CS 290
Host-based Security and Malware

Christopher Kruegel

chris@cs.ucsb.edu

Unix

Unix / Linux

- Started in 1969 at AT&T / Bell Labs
- Split into a number of popular branches
  - BSD, Solaris, HP-UX, AIX
- Inspired a number of Unix-like systems
  - Linux, Minix
- Standardization attempts
  - POSIX, Single Unix Specification (SUS), Filesystem Hierarchy Standard (FHS), Linux Standard Base (LSB), ELF

Unix

- Kernel vulnerability
  - usually leads to complete system compromise
  - attacks performed via system calls
- Solaris / NetBSD call gate creation input validation problem
  - malicious input when creating a LDT (x86 local descriptor table)
  - used in 2001 by Last Stage of Delirium to win Argus Pitbull Competition
- Kernel Integer Overflows
  - FreeBSD `procfs` code (September 2003)
  - Linux `brk()` used to compromise debian.org (December 2003)
  - Linux `setsockopt()` (May 2004)
- Linux Memory Management
  - `mremap()` and `munmap()` (March 2004)

Unix

- More recent Linux vulnerabilities
  - Linux message interface (August 2005, CAN-2005-2490)
  - race condition - `proc` and `prctl` (July 2006, CVE-2006-3626)
  - local privilege escalation - (September 2007, CVE 2007-4573)
  - security bypass and DoS (May 2008, CVE-2008-2148, CVE-2008-2137)

- Device driver code is particularly vulnerable
  - (most) drivers run in kernel mode, either kernel modules or compiled-in
  - often not well audited
  - very large code based compared to core services

- Examples
  - `aironet`, `asus_acpi`, `decnet`, `mpu401`, `msnd`, and `pss` (2004) found by sparse (tool developed by Linus Torvalds)
  - remote root (MadWifi - 2006, Broadcom - 2006)

Unix

- Code running in user mode is always linked to a certain identity
  - security checks and access control decisions are based on user identity
- Unix is user-centric
  - no roles
- User
  - identified by user name (UID), group name (GID)
  - typically authenticated by password (stored encrypted)
- User `root`
  - superuser, system administrator
  - special privileges (access resources, modify OS)
  - cannot decrypt user passwords

Process Management

- Process
  - implements user-activity
  - entity that executes a given piece of code
  - has its own execution stack, memory pages, and file descriptors table
  - separated from other processes using the virtual memory abstraction
- Thread
  - separate stack and program counter
  - share memory pages and file descriptor table

Process Management

- Process Attributes
  - process ID (PID)
    - uniquely identified process
  - (real) user ID (UID)
    - ID of owner of process
  - effective user ID (EUID)
    - ID used for permission checks (e.g., to access resources)
  - saved user ID (SUID)
    - to temporarily drop and restore privileges
  - lots of management information
    - scheduling
    - memory management, resource management

Process Management

- Switching between IDs
  - uid-setting system calls

    int setuid(uid_t uid)

    int seteuid(uid_t uid)

    int setresuid(uid_t ruid, uid_t euid, uid_t suid)

- Can be tricky
  - POSIX 1003.1:

    If the process has appropriate privileges, the setuid(newuid) function sets the real user ID, effective user ID, and the [saved user ID] to newuid.
  - what are appropriate privileges?

    Solaris: EUID = 0; FreeBSD: newuid = EUID;

    Linux: SETUID capability

Process Management

Bug in sendmail 8.10.1:

- call to setuid(getuid()) to clear privileges (effective UID is root)
- on Linux, attacker could clear SETUID capability
- call clears EUID, but SUID remains root

Further reading

Setuid Demystified
Hao Chen, David Wagner, and Drew Dean
11th USENIX Security Symposium, 2002

User Authentication

- How does a process get a user ID?
- ➢ Authentication

- Passwords
  - user passwords are used as keys for `crypt()` function
  - runs DES algorithm 25 times on a block of zeros
  - 12-bit "salt"
    - 4096 variations
    - chosen from date, not secret
    - prevent same passwords to map onto same string
    - make dictionary attacks more difficult

- Password cracking
  - dictionary attacks, rainbow tables
  - `Crack, JohnTheRipper`

User Authentication

- Shadow passwords
  - password file is needed by many applications to map user ID to user names
  - encrypted passwords are not

- `/etc/shadow`
  - holds encrypted passwords
  - account information
    - last change date
    - expiration (warning, disabled)
    - minimum change frequency
  - readable only by superuser and privileged programs
  - MD5 hashed passwords (default) to slow down guessing

User Authentication

- Shadow passwords
  - a number of other encryption / hashing algorithms were proposed
  - blowfish, SHA-1, …
- Other authentication means possible
  - Linux PAM (pluggable authentication modules)
  - Kerberos
  - Active directory (Windows)

Group Model

- Users belong to one or more groups
  - primary group (stored in `/etc/password`)
  - additional groups (stored in `/etc/group`)
  - possibility to set group password
  - and become group member with `newgrp`

- `/etc/group`

```
groupname : password : group id : additional users
root:x:0:root
bin:x:1:root,bin,daemon
users:x:100:chris
```

