CSE543 - Introduction to Computer and Network Security

Module: Operating System Security

Professor Trent Jaeger
OS Security

- So, you have built an operating system that enables user-space processes to access hardware resources
  - Thru various abstractions: files, pages, devices, etc.
- Now, you want your operating system to enforce security requirements for your application processes
  - What do you do?
OS Security

• We learned about a few things last time that will hopefully help you

• Your OS must implement a
  ‣ Protection system

• Your protection system is enforced by a
  ‣ Reference validation mechanism

• Which do you tackle first?
Multiprocessor Systems

• Major Effort: *Multics*
  ‣ Multiprocessing system -- developed many OS concepts
    • Including security
  ‣ Begun in 1965
    • Development continued into the mid-70s
  ‣ Used until 2000
  ‣ Initial partners: MIT, Bell Labs, GE/Honeywell
  ‣ Other innovations: hierarchical filesystems, dynamic linking

• Subsequent proprietary system, *SCOMP*, became the basis for secure operating systems design
Multics Goals

• Secrecy
  ‣ Multilevel security

• Integrity
  ‣ Rings of protection

• Resulting system is considered a high point in secure system design
Multics Basics

• **Processes** are programs that are executing within Multics (seems obvious now ...)
  ‣ Protection domain is a list of segments
  ‣ Stored in the process descriptor segment
• **Segments** are stored value regions that are accessible by processes, e.g., memory regions, secondary storage
  ‣ Segments can be organized into hierarchies
  ‣ Local segments (memory addresses) are accessed directly
  ‣ Non-local segments are addressed by hierarchy
    • /tape/drive1/top10k
    • /etc/conf/http.conf
    • This is the genesis of the modern hierarchical filesystem!
Segment Management

- PDS acts like segment working set for process
  - Segments are addressed by name (path)
  - If authorized, added to PDS
  - Multics security is defined with respect to segments
Multics Functionality

- Not all code executing on Multics (or any OS) is the same
  - Kernel
  - System processes
  - Middleware
  - Important User Processes
  - Untrusted User Processes
  - ...

- Previously, all ran in one protection domain
  - Multics supported multiple address spaces (processes)
  - But, how do you protect one process from another?
Protection Rings

• Successively less-privileged “domains”
• Modern CPUs support 4 rings
  ‣ Use 2 mainly: Kernel and user
• Intel x86 rings
  ‣ Ring 0 has kernel
  ‣ Ring 3 has application code

• Example: Multics (64 rings in theory, 8 in practice)
What Are Protection Rings?

- Coarse-grained, Hardware Protection Mechanism
- Boundary between Levels of Authority
  ‣ Most privileged -- ring 0
  ‣ Monotonically less privileged above
- Fundamental Purpose
  ‣ Protect system integrity
    • Protect kernel from services
    • Protect services from apps
    • So on...
Intel Protection Ring Rules

• Each Memory Segment has a privilege level (ring number)

• The CPU has a Current Protection Level (CPL)
  ‣ Level of the segment where instructions are being read

• Program can read/write in segments of higher level than CPL
  ‣ kernel can read/write user space
  ‣ user cannot read/write kernel
  • why not?
Protection Ring Rules

• Program cannot call code of higher privilege directly
  ‣ Gate is a special memory address where lower-privilege code can call higher
  • Enables OS to control where applications call it (system calls)
Multics Interpretation

- Kernel resides in ring 0
- Process runs in a ring \( r \)
  - Access based on current ring
- Process accesses data (segment)
  - Each data segment has an access bracket: \((a_1, a_2)\)
  - \( a_1 \leq a_2 \)
  - Describes read and write access to segment
  - \( r \) is the current ring
  - \( r \leq a_1 \): access permitted
  - \( a_1 < r \leq a_2 \): \( r \) and \( x \) permitted; \( w \) denied
  - \( a_2 < r \): all access denied
Multics Interpretation (con’t)

