CSci 4271W
Development of Secure Software Systems
Day 10: Unix Access Control

Stephen McCamant
University of Minnesota, Computer Science & Engineering

Outline

Random canary

- Can't reproduce because attacker can't guess
- For efficiency, usually one per execution
- Ineffective if disclosed

XOR canary

- Want to protect against non-sequential overwrites
- XOR return address with value `c` at entry
- XOR again with `c` before return
- Standard choice for `c`: see random canary

Further refinements

- More flexible to do earlier in compiler
- Rearrange buffers after other variables
  - Reduce chance of non-control overwrite
- Skip canaries for functions with only small variables
  - Who has an overflow bug in an 8-byte array?

What's usually not protected?

- Backwards overflows
- Function pointers
- Adjacent structure fields
- Adjacent static data objects

Where to keep canary value

- Fast to access
- Buggy code/attacker can't read or write
- Linux/x86: `%gs:0x14`

Complex anti-canary attack

- Canary not updated on `fork` in server
- Attacker controls number of bytes overwritten

Complex anti-canary attack

- Canary not updated on `fork` in server
- Attacker controls number of bytes overwritten
- ANRY BNRY CNRY DNRY ENRY FNRY
- search 2^{32} $\rightarrow$ search $4 \cdot 2^8$

Shadow return stack

- Suppose you have a safe place to store the canary
- Why not just store the return address there?
- Needs to be a separate stack
- Ultimate return address protection

Outline

Basic idea

- "Address Space Layout Randomization"
- Move memory areas around randomly so attackers can't predict addresses
- Keep internal structure unchanged
  - E.g., whole stack moves together

Code and data locations

- Execution of code depends on memory location
- E.g., on x86-64:
  - Direct jumps are relative
  - Function pointers are absolute
  - Data can be relative (`%rip`-based addressing)

Relocation (Windows)

- Extension of technique already used in compilation
- Keep table of absolute addresses, instructions on how to update
- Disadvantage: code modifications take time on load, prevent sharing

PIC/PIE (GNU/Linux)

- "Position-Independent Code / Executable"
- Keep code unchanged, use register to point to data area
- Disadvantage: code complexity, register pressure hurt performance (especially 32-bit)

What's not covered

- Main executable (Linux PIC)
- Incompatible DLLs (Windows)
- Relative locations within a module/area

Entropy limitations

- Intuitively, *entropy* measures amount of randomness, in bits
- Random 32-bit int: 32 bits of entropy
- ASLR page aligned, so at most $32 - 12 = 20$ bits of entropy
- Other constraints further reduce possibilities

Leakage limitations

- If an attacker learns the randomized base address, can reconstruct other locations
- Any stack address $\rightarrow$ stack unprotected, etc.

Outline

Configurability

- Basic idea: let one mechanism (implementation) support a variety of security policies
- I.e., make security a system configuration
- Classic example for today: OS access control
- Flexible mechanism to support different policies
- Trade-off: an incorrect configuration can lead to insecurity

Confidentiality and integrity

- Access control directly serves two security goals:
- Confidentiality, opposite of information disclosure
- Integrity, opposite of tampering
- By prohibiting read and write operations respectively

Access control policy

- Decision-making aspect of OS
- Should subject S (user or process) be allowed to access object (e.g., file) O?
- Complex, since admininstrator must specify what should happen

Access control matrix

	grades.txt	/dev/hda	/usr/bin/bcvi
Alice	r	rw	rx
Bob	rw	-	rx
Carol	r	-	rx

Slicing the matrix

- $O(nm)$ matrix impractical to store, much less administer
- Columns: access control list (ACL)
  - Convenient to store with object
  - E.g., Unix file permissions
- Rows: capabilities
  - Convenient to store by subject
  - E.g., Unix file descriptors

Groups/roles

- Simplify by factoring out commonality
- Before: users have permissions
- After: users have roles, roles have permissions
- Simple example: Unix groups
- Complex versions called role-based access control (RBAC)

Outline

One namespace

- All files can be accessed via *absolute pathnames* made of directory components separated by slashes
- I.e., everything is a descendant of a root directory named /