- Special group `wheel`
  - protect `root` account by limiting user accounts that can perform `su`

File System

- File tree
  - primary repository of information
  - hierarchical set of directories
  - directories contain file system objects (FSO)
  - root is denoted "/"
- File system object
  - files, directories, symbolic links, sockets, device files
  - referenced by *inode* (index node)

File System

- Access Control
  - permission bits
  - `chmod, chown, chgrp, umask`
  - file listing:

```
          -       rwx     rwx     rwx
(file type) (user)  (group) (other)
```

Type	r	w	x	s	t
File	read access	write access	execute	suid / sgid inherit id	sticky bit
Directory	list files	insert and remove files	stat / execute files, chdir	new files have dir-gid	files only delete-able by owner

SUID Programs

- Each process has *real* and *effective* user / group ID
  - usually identical
  - real IDs
    - determined by current user
    - `authentication (login, su)`
  - effective IDs
    - determine the "rights" of a process
    - system calls (e.g., `setuid()`)
  - `suid` / `sgid` bits
    - to start process with effective ID different from real ID
    - attractive target for attacker
- Never use SUID shell scripts (multiplying problems)

File System

- Shared resource
  - susceptible to race condition problems
- Time-of-Check, Time-of-Use (TOCTOU)
  - common race condition problem
  - problem:

    Time-**O**f-**C**heck (t_1): validity of assumption *A* on entity *E* is checked

    Time-**O**f-**U**se (t_2): assuming *A* is still valid, E is used

    Time-Of-Attack (t_3): assumption *A* is invalidated

    $t_1 < t_3 < t_2$

TOCTOU

- Steps to access a resource
  1. obtain reference to resource
  2. query resource to obtain characteristics
  3. analyze query results
  4. if resource is fit, access it

- Often occurs in Unix file system accesses
  – check permissions for a certain file name (e.g., using **`access(2)`**)
  – open the file, using the file name (e.g., using **`fopen(3)`**)
  – four levels of indirection (symbolic link - hard link - inode - file descriptor)

- Windows uses file handles and includes checks in API open call

Overview

- Case study

```
/* access returns 0 on success */
if(!access(file, W_OK)) {
    f = fopen(file, "wb+");
    write_to_file(f);
} else {
    fprintf(stderr, "Permission denied when trying to open %s.\n", file);
}
```

- Attack

```
$ touch dummy; ln –s dummy pointer
$ rm pointer; ln –s /etc/passwd pointer
```

Examples

- TOCTOU Examples
  - Filename Redirection
  - Setuid Scripts
    1. exec() system call invokes seteuid() call prior to executing program
    2. program is a script, so command interpreter is loaded first
    3. program interpreted (with root privileges) is invoked on script name
    4. attacker can replace script content between step 2 and 3

Examples

- TOCTOU Examples
  - Directory operations
    - **rm** can remove directory trees, traverses directories depth-first
    - issues **chdir("..")** to go one level up after removing a directory branch
    - by relocating subdirectory to another directory,

      arbitrary files can be deleted

  - Temporary files
    - commonly opened in **/tmp** or **/var/tmp**
    - often guessable file names

Temporary Files

"Secure" procedure for creating temporary files

1. pick a prefix for your filename
2. generate at least 64 bits of high-quality randomness
3. base64 encode the random bits
4. concatenate the prefix with the encoded random data
5. set umask appropriately (0066 is usually good)
6. use **fopen(3)** to create the file, opening it in the proper mode
7. delete the file immediately using **unlink(2)**
8. perform reads, writes, and seeks on the file as necessary
9. finally, close the file

Temporary Files

- Library functions to create temporary files can be insecure
  - **mktemp(3)** is not secure, use **mkstemp(3)** instead
  - old versions of **mkstemp(3)** did not set umask correctly

- Temp Cleaners
  - programs that clean "old" temporary files from **temp** directories
  - first **lstat(2)** file, then use **unlink(2)** to remove files
  - vulnerable to race condition when attacker replaces file between **lstat(2)** and **unlink(2)**
  - arbitrary files can be removed
  - delay program long enough until temp cleaner removes active file

Prevention

- “Handbook of Information Security Management” suggests
  1. increase number of checks
  2. move checks closer to point of use
  3. immutable bindings

- Only number 3 is acceptable!

- Immutable bindings
  - operate on file descriptors
  - do not check access by yourself (i.e., no use of **access(2)**)
    drop privileges instead and let the file system do the job

- Use the **O_CREAT | O_EXCL** flags to create a new file with **open(2)** and be prepared to have the open call fail

Prevention

Series of papers on the access system call

Fixing races for fun and profit: how to use access(2)
D. Dean and A. Hu
Usenix Security Symposium, 2004

Fixing races for fun and profit: howto abuse atime
N. Borisov, R. Johnson, N. Sastry, and D. Wagner
Usenix Security Symposium, 2005

Portably Solving File TOCTTOU Races with Hardness Amplification
D. Tsafrir, T. Hertz, D. Wagner, and D.Da Silva
Usenix Conference on File and Storage Technologies (FAST), 2008

Prevention

<u>Series of papers on the access system call</u>

Fixing races for fun and profit: howto use access(2)

