CS202 (003): Operating Systems
Unix Security

Last time

Protections and security in Unix

U(ser)ID and G(roup)ID

Files and directories are access-controlled:
system stores with each file who owns it (in inode)

Root (UID 0)

Has all the permissions: read any file, do anything, …

Some legitimate actions require more privileges than UID

How should users change their passwords (root-owned)?

Each process has a real and effective UID/GID

Real is user who launched the program, effective is owner/group executables, used in access checks

Setuid

a program that is run in with **raised privilege level**

Example attacks

Attacker Setup:

```
close(2);                  // Attacker closes stderr (file descriptor 2)
exec("/usr/bin/passwd") // Then launches the passwd program
```

passwd:

```
main() {
    fd = open("/etc/passwd", ...);  // Opens the password file
    ……
    fprintf(stderr, "Err msg\n");   // Tries to print an error message to stderr
}
```

Example attacks

IFS (Internal File Separator)
a special shell environment variable in Unix and Unix-like systems that defines the characters the shell uses to split words and process command lines

Example attacks

The Starting Point:

- There was a program called "preserve" that was installed with setuid **root** permissions
- This program was used by old text editors like vi to create backup files in root-accessible directories
- When preserve runs, it uses the system() call to execute "/bin/mail" to notify users about backups

The Vulnerability Chain:

1. The attacker first manipulates the IFS (Internal Field Separator) environment variable, setting it to "/"
2. When running vi, which triggers preserve:
    - vi executes preserve with setuid root privileges
    - preserve then calls system("/bin/mail")
    - Due to the modified IFS, the shell parses "/bin/mail" as two separate words: "bin" and "mail"

The Exploit:

- The attacker creates a malicious executable named "bin" in their directory
- When the system() call runs, instead of executing /bin/mail, it finds and executes the attacker's "bin" program
- The malicious "bin" program, now running with root privileges, can:
    1. Reset IFS to normal (spaces, tabs, newlines)
    2. Create a copy of /bin/sh
    3. Change the ownership to root (chown root)
    4. Set the setuid bit (chmod 4755)

http://web.deu.edu.tr/doc/oreily/networking/puis/ch05_05.htm

Example attacks

The Starting Point:

- There was a program called "preserve" that was installed with setuid **root** permissions
- This program was used by old text editors like vi to create backup files in root-accessible directories
- When preserve runs, it uses the system() call to execute "/bin/mail" to notify users about backups

The Vulnerability Chain:

1. The attacker first manipulates the IFS (Internal Field Separator) environment variable, setting it to "/"
2. When running vi, which triggers preserve:
   - vi executes preserve with setuid root privileges
   - preserve then calls system("/bin/mail")
   - Due to the modified IFS, the shell parses "/bin/mail" as two separate words: "bin" and "mail"

The Exploit:

- The attacker creates a malicious executable named "bin" in their directory
- program
-

How can we fix this?

(shell has to ignore IFS if the shell is running as root or if EUID != UID)

3. Change the ownership to root (chown root)
4. Set the setuid bit (chmod 4755)

http://web.deu.edu.tr/doc/oreily/networking/puis/ch05_05.htm

Example attacks

ptrace

Provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers.

It is primarily used to implement breakpoint debugging and system call tracing.

https://man7.org/linux/man-pages/man2/ptrace.2.html

Example attacks

Attack 1 - Direct Privilege Escalation:

- The fundamental issue is an unprivileged process attempting to ptrace a privileged (setuid) program
- This would allow the attacker to manipulate the memory of a root process, effectively gaining root privileges
- The solution implemented was to prevent processes from ptracing more privileged processes or processes owned by other users
- The security check requires the tracing process to have matching real and effective UIDs as the target

Attack 2 - Privilege Escalation via exec():

- More subtle attack where an unprivileged process A traces another unprivileged process B
- Initially this is fine since they have the same privileges
- The vulnerability occurs when B executes a setuid program (like 'su')
- This would result in A having debug control over a now-privileged process
- The fix was to disable the setuid bit when a traced process calls exec()
- An exception is made for root, which can still ptrace any process

Attack 3 - Complex Privilege Escalation Chain:

- This is a sophisticated attack that bypasses the previous two fixes
- Process A traces Process B (both unprivileged)
- A executes "su attacker" (becoming temporarily root during su execution)
- During this window, B executes "su root"
- Because A is temporarily root, B's exec() maintains the setuid bit (bypassing Attack 2's fix)
- The attacker can then manipulate B's memory during the password check
- This results in A being connected to a root shell

Example attacks

TOCTTOU attacks (time-of-check-to-time-of-use)
Exploit the time gap between when a program checks a resource's properties and when it actually uses that resource. This race condition can lead to serious security vulnerabilities.

Example attacks

Problem:

```
fd = open(logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
```

a setuid program that is readable/writable by everyone

Fix #1:

```
if (access(logfile, W_OK) < 0)
    return ERROR;
fd = open(logfile, ...);
```

Does this solve the problem?

