Advanced Systems Security: Ordinary Operating Systems

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
UNIX and Windows

• If you want to run an application, you have to install one of these
  ‣ Where UNIX encompasses Linux and Mac OS X

• Common understanding
  ‣ They are insecure

• Why?
UNIX Access Control

• On Files
  ‣ All objects are files
  ‣ Not exactly true

• Classical Protection System
  ‣ Limited access matrix
  ‣ Discretionary protection state operations

• Practical model for end users
  ‣ Still involves some policy specification
## UNIX Mode Bits

<table>
<thead>
<tr>
<th>Mode</th>
<th>User</th>
<th>Group</th>
<th>Size</th>
<th>Date/Time</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>-rw-rw-r--</td>
<td>1 pbg</td>
<td>staff</td>
<td>31200</td>
<td>Sep 3 08:30</td>
<td>intro.ps</td>
</tr>
<tr>
<td>drwx-------</td>
<td>5 pbg</td>
<td>staff</td>
<td>512</td>
<td>Jul 8 09:33</td>
<td>private/</td>
</tr>
<tr>
<td>drwxrwxr-x</td>
<td>2 pbg</td>
<td>staff</td>
<td>512</td>
<td>Jul 8 09:35</td>
<td>doc/</td>
</tr>
<tr>
<td>drwxrwx---</td>
<td>2 pbg</td>
<td>student</td>
<td>512</td>
<td>Aug 3 14:13</td>
<td>student-proj/</td>
</tr>
<tr>
<td>-rw-r--r--</td>
<td>1 pbg</td>
<td>staff</td>
<td>9423</td>
<td>Feb 24 2003</td>
<td>program.c</td>
</tr>
<tr>
<td>-rwrx-r-x</td>
<td>1 pbg</td>
<td>staff</td>
<td>20471</td>
<td>Feb 24 2003</td>
<td>program</td>
</tr>
<tr>
<td>drwx--x--x</td>
<td>4 pbg</td>
<td>faculty</td>
<td>512</td>
<td>Jul 31 10:31</td>
<td>lib/</td>
</tr>
<tr>
<td>drwx-------</td>
<td>3 pbg</td>
<td>staff</td>
<td>1024</td>
<td>Aug 29 06:52</td>
<td>mail/</td>
</tr>
<tr>
<td>drwxrwxrwx</td>
<td>3 pbg</td>
<td>staff</td>
<td>512</td>
<td>Jul 8 09:35</td>
<td>test/</td>
</tr>
</tbody>
</table>
Windows Access Control

- On Objects
  - Arbitrary classes can be defined
  - New classes can be defined (Active Directory)

- Classical Protection System
  - Full-blown ACLs (even negative ACLs)
  - Discretionary protection state operations

- Not so usable
  - Few people have experience
Windows Access Control

![Diagram of Windows Access Control](image)

- **Access Token (user 1)**
  - user 1 SID
  - Group SIDs
  - Privilege information
  - Other access information
  - Jo
    - Group 1
    - Group 2

- **Access Token (user 2)**
  - Carl
    - Group 2
    - Group 3

- **Security Descriptor**
  - Owner SID
  - Group SID
  - SACL

- **DACL**
  - **ACE1** – Denied
    - Jo
    - Read, Write, Execute
  - **ACE2** – Allowed
    - Group 1
    - Read, Write, Execute
  - **ACE3** – Allowed
    - Group 3
    - Read, Write, Execute

- **Access is denied**
- **Access is allowed**
Vulnerabilities

• Unexpected function run by victim (flaw) that enables an attacker (with access) to gain unauthorized privileges (i.e., the victim’s)
  ‣ Privilege escalation

• Two views
  ‣ Victim could do something for the attacker
  ‣ Attacker could “take over” process

• Either way, the attacker is trying to use the victim’s privileges
Take Over Process

• Gain control of execution

• Return-oriented programming
  ‣ Generalization of buffer overflows
  ‣ http://cseweb.ucsd.edu/~hovav/dist/rop.pdf
Return-oriented Programming

- Gain control stack pointer
  - How?

- Can execute an exploit with existing code
  - Turing-complete environment with enough existing code
Return-oriented Programming

• Gadgets

Immediate constants

- Instructions can encode constants
- Return-oriented equivalent:
  - Store on the stack;
  - Pop into register to use

mov $0xdeadbeef, %eax
(bb ef be ad de)

pop %ebx; ret

0xdeadbeef
Confused Deputy

• A server process has the privileges necessary to service requests from all its clients
  ‣ Can a client trick the server into using its permissions for the client?

