializers
    b. binary initializers
4. Cat is launched.
    a. __libc_start_main()
    b. main()
5. Cat reads its arguments and environment.
6. Cat does its thing:
    a. opens /flag
    b. reads the data
    c. writes the data to stdout
7. Cat terminates.

Cat is loaded.

```
yans@cse466 ~ $ file /bin/cat
/bin/cat: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=747e524bc20d33ce25ed4aea108e3025e5c3b78f, stripped
```

Cat is loaded.

```
yans@cse466 ~ $ file /bin/cat
/bin/cat: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=747e524bc20d33ce25ed4aea108e3025e5c3b78f, stripped
```

Cat is loaded.

```
yans@cse466 ~ $ file /bin/cat
/bin/cat: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=747e524bc20d33ce25ed4aea108e3025e5c3b78f, stripped
```

What is an ELF?

ELF is a binary file format.

Contains the program and its data.
Describes how the program should be loaded (*program/segment headers*).
Contains metadata describing program components (*section headers*).

Several ways to dig in:

- readelf
- objdump
- kaitai struct (https://ide.kaitai.io/)
- patchelf
- objcopy
- gcc
- nm
- strip

Interesting Parts

- Program headers.
- Section headers.
- dynamic interpreter preference (more on this later)
- rpath for library search (more on this later)
- Needed libraries.
- Import symbols.
- Export symbols.
- Relocation entries.
- The actual content!
    - objcopy

The loading process

1. File access checks (CAP_DAC_OVERRIDE, then filesystem perms).
2. A new process entry is created.
3. The binary is loaded.
    a. the rise of Position Independent Executables
4. The libraries are located.
    a. LD_PRELOAD environment variable, and anything in /etc/ld.so.preload
        i. (LD_PRELOAD is functionally ignored for setuid binaries)
    b. DT_RPATH specified in the binary file (can be modified with patchelf)
    c. LD_LIBRARY_PATH environment variable (can be set in the shell)
    d. DT_RUNPATH specified in the binary file (can be modified with patchelf)
    e. system-wide configuration (/etc/ld.so.conf)
    f. /lib and /usr/lib
5. The libraries are loaded.
    a. conceptually the same as any other binary (including needing other libraries!)
    b. relocations updated

Process memory space

Each Linux process has a *memory space*. It contains:

- the binary
- the libraries
- the "heap" (for dynamically allocated memory)
- the "stack" (for function local variables)
- any memory specifically mapped my the program
- some helper regions
- kernel code in the "upper half" of memory (above 0x8000000000000000 on 64-bit architectures), inaccessible to the process

This *virtual* memory, which is dedicated to your process, is backed by your machine's *physical* memory, which is shared among the whole system.

You can see this whole space by looking at `/proc/self/maps`

Good resource:
https://gist.github.com/CMCDragonkai/10ab53654b2aa6ce55c11cfc5b2432a4

What performs the actual loading?

Process loading is done by the ELF interpreter specified in the binary.

```
    $ readelf -a /bin/cat | grep interpret
    [Requesting program interpreter:
/lib64/ld-linux-x86-64.so.2]
```

Colloquially known as "the loader".

Can be overridden: `/lib64/ld-linux-x86-64.so.2 /bin/cat /flag`
Or changed permanently: `patchelf --set-intepreter`

The Standard C Library

libc.so is linked by almost every process.

Provides functionality you take for granted:

- printf()
- scanf()
- socket()
- atoi()
- malloc()
- free()
- xdr_keystatus()

Good resource:
https://gist.github.com/CMCDragonkai/10ab53654b2aa6ce55c11cfc5b2432a4

cat /flag

1. A process is created.
2. Cat is loaded.
    a. binary
    b. shared libraries
3. **Cat is initialized.**
    a. shared library initializers
    b. binary initializers
4. Cat is launched.
    a. __libc_start_main()
    b. main()
5. Cat reads its arguments and environment.
6. Cat does its thing:
    a. opens /flag
    b. reads the data
    c. writes the data to stdout
7. Cat terminates.

Cat is initialized.

Every ELF binary can specify *constructors*, which are functions that run before the program is actually launched.

For example, depending on the version, libc can initialize memory regions for dynamic allocations (malloc/free) when the program launches.

You can specify your own!

```
__attribute__((constructor)) void haha()
{
      puts("Hello world!");
}
```

Demo: LD_PRELOAD and constructors.

cat /flag

1. A process is created.
2. Cat is loaded.
    a. binary
    b. shared libraries
3. Cat is initialized.
    a. shared library initializers
    b. binary initializers
4. **Cat is launched.**
    a. __libc_start_main()
    b. main()
5. Cat reads its arguments and environment.
6. Cat does its thing:
    a. opens /flag
    b. reads the data
    c. writes the data to stdout
7. Cat terminates.

Cat is launched.

A normal ELF automatically calls __libc_start_main() in libc, which in turn calls the program's main() function.

Your code is running!

Now what?

cat /flag

1. A process is created.
2. Cat is loaded.
    a. binary
    b. shared libraries
3. Cat is initialized.
    a. shared library initializers
    b. binary initializers
4. Cat is launched.
    a. __libc_start_main()
    b. main()
5. **Cat reads its arguments and environment.**
6. Cat does its thing:
    a. opens /flag
    b. reads the data
    c. writes the data to stdout
7. Cat terminates.

Cat reads its arguments and environment.

```
int main(int argc, void **argv, void **envp);
```

Your processes' entire input from the outside world, at launch, comprises of:

- the loaded objects (binaries and libraries)
- command-line arguments in argv
- "environment" in envp

Of course, programs need to keep interacting with the outside world.