Attack Sequence:

The attacker runs the setuid program with "/tmp/X" as the logfile parameter

```
    The program                              Attacker
                                             create("/tmp/X");
  check access("/tmp/X") --> OK
                                             unlink("/tmp/X");
                                            symlink("/etc/passwd", "/tmp/X")
    open("/tmp/X")
```

The program then opens what it thinks is "/tmp/X" but is actually "/etc/passwd"

Issue: check (access()) and use (open()) operations are not atomic

Result: The privileged program inadvertently writes to the password file

Example attacks

Problem:

```
fd = open(logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
```

a setuid program that is readable/writable by everyone

Fix #2:

```
Use file descriptor-based operations that are relative to an already opened directory:
  openat(), renameat(), unlinkat(), symlinkat(), faccessat()
  fchown(), fchownat(), fchmod(), fchmodat(), fstat(), fstatat()
```

```
// CHECK: Does /home/user/file exist?
if (access("/home/user/file", W_OK) < 0)
    return ERROR;

// USE: Open /home/user/file
// BUT what if the path changed
   between check and use?
fd = open("/home/user/file", O_WRONLY);
```

```
// Open the directory first
int dir_fd = open("/home/user", O_DIRECTORY);

// All subsequent operations are relative to this diectory
// Even if attacker changes symlinks/paths,
    we're still operating relative to our original directory
if (faccessat(dir_fd, "file", W_OK, 0) < 0)
    return ERROR;

fd = openat(dir_fd, "file", O_WRONLY);
```

Example attacks

Problem:

```
fd = open(logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
```

a setuid program that is readable/writable by everyone

Fix #3:

Path Traversal Verification:

- Manually traverse the path
- Verify each directory component
- Check for unexpected symbolic links
- Verify path hasn't been modified during operations

Fix #4:

Transactional Approaches:

- Use operating system-level transactions where available

Power up to Terminal

- Step 1: Power up
- Step 2: Firmware
- Step 3: OS bootloader to Kernel
- Step 4: Kernel
- Step 5: init
- Step 6: login(1)

Step 1: Power up

Processor Initialization

- Zero out registers
- Set control registers to default values (Intel defaults in Software Development Manual)
- **Enter Real Mode:**
  - No paging - all addresses are physical
  - Up to 1MB physical memory access

Firmware Loading

- Processor copies executable from ROM to RAM
- Jumps to known offset (historically 0xFFFF0)
- **Modern Firmware:**
  - Stored on EEPROMs/Flash for upgrades
  - Settings stored in battery-backed CMOS

Step 2: Firmware

Firmware

Responsible for hardware initialization and providing a runtime for the kernel during early boot
On recent Intel machines, firmware can be broadly classified as either BIOS or **UEFI** firmwares.
Both are specifications, and many different implementations exist for both

UEFI Initialization Steps

- Switch to Long Mode
  - Enable paging & 64-bit addressing
  - Create identity mapped page table
  - Install IDT for interrupts
  - Initialize processor structures
- Initialize Devices
  - Disks, USB, Display, Input devices
  - Network cards and peripherals
- Mount VFAT partition & load OS bootloader

Key Components

- UEFI Services
  - Network communication
  - File operations
  - Display & input handling
- Device Tree (CONFIGURATION_TABLE)
  - Lists all connected devices
  - Specifies I/O methods & addresses
  - Maps interrupt routing
-

Note: UEFI (Unified Extensible Firmware Interface) handles hardware initialization and provides runtime for early kernel boot

Step 3: OS bootloader to Kernel

The UEFI firmware loads and executes the OS bootloader.

On recent Linux kernels the bootloader is the vmlinuz file with a stub for UEFI, we only consider this case here.

vmlinuz Structure

1
PE Header + EFI stub

2
ELF header + decompression stub

3
Compressed kernel data (bzip2/gzip)

Execution Flow

1. EFI Stub
Loads and executes decompression stub

2. Decompression
Uncompresses kernel data into memory

3. Kernel Launch
Executes kernel with CONFIGURATION_TABLE pointer

4. Firmware Exit
Kernel terminates UEFI firmware

Note: vmlinuz requires root directory device ID as argument (root=<device>)

Step 4: Kernel

Memory Management

Switch from identity-mapped virtual address space

Interrupt Handling

Rewrite interrupt descriptor table

Device Management

- Load and initialize device drivers
- Use CONFIGURATION_TABLE information
- Drivers run as part of kernel
- Communication via /dev files

Root Device

- Mount root device
- Run fsck if required

Final Step

Fork and launch init process

Step 5: init

Executed as root

System Initialization

- Device Configuration
  - Network IP (DHCP)
  - GPU resolution
  - Power management
- Communication via /dev

Daemon Management

- Launch system services (sshd, httpd)
- Handle port binding (1-1024)
- Manage privileges
- Monitor & restart on failures

Session Management

- Launch login manager (login(1))
- Handle user sessions
- Restart on user logout

Note: Most distributions now use systemd, but this represents a simpler init.rc-like implementation

Step 6: login(1)

Must run as root

Authentication

- Prompt for username and password
- Verify against:
  - /etc/passwd
  - /etc/shadow

Login Sequence

1. Fork new process

   Parent waits for child exit
2. Set user permissions
   - setuid(2) for user ID
   - setgid(2) for group ID
3. Change to user's home directory
4. Launch login shell (e.g., bash)

Logout Process

1. User kills shell process → Parent login process exits
2. init detects exit (via wait) → Starts new login process

Remarks

Two operating systems

Firmware: a simple operating system providing a few services. It does not support multiple processes, and has only limited functionality.

Kernel: a richer set of functionality, including schedulers, etc.

Different architectures

Monolithic Kernels : Device drivers are a part of the kernel (like in Linux)

Microkernels: Device drivers and many other portions are run as independent processes

https://cs.nyu.edu/~mwalfish/classes/ut/s10-cs372h/ref/ast-torvalds.html

Final Exam Logistics

- Happens on 12/17 12-1:45pm (105 mins) at 7 East 12th St Room LL23
- Closed book, 1 letter-sized double-sided cheat sheet allowed
- Format similar to the midterm exam
- **Everything we covered in this class might show up in exam**
- Bring your ID, Any electronics NOT allowed

Review session on Thursday!