• Have the server access an object whose name is supplied by a client
Name Resolution Attacks

• A server may use a file under the control of a client
  ‣ Client may provide file by name of file not normally accessible to adversary

• A server may be redirected to a file of client’s choice
  ‣ Client may provide bindings used to resolve name to redirect to file not normally accessible to adversary
Name Resolution Attacks

- Multiple names for a single inode
- Run
  - ln -s /etc/passwd badlink
  - setuid_program badlink < *passwd-entry*
  - To overwrite /etc/passwd
- Programs have to be aware of which files they are using
- open(file, O_NOFOLLOW, mode)
  - Prevents open from following a link
The OS Will Protect Me

• User-space vulnerabilities are expected
  ‣ Those processes are untrusted

• OS policies will protect the system from harm
  ‣ Only the OS and a few processes need to be trusted

• Just need to specify the access control policy
  ‣ We can specify anything we want
  ‣ We have an access matrix
The OS is Limited

- So why are these UNIX/Windows fault?
  - Isn’t it just the programmer’s fault?

- What aspect of vulnerability does the OS control?
Secrecy

• Does the following protection state ensure the secrecy of J’s private key in $O_1$?

<table>
<thead>
<tr>
<th></th>
<th>$O_1$</th>
<th>$O_2$</th>
<th>$O_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>S_2</td>
<td>N</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>S_3</td>
<td>N</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
Integrity

- Does the following access matrix protect the integrity of J’s public key file $O_2$?

<table>
<thead>
<tr>
<th></th>
<th>$O_1$</th>
<th>$O_2$</th>
<th>$O_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>$S_2$</td>
<td>N</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>$S_3$</td>
<td>N</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>W</td>
<td>W</td>
</tr>
</tbody>
</table>
Benign vs. Byzantine

- Does it matter if we do not trust some of J’s processes?

<table>
<thead>
<tr>
<th></th>
<th>$O_1$</th>
<th>$O_2$</th>
<th>$O_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J$</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>$S_2$</td>
<td>N</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>$S_3$</td>
<td>N</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
Protection and Security

• Protection
  ‣ Security goals met under **benign** processes
  ‣ Protects against error by non-malicious entity

• Security
  ‣ Security goals met under **Byzantine** processes
  ‣ Any benign process can come under the control of an attacker
  ‣ Protects against any malicious entity

• For J: A benign process won’t accidentally leak a key, but it would under attackers’ control
Is Fixing the Policy Enough?

- No, as these systems do not satisfy the reference monitor concept
Complete Mediation

- **Mediation**: Does interface mediate correctly?
  - No. Several operations impact security that are ignored.

- **Mediation**: On all resources?
  - UNIX: No. No network.
  - Windows: Could.

- **Mediation**: Verifably?
  - Ha.
**Tamperproof**

- **Tamperproof**: Is reference monitor protected?
  - Operating system is not protected (see Rootkits)
  - Kernel modules, trusted processes, extensible function
  - Policy can be modified by untrusted processes (discretionary)

- **Tamperproof**: Is system TCB protected?
  - We don’t really know what this is
  - All root and setuid processes and ones they depend on
  - Plus anything an admin runs as root
Verification

- **Verifiable**: Is TCB code base correct?
  - No.

- **Verifiable**: Does the protection system enforce the system’s security goals?
  - Goals?
  - See Protection v. Security again
Take Away

- Conventional operating systems are insecure
- They run programs that suffer from many types of vulnerabilities
- They are designed to enable protection under benign programs, not secure a system from a directed attacker
- They do not satisfy reference monitor concept
  - Also fail to implement a mandatory protection system
Ordinary System Defenses

• Haven’t people tried to prevent such attacks?
  ‣ Yes, but it isn’t easy to block entirely

• Prevent gaining control of code
  ‣ Prevent modifying instruction pointers
  ‣ Prevent code injection

• Prevent confused deputy (name resolution attacks)
  ‣ Prevent unauthorized use of links
  ‣ Prevent file squatting
  ‣ Prevent confused deputy
StackGuard

- Before and After

Figure 1: "Stack Smash" Attack Against Activation Record
StackGuard

• Does it prevent all modifications that can update memory that holds executable addresses?
  ‣ What about function pointers?
StackGuard for Heap Memory

- Can it be done?
  - How?

- Why hasn’t that been adopted?
  - Note that it took over ten years for StackGuard to be adopted
Prevent Code Injection

- Inject code onto stack
Prevent Code Injection

- Linux PaX Patch
  - By Solar Designer
Non-Executable Pages

• $W \oplus X$
  ‣ Page can either be written or executed, but not both

• What can be written?

• What can be executed?
• W XOR X
  ‣ Doesn’t prevent all attacks
• Return-oriented programming

any sufficiently large program codebase

arbitrary attacker computation and behavior, without code injection

(in the absence of control-flow integrity)
ASLR

• Address Space Layout Randomization
  ‣ Randomly locate of key “segments” of memory

• Data areas: Stack
  ‣ Why does this make code injection difficult?

• Data areas: Heap
  ‣ Does this work as well?

• Code areas: Libraries
  ‣ What does this prevent?
Limitations of Such Defenses

- All memory that contains a code address must be protected
  - StackGuard only protects return address
- Can execute existing code (no need to inject)
  - Return-oriented programming
- Not full ASLR
  - Not all code segments are randomized
  - Code must be position-independent
  - May be broken if entropy is too low
Defenses for Name Resolution

- System call APIs have extended for programmers to prevent such attacks

- Open – turns a name into a file descriptor
  - Flags for limiting resources access
  - O_NOFOLLOW – “If pathname is a symbolic link, then the open fails”
  - O_EXCL – “if this flag is specified in conjunction with O_CREAT, and pathname already exists, then open() will fail.”