- Also different procedure segments
  - with call brackets: \((c_1, c_2), c_1 \leq c_2\)
  - and access brackets \((a_1, a_2)\)
  - The following must be true \((a_2 == c_1)\)
  - Rights to execute code in a new procedure segment
    - \(r < a_1\): access permitted with ring-crossing fault
    - \(a_1 \leq r \leq a_2 = c_1\): access permitted and no fault
    - \(a_2 < r \leq c_2\): access permitted through a valid gate
    - \(c_2 < r\): access denied

- What’s it mean?
  - case 1: ring-crossing fault changes procedure’s ring
    - increases from \(r\) to \(a_1\)
  - case 2: keep same ring number
  - case 3: gate checks args, decreases ring number

- Target code segment defines the new ring
Examples

• Process in ring 3 accesses data segment
  ‣ access bracket: (2, 4)
  ‣ What operations can be performed?

• Process in ring 5 accesses same data segment
  ‣ What operations can be performed?

• Process in ring 5 accesses procedure segment
  ‣ access bracket (2, 4)
  ‣ call bracket (4, 6)
  ‣ Can call be made?
  ‣ How do we determine the new ring?
  ‣ Can new procedure segment access the data segment above?
More Multics Segments

- **Named segments** are protected by access control lists and MLS protections
  - Hierarchically arranged
  - Precursor to hierarchical file systems

- **Memory segment** access is controlled by hardware monitor
  - Multics hardware retrieves *segment descriptor word* (SDW)
    - Like a file descriptor
  - Based on rights in the SDW determines whether can access segment

- **Master mode** (like root) can override protections
Multics & Reference Monitor

• Multics Final Report - Schroeder
  ‣ Multics is too large to satisfy verifiability
  ‣ Multics was 44K SLOC!
• What about tamperproofing and complete mediation?
Multics Vulnerability Analysis

• Karger and Schell - Detailed security analysis covering
  ‣ Hardware
  ‣ Software
  ‣ Procedural features (administration)

• Good news
  ‣ Design for security
  ‣ System language prevents buffer overflows
    • Defined buffer sizes
  ‣ Hardware features prevent buffer overflows
    • Addressing off segment is an error
    • Stack grows up
  ‣ System is much smaller than current UNIX system
Vulnerabilities Found

• Hardware
  ‣ Indirect addressing -- incomplete mediation
    • Check direct, but not indirect address
  ‣ Mistaken modification introduced the error

• Software
  ‣ Ring protection (done in software)
    • Argument validation was flawed
    • Certain type of pointer was handled incorrectly
  ‣ Master mode transfer
    • For performance, run master mode program (signaler) in user ring
    • Development assumed trusted input to signaler -- bad combo

• Procedural
  ‣ Trap door insertion goes undetected
Now forward to UNIX ...
Privilege Escalation

• Sounds bad, but...
  ‣ Many programs depend on a mechanism to increase rights

• Example: **Password**
  ‣ Run shell as you, but need to change your password
  ‣ Requires modification to file `/etc/shadow`
  ‣ Only modifiable by root
  ‣ What do you do?
UID Transition: Setuid

• A special bit in the mode bits
• Execute file
  ‣ Resulting process has the effective (and fs) UID/GID of file owner
• Enables a user to *escalate privilege*
  ‣ For executing a trusted service
• **Downside:** User defines execution environment
  ‣ e.g., Environment variables, input arguments, open descriptors, etc.
• Service must protect itself or user can gain root access

• **All UNIX services involve root processes – many via setuid**
UNIX Security Limitations

• **Circa 2000 Problems**
  ▶ Discretionary access control
  ▶ Setuid root processes
  ▶ Network-facing daemons vulnerable
  ▶ Name resolution vulnerabilities (we still have those)

• What can we do?
UNIX Security Limitations

• *Circa 2000 Problems*
  ‣ Discretionary access control
  ‣ Setuid root processes
  ‣ Network-facing daemons vulnerable
  ‣ Name resolution vulnerabilities (we still have those)

• What can we do?
  ‣ *Reference validation mechanism* that satisfies *reference monitor concept*

  ‣ *Protection system with mandatory access control* (mandatory protection system)
Linux Security Modules

• Reference validation mechanism for Linux
  ‣ Upstreamed in Linux 2.6
  ‣ Support modular enforcement - you choose
    • SELinux, AppArmor, POSIX Capabilities, SMACK, ...