K-race [do multiple access and open calls, and check that open always opens the same file]

Fixing races for fun and profit: howto abuse atime

File system maze [make very long directory paths, ensuring that it takes long time between each open and access call]

Portably Solving File TOCTTOU Races with Hardness Amplification

Fix to k-race [check each part of the path with a K-race *]*

Locking

- Ensures exclusive access to a certain resource
- Used to circumvent accidental race conditions
  - advisory locking (processes need to cooperate)
  - not mandatory, therefore not secure
- Often, files are used for locking
  - portable (files can be created nearly everywhere)
  - “stuck” locks can be easily removed
- Simple method
  - open file using the `O_EXCL` flag

Shell

- Shell
  - one of the core Unix application
  - both a command language and programming language
  - provides an interface to the Unix operating system
  - rich features such as control-flow primitives, parameter passing, variables, and string substitution
  - communication between shell and spawned programs via redirection and pipes
  - different flavors
    - bash and sh, tcsh and csh, ksh

Shell Attacks

- Environment Variables
  - $HOME and $PATH can modify behavior of programs that operate with relative path names
  - $IFS – internal field separator
    - used to parse tokens
    - usually set to [\t\n] but can be changed to "/"
    - "`/bin/ls`" is parsed as "`bin ls`" calling bin locally
    - IFS now only used to split expanded variables
  - `preserve` attack (`/usr/lib/preserve` is SUID)
    - called "`/bin/mail`" when `vi` crashes to preserve file
    - change IFS, create `bin` as link to `/bin/sh`, kill `vi`

Shell Attacks

- Control and escape characters
  - can be injected into command string
  - modify or extend shell behavior
  - user input used for shell commands has to be rigorously sanitized
  - easy to make mistakes
  - classic examples are `` `;' `` and `` `&' ``

- Applications that are invoked via shell can be targets as well
  - increased vulnerability surface

- Restricted shell
  - invoked with `-r`
  - more controlled environment

Shell Attacks

- `system(char *cmd)`
  - function called by programs to execute other commands
  - invokes shell
  - executes string argument by calling `/bin/sh -c string`
  - makes binary program vulnerable to shell attacks
  - especially when user input is utilized

- `popen(char *cmd, char *type)`
  - forks a process, opens a pipe and invokes shell for `cmd`

File Descriptor Attacks

- SUID program opens file
- forks external process
  - sometimes under user control
- on-execute flag
  - if `close-on-exec` flag is not set, then new process inherits file descriptor
  - `launch` program works exactly like this
  - malicious attacker might exploit such weakness
- Linux Perl 5.6.0
  - getpwuid() leaves /etc/shadow opened (June 2002)
  - problem for Apache with mod_perl

Resource Limits

- File system limits
  - *quotas*
  - restrict number of storage blocks and number of inodes
  - hard limit
    - can never be exceeded (operation fails)
  - soft limit
    - can be exceeded temporarily
  - can be defined per mount-point
  - defend against resource exhaustion (denial of service)

- Process resource limits
  - number of child processes, open file descriptors

Signals

- Signal
  - simple form of interrupt
  - asynchronous notification
  - can happen anywhere for process in user space
  - used to deliver segmentation faults, reload commands, …
  - `kill` command

- Signal handling
  - process can install signal handlers
  - when no handler is present, default behavior is used
    - ignore or kill process
  - possible to catch all signals except SIGKILL (-9)

Signals

- Security issues
  - code has to be be re-entrant
    - atomic modifications
    - no global data structures
  - race conditions
  - unsafe library calls, system calls
  - examples
    - wu-ftpd 2001, sendmail 2001 + 2006, stunnel 2003, ssh 2006
- Secure signals
  - write handler as simple as possible
  - block signals in handler

Shared Libraries

- Library
  - collection of object files
  - included into (linked) program as needed
  - code reuse

- Shared library
  - multiple processes share a single library copy
  - save disk space (program size is reduced)
  - save memory space (only a single copy in memory)
  - used by virtually all Unix applications (at least libc.so)
  - check binaries with `ldd`

Shared Libraries

- Static shared library
  - address binding at link-time
  - not very flexible when library changes
  - code is fast
- Dynamic shared library
  - address binding at load-time
  - uses procedure linkage table (PLT) and global offset table (GOT)
  - code is slower (indirection)
  - loading is slow (binding has to be done at run-time)
  - classic `.so` or `.dll` libraries
- PLT and GOT entries are very popular attack targets
  - more when discussing buffer overflows

Shared Libraries

- Management
  - stored in special directories (listed in `/etc/ld.so.conf`)
  - manage cache with `ldconfig`

- Preload
  - override (substitute) with other version
  - use `/etc/ld.so.preload`
  - can also use environment variables for override
  - possible security hazard
  - now disabled for SUID programs (old Solaris vulnerability)

Advanced Security Features

- Address space protection
  - address space layout randomization (ASLR)
  - non-executable stack (based on NX bit or PAX patches)

- Mandatory access control extensions
  - SELinux
  - role-based access control extensions
  - capability support

- Miscellaneous improvements
  - hardened chroot jails
  - better auditing