(demo: environment)

cat /flag

1. A process is created.
2. Cat is loaded.
    a. binary
    b. shared libraries
3. Cat is initialized.
    a. shared library initializers
    b. binary initializers
4. Cat is launched.
    a. __libc_start_main()
    b. main()
5. Cat reads its arguments and environment.
6. **Cat does its thing:**
    a. opens /flag
    b. reads the data
    c. writes the data to stdout
7. Cat terminates.

Using library functions

The binary's *import symbols* have to be resolved using the libraries' *export symbols*.

In the past, this was an on-demand process and carried great peril.

In modern times, this is all done when the binary is loaded, and is much safer.

We'll explore this further in the future.

Interacting with the environment

Almost all programs have to interact with the outside world!

This is primarily done via *system calls* (`man syscalls`). Each system call is well-documented in section 2 of the man pages (i.e., `man 2 open`).

We can trace process system calls using `strace`.

System Calls

System calls have very well-defined interfaces that very rarely change.

There are over 300 system calls in Linux. Here are some examples:

`int open(const char *pathname, int flags)` - returns a file new file descriptor of the open file (also shows up in /proc/self/fd!)
`ssize_t read(int fd, void *buf, size_t count)` - reads data from the file descriptor
`ssize_t write(int fd, void *buf, size_t count)` - writes data to the file descriptor
`pid_t fork()` - forks off an *identical* child process. Returns 0 if you're the child and the PID of the child if you're the parent.
`int execve(const char *filename, char **argv, char **envp)` - *replaces* your process with another program.
`pid_t wait(int *wstatus)` - wait on a child process to exit, return the PID, and write its status into *wstatus.

Typical signal combinations:

- fork, execve, wait (think: a shell)
- open, read, write (cat)

Signals

System calls are a way for a process to call into the OS. What about the other way around?

Enter: signals. Relevant process calls:

`sighandler_t signal(int signum, sighandler_t handler)` - register a signal handler
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)` - more modern way of registering a signal handler
`int kill(pid_t pid, int sig)` - send a signal to a process.

A signal causes execution to pause and the appropriate handler to run.

Signal handlers are functions that take a single argument: the signal number.

If there is no handler for a signal, the default action is used (often, kill).

SIGKILL (signal 9) and SIGSTOP (signal 19) cannot be handled.

Signals

Full list in section 7 of man (`man 7 signal`) and `kill -l`. Common signals:

SIGHUP	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	Term	Interrupt from keyboard
SIGQUIT	Core	Quit from keyboard
SIGILL	Core	Illegal Instruction
SIGABRT	Core	Abort signal from abort(3)
SIGFPE	Core	Floating-point exception
SIGKILL	Term	Kill signal
SIGSEGV	Core	Invalid memory reference
SIGPIPE	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	Term	Timer signal from alarm(2)
SIGTERM	Term	Termination signal
SIGUSR1	Term	User-defined signal 1
SIGUSR2	Term	User-defined signal 2
SIGCHLD	Ign	Child stopped or terminated
SIGCONT	Cont	Continue if stopped
SIGSTOP	Stop	Stop process
SIGTSTP	Stop	Stop typed at terminal
SIGTTIN	Stop	Terminal input for background process
SIGTTOU	Stop	Terminal output for background process

Signals

System calls are a way for a process to call into the OS. What about the other way around?

Enter: signals. Relevant process calls:

`sighandler_t signal(int signum, sighandler_t handler)` - register a signal handler
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)` - more modern way of registering a signal handler
`int kill(pid_t pid, int sig)` - send a signal to a process.

When a process receives a signal, its execution is paused and the handler for the appropriate signal runs.

Signal handlers are functions that take a single argument: the signal number.

Shared memory

Another way of interacting with the outside world is by sharing memory with other processes.

Requires system calls to establish, but once established, communication happens without system calls.

Easy way: use a shared memory-mapped file in /dev/shm.

cat /flag

1. A process is created.
2. Cat is loaded.
    a. binary
    b. shared libraries
3. Cat is initialized.
    a. shared library initializers
    b. binary initializers
4. Cat is launched.
    a. __libc_start_main()
    b. main()
5. Cat reads its arguments and environment.
6. Cat does its thing:
    a. opens /flag
    b. reads the data
    c. writes the data to stdout
7. **Cat terminates.**

Process termination

Processes terminate by one of two ways:

1. Receiving an unhandled signal.
2. Calling the exit() system call: `int exit(int status)`

All processes must be "reaped":

- after termination, they will remain in a zombie state until they are wait()ed on by their parent.
- When this happens, their exit code will be returned to the parent, and the process will be freed.
- If their parent dies without wait()ing on them, they are re-parented to PID 1 and will stay there until they're cleaned up (or their container is terminated).

cat /flag

1. A process is created.
2. Cat is loaded.
    a. binary
    b. shared libraries
3. Cat is initialized.
    a. shared library initializers
    b. binary initializers
4. Cat is launched.
    a. __libc_start_main()
    b. main()
5. Cat reads its arguments and environment.
6. Cat does its thing:
    a. opens /flag
    b. reads the data
    c. writes the data to stdout
7. Cat terminates.

Homework 2 - Linux Revenge

We now know SUID is dangerous, and we can read /flag. How many SUID binaries can we abuse so much that we actually get arbitrary execution as root (not just a root read)?

The homework:

- netcat to cse466.pwn.college, port 23 (the password will be posted on the class mailing list)
- enter your hacker alias (you can choose a new one) and ASU ID.
- Choose a single binary to make SUID.
- Use that single binary to execute the "/get_flag" binary (which is only executable by root).
- That binary will print "/flag" for you. You will not be able to read "/flag" even as root!
- Log out ("exit" command).
- Provide the flag when asked.
- Scoring is done automatically.
- **2 points** per unique binary (keyed on basename this time; we're closing the path tricks).