Least-Privilege Isolation: The OKWS Web Server

Brad Karp
UCL Computer Science

CS GZ03 / M030
13th December 2017
Can We Prevent All Exploits?

• Many varieties of exploits
  – Stack smashing, format strings, heap smashing, return-to-libc

• As many proposed defenses
  – W⊕X, ASLR, TaintCheck, StackGuard, ...

• Exploit-specific defenses help, but ever-more vulnerabilities, and adversaries creative

• Not just a problem with C; consider SQL injection in a Python script:
  \[ q = "SELECT orders FROM accounts WHERE name = " + \text{name} \]
  \[ \text{db.execute(q)} \]

• Programmers make errors
Can We Prevent All Exploits?

• Many varieties of exploits
  – Stack smashing, format strings, heap smashing,

If vulnerabilities and errors are here to stay, how can we limit the harm attackers can do when they exploit a server?

vulnerabilities, and adversaries creative

• Not just a problem with C; consider SQL injection in a Python script:
  
  q = "SELECT orders FROM accounts WHERE name = " + name
  
  db.execute(q)

• Programmers make errors
Problem: Sharing Services, But Isolating Data

- Servers often hold sensitive data
  - e.g., amazon.com user’s credit card number
- Single server shared by distinct users, who often shouldn’t see one another’s data
  - e.g., different amazon.com shoppers
- Subsystems on single server must cooperate
  - e.g., amazon.com web interface and back-end order database
- Goal: prevent users from obtaining/modifying data other than their own
  - I shouldn’t be able to retrieve your order (and credit card number), even if I exploit amazon’s web server
Approach: Compartmentalization

• Give each subsystem **minimal access to system data and resources to do its job**
  – If subsystem exploited, at least minimize data it can read or modify

• Define **narrow interfaces between subsystems**, that allow only exact operations required for application

• Design **assuming exploit may occur**, especially in subsystems closest to users
Idea: Principle of Least Privilege (PoLP)

- Each subsystem should only have access to read/modify data needed for its job
- Cannot be enforced within subsystem—must be enforced externally (i.e., by OS)
- Must decompose system into subsystems
  - Must reason carefully about truly minimal set of privileges needed by each subsystem
- Must be able to grant privileges in fine-grained manner
  - Else privileges granted to subsystem may be too generous...
Idea: Privilege Separation

• Determine which subsystems most exposed to attack

• Reduce privileges of most exposed subsystems
  – e.g., amazon payment page can only insert into order database, and order database doesn’t have integrated web interface with direct access to data
  – e.g., ssh login daemon code that processes network input shouldn’t run as root
OKWS: A PoLP Web Server on UNIX

• Before OKWS:
  – Apache web server process monolithic; all code runs as same user
  – Exploit Apache, and all data associated with web service becomes accessible

• How might we separate a web server into subsystems, to apply PoLP?

• Split into multiple processes, each with different, minimal privileges, running as different user IDs
  – Use UNIX isolation mechanisms to prevent subsystems from reading/modifying each other’s data
UNIX Tools for PoLP: chroot()

- **chroot()** system call: set process’s notion of file system root; thereafter, can’t change directories above that point
- So can do:
  ```c
  chdir("/usr/local/alone");
  chroot("/usr/local/alone");
  setuid(61100); (unprivileged user ID)
  ```
- Now process has **no access to any of filesystem but what’s in tree rooted at /usr/local/alone**
  - No access to the many UNIX setuid-root programs, or to sensitive data elsewhere on disk
  - But **must a priori set up all system files needed by process in directory,** e.g., shared libraries, &c.
UNIX Tools for PoLP: File Descriptor Passing

- Initially, parent server process privileged
- Want to run subsystem in child process, but with minimal privileges (e.g., child chroot()ed)
- Idea: privileged parent opens files needed by unprivileged child, passes child open file descriptors to these files when it fork()s child
  - Child can read these files, even if it can’t open them (i.e., because of chroot())
- Can also pass file descriptors dynamically (after fork()) with sendmsg()
  - Process that faces network can accept connection, pass socket for that connection to another process
UNIX Tools for PoLP: File Descriptor Passing

- Initially, parent server process **privileged**
- Want to run subsystem in child process, but with

**Powerful primitive:** means can run subsystem with minimal privilege (e.g., can’t bind to privileged port 80), but grant it **specific network connections or specific files**

  - Child can read these files, **even if it can’t open them** (i.e., because of chroot())

- Can also pass file descriptors **dynamically** (after fork()) with **sendmsg()**
  - Process that faces network can accept connection, pass socket for that connection to another process
OKWS System Design

- okd process parses user input, holds no sensitive data
- svc_i process parses user input for one service; runs in chroot(ed) “jail”
- database proxy process only accepts authenticated requests for subset of narrow RPC interface; can read sensitive data
Analyzing Privilege-Separated Designs

- What data does subsystem have access to, with what permissions?
- How complex is the code in a subsystem (e.g., parsing notoriously hard to get right)?
- What input does a subsystem receive?
  - Less structured → more worrying
  - e.g., okld runs as root; should we worry about exploits of it?
Strength of Isolation vs. Performance

- One process per user gives strictest isolation, but means many, many processes → low performance
- OKWS uses one process per service for performance reasons; so compromised service may reveal one user’s data to another
OKWS Summary

• Shows that PoLP and privilege separation hold real promise for limiting harm exploits can do

• Programming model for services requires new style of programming
  – Can’t use the file system; services chroot()ed
  – Must define narrow, per-service interfaces to database
  – Must communicate explicitly using RPC between service and database