• 150+ authorization hooks
  ‣ Mediate security-sensitive operations on
    • Files, dirs/links, IPC, network, semaphores, shared memory, ...
  ‣ Variety of operations per data type
    • Control access to read of file data and file metadata separately

• Hooks are restrictive
Linux Security Modules

```c
linux/fs/read_write.c:

ssize_t vfs_read(...) {
    ...
    ret = security_file_permission(file, ...);
    if (!ret) {
        ret = file->f_op->read(file, ...); ...
    }
    ...
}
```

Security check function

Security sensitive operation
• Now LSMs are always compiled into the kernel
LSM & Reference Monitor

- Does LSM satisfy reference monitor concept?
LSM & Reference Monitor

• Does LSM satisfy reference monitor concept?
  ‣ Tamperproof
    • Can MAC policy be tampered?
    • Can kernel be tampered? By network threats?
LSM & Reference Monitor

• Does LSM satisfy reference monitor concept?
  ‣ Tamperproof
    • Can MAC policy be tampered?
    • Can kernel be tampered? By network threats?
  ‣ Verifiable
    • How large is kernel?
    • Can we perform complete testing?
LSM & Reference Monitor

- Does LSM satisfy reference monitor concept?
  - Tamperproof
    - Can MAC policy be tampered?
    - Can kernel be tampered? By network threats?
  - Verifiable
    - How large is kernel?
    - Can we perform complete testing?
  - Complete Mediation
    - What is a security-sensitive operation?
    - Do we mediate all paths to such operations?
LSM & Complete Mediation

• Security-sensitive operations
  ‣ Instructions? Which?
  ‣ Structure member accesses? To what data?
  ‣ Data types whose instances may be controlled
    • Inodes, files, IPCs, tasks, ...

• Approaches
  ‣ Mediation: Check that authorization hook dominates all control-flow paths to structure member access on security-sensitive data type
  ‣ Consistency: Check that every structure member access that is mediated once is always mediated
    • Several bugs found - some years later
LSM & Complete Mediation

- Static analysis of Zhang, Edwards, and Jaeger [USENIX Security 2002]
  - Based on a tool called CQUAL
- Found a TOCTTOU bug
  - Authorize filp in `sys_fcntl`
  - But pass fd again to `fcntl_getlk`
- Many supplementary analyses were necessary to support CQUAL

```c
/* from fs/fcntl.c */
long sys_fcntl(unsigned int fd,  
               unsigned int cmd,  
               unsigned long arg)
{
    struct file * filp;
    ...
    filp = fget(fd);
    ...
    err = security_ops->file_ops
           ->fcntl(filp, cmd, arg);
    ...
    err = do_fcntl(fd, cmd, arg, filp);
    ...
}
static long
do_fcntl(unsigned int fd,  
          unsigned int cmd,  
          unsigned long arg,  
          struct file * filp) {
    ...
    switch(cmd){
    ...
        case F_SETLK:
            err = fcntl_setlk(fd, ...);
    ...
    ...
}
/* from fs/locks.c */
fcntl_getlk(fd, ...) {  
    struct file * filp;
    ...
    filp = fget(fd);
    /* operate on filp */
    ...
}
```

Figure 8: Code path from Linux 2.4.9 containing an exploitable type error.
Take Away

• **Goal**: Build authorization into operating systems
  ‣ Multics and Linux

• **Requirements**: Reference monitor
  ‣ Satisfy reference monitor concept

• Multics
  ‣ Mediation on segments
  ‣ Still some bugs; not convinced could verify

• Linux
  ‣ Did not enforce security *(DAC, Setuid, root daemons)*
  ‣ Linux Security Modules
  ‣ Approximates reference monitor assuming network threats only -- some challenges in ensuring complete mediation