- Programmer flags to constrain resources retrieved
Control Use of Links

- **O_NOFOLLOW** – “If *pathname* is a symbolic link, then the open fails”

- Programmer can use if they wish to prevent retrieval of a resource using a symbolic link
  - If adversary creates a link from `/usr/home/client/data` to `/etc/passwd`
  - Programmer’s use of O_NOFOLLOW prevents an open of `/usr/home/client/data` to follow a link

- Sometimes want to follow an untrusted link (to an untrusted file)
Prevent File Squats

- **O_EXCL** – “if this flag is specified in conjunction with **O_CREAT** and `pathname` already exists, then `open()` will fail.”

- Programmer can use if they wish to check whether a file being created already exists
  - If adversary creates a file `/tmp/XI1_l_temp/file` a file used by the victim in advance
  - Programmer’s use of **O_EXCL** with **O_CREAT** prevents an open of `/tmp/XI1_l_temp/file` if it already exists

- However, it may be OK for some files to exist already (if no adversary could have tampered them)
Check File Metadata

- Other system calls can check properties of file
  - `stat()` stats the file pointed to by `path` and fills in `buf`.

```c
struct stat {
    dev_t    st_dev;    /* ID of device containing file */
    ino_t    st_ino;    /* inode number */
    mode_t   st_mode;   /* protection */
    nlink_t  st_nlink;  /* number of hard links */
    uid_t    st_uid;    /* user ID of owner */
    gid_t    st_gid;    /* group ID of owner */
    dev_t    st_rdev;   /* device ID (if special file) */
    off_t    st_size;   /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t st_blocks; /* number of 512B blocks allocated */
    time_t   st_atime;  /* time of last access */
    time_t   st_mtime;  /* time of last modification */
    time_t   st_ctime;  /* time of last status change */
};
```
Check File Metadata

• `lstat()` is identical to `stat()`, except that if `path` is a symbolic link, then the link itself is stat-ed, not the file that it refers to.
  ‣ Enables programmer to see that a pathname is really a link

• `fstat()` is identical to `stat()`, except that the file to be stat-ed is specified by the file descriptor `fd`.
  ‣ Enables programmer to check the metadata of the file that was really retrieved
  ‣ Inode number, permissions, …
Check File Metadata

- How to use these to prevent link traversal attacks
  - Particularly TOCTTOU attacks

- Could check if a link before opening
  - Which system call?

- Then do open of file
  - How do we know it is the same file we checked?

- Could check that the file descriptor is file expected
  - Which system call?
Problems with Using These

• Can we find prevent TOCTTOU?
  ‣ And adversary can accurately insert changes
  ‣ LAOF – sLaAsOF
  ‣ s – secret file (target)
  ‣ a – accessible file (one that is legal for adversary)
  ‣ Make adversary win k races of this type

• Expensive
  ‣ Running these system calls is much more expensive than open alone
Safe-Open

- Prevent link traversal and races
  - LOF in programs
  - For every path element
  - Safe-open paper has system version

- Expensive
  - Over 70% overhead vs open alone
  - Over 100% overhead if path length $\geq 7$

- Do people use safe-open?
  - Apache docs inform admins that this defense can be disabled to improve performance
Confine Processes

- OK, so some processes may be compromised
  - Hopefully, they are unprivileged
  - Can we confine them?
- What is the confinement mechanism in UNIX?
Confine Processes

- OK, so some unprivileged processes may be compromised
- Can we confine them to limit possible damage?
- How do we do that in UNIX?

Chroot

- Create a *domain* in which a process is *confined*
  - Process can only read/write within file system subtree
  - Applies to all descendant processes
  - Can carry file descriptors in ‘chroot jail’
Chroot Vulnerability

• Unfortunately, chroot can trick its own system
  – define a passwd file at `<newroot>/etc/passwd`
  – run `su`
    • `su` thinks that this is the real passwd file
  – gives root access
    • Use `mknod` to create device file to access physical memory

• Setup requires great care
  – Never run chroot process as root
  – Must not be able to get root privileges
  – No control by chrooted process (user) of contents in jail
  – Be careful about descriptors, open sockets, IPC that may be available
Is Chroot an MPS?

- Protection State
  - Confine to name space
  - But can change within the namespace
- Labeling State
  - Could create root objects perhaps
- Transition State
  - No rules governing transitions
- So, no…
Take Away

- Conventional operating systems are insecure
- Many defenses have been proposed, but each provide only partial protection
- Adversary gains control of program …
  - By guiding with data (name resolution)
  - By gaining control of execution (return-oriented programming)
- A variety of defenses have been proposed
  - Attack specific
  - Confinement
- But, they don’t satisfy MPS or ref monitor?