Filesystems and mounting

- There may be multiple filesystems, like disk partitions or removable devices
- One filesystem is the root filesystem that includes the root directory
- Other filesystems are mounted in place of a directory
  - E.g., `/media/smccaman/mp3player/podcast.mp3`

Special files and devices

- Some hardware devices (disks, serial ports) also look like files
  - Usually kept under `/dev`
- Some special data sources look like devices
  - `/dev/null`, `/dev/zero`, `/dev/urandom`
- Some OS data also available via `/proc` and `sys` filesystems
  - E.g., `/proc/self/maps`

Current directory, relative paths

- At a given moment, each process has a current working directory
  - Changed by `cd` shell command, `chdir` system call
- Pathnames that do not start with / are interpreted *relative* to the current directory

Inodes

- Most information about a file is a structure called an inode
- Includes size, owner, permissions, and a unique inode number
- Inodes exist independently of pathnames

Directory entries and links

- A directory is a list of directory entries, each mapping from a name to an inode
- These mappings are also called links
- "Deleting a file" is really removing a directory entry
  - The system call `unlink`

Entries . and ..

- Every directory contains entries named . and ..
- . links back to the directory itself
- .. links back to the *parent* directory, or itself for the root

(Hard) links

- Multiple directory entries can link to the same inode
- These are called hard links
- Only allowed within one filesystem, and not for directories

Symbolic links

- Symbolic links are a different linking method
- A symbolic link is an inode that contains a pathname
- Most system calls follow symbolic as well as hard links to operate on they point to

Outline

UIDs and GIDs

- To kernel, users and groups are just numeric identifiers
- Names are a user-space nicety
  - E.g., `/etc/passwd` mapping
- Historically 16-bit, now 32
- User 0 is the special superuser `root`
  - Exempt from all access control checks

File mode bits

- Core permissions are 9 bits, three groups of three
- Read, write, execute for user, group, other
- `ls` format: `rwx r-x r--`
- Octal format: 0754

Interpretation of mode bits

- File also has one user and group ID
- Choose one set of bits
  - If users match, use user bits
  - If subject is in the group, use group bits
  - Otherwise, use other bits
- Note no fallback, so can stop yourself or have negative groups

Directory mode bits

- Same bits, slightly different interpretation
- Read: list contents (e.g., `ls`)
- Write: add or delete files
- Execute: traverse
- X but not R means: have to know the names

Other permission rules

- Only file owner or root can change permissions
- Only root can change file owner
  - Former System V behavior: "give away `chown`"
- Setuid/gid bits cleared on `chown`
  - Set owner first, then enable setuid

Non-checks

- File permissions on `stat`
- File permissions on link, unlink, rename
- File permissions on read, write
- Parent directory permissions generally
  - Except traversal
  - I.e., permissions not automatically recursive

Outline

Process UIDs and `setuid(2)`

- UID is inherited by child processes, and an unprivileged process can't change it
- But there are syscalls root can use to change the UID, starting with `setuid`
- E.g., login program, SSH server

Setuid programs, different UIDs

- If 04000 "setuid" bit set, newly exec'd process will take UID of its file owner
  - Other side conditions, like process not traced
- Specifically the *effective UID* is changed, while the *real UID* is unchanged
  - Shows who called you, allows switching back

More different UIDs

- Two mechanisms for temporary switching:
  - Swap real UID and effective UID (BSD)
  - Remember *saved UID*, allow switching to it (System V)
- Modern systems support both mechanisms at the same time

Setgid, games

- Setgid bit 02000 mostly analogous to setuid
- But note no supergroup, so UID 0 is still special
- Classic application: setgid `games` for managing high-score files

Special case: `/tmp`

- We'd like to allow anyone to make files in `/tmp`
- So, everyone should have write permission
- But don't want Alice deleting Bob's files
- Solution: "sticky bit" 01000

Special case: group inheritance

- When using group to manage permissions, want a whole tree to have a single group
- When 02000 bit set, newly created entries with have the parent's group
  - (Historic BSD behavior)
- Also, directories will themselves inherit 02000

Other permission rules

- Only file owner or root can change permissions
- Only root can change file owner
  - Former System V behavior: "give away `chown`"
- Setuid/gid bits cleared on `chown`
  - Set owner first, then enable setuid