6.858 Lecture 4
OKWS

Administrivia:

Lab 1 due this Friday.

Today's lecture: How to build a secure web server on Unix. The design of our lab web server, zookws, is inspired by OKWS.

Privilege separation
- Big security idea
- Split system into modules, each with their own privilege
    - Idea: if one module is compromised, then other modules won't be
- Use often:
    - Virtual machines (e.g., run web site in its own virtual machine)
    - SSH (seperates sshd, agent)
- Challenges:
    - Modules need to share
    - Need OS support
    - Need to use OS carefully to set things up correctly
    - Performance

OKWS
- Interesting case study of privilege separation
    - Lots of sharing between services
        - strict partitioning doesn't work
    - Lots of code
- Not widely used outside of OKcupid
    - Many web sites have their privilege separation plan
    - But no papers describing their plans

Background: security and protection in Unix
Typical principals: user IDs, group IDs (32-bit integers).
- Each process has a user ID (uid), and a list of group IDs (gid + grouplist).
- For mostly-historical reasons, a process has a gid + extra grouplist.
- Superuser principal (root) represented by uid=0, bypasses most checks.

What are the objects + ops in Unix, and how does the OS do access control?
1. Files, directories.
    - File operations: read, write, execute, change perms, ..
    - Directory operations: lookup, create, remove, rename, change perms, ..
    - Each inode has an owner user and group.
    - Each inode has read, write, execute perms for user, group, others.
    - Typically represented as a bit vector written base 8 (octal); octal works well because each digit is 3 bits (read, write, exec).

- Who can change permissions on files? Only user owner (process UID).
- Hard link to file: need write permission to file.
    - Possible rationale: quotas.
    - Possible rationale: prevent hard-linking /etc/passwd to /var/mail/root, with a world-writable /var/mail.
- Execute for directory means being able to lookup names (but not ls).
- Checks for process opening file /etc/passwd:
    - Must be able to look up 'etc' in /, 'passwd' in /etc.
    - Must be able to open /etc/passwd (read or read-write).
- Suppose you want file readable to intersection of group1 and group2.
    - Is it possible to implement this in Unix?

2. File descriptors.
    - File access control checks performed at file open.
    - Once process has an open file descriptor, can continue accessing.
    - Processes can pass file descriptors (via Unix domain sockets).
3. Processes.
    - What can you do to a process?
        - debug (ptrace), send signal, wait for exit & get status, ..
    - Debugging, sending signals: must have same UID (almost).
        - Various exceptions, this gets tricky in practice.
    - Waiting / getting exit status: must be parent of that process.
4. Memory.
    - One process cannot generally name memory in another process.
    - Exception: debug mechanisms.
    - Exception: memory-mapped files.
5. Networking.
    - Operations:
        - bind to a port
        - connect to some address
        - read/write a connection
        - send/receive raw packets
    - Rules:
        - only root (UID 0) can bind to ports below 1024; (e.g., arbitrary user cannot run a web server on port 80.)
        - only root can send/receive raw packets.
        - any process can connect to any address.
        - can only read/write data on connection that a process has an fd for.
    - Additionally, firewall imposes its own checks, unrelated to processes.

How does the principal of a process get set?

- System calls: setuid(), setgid(), setgroups().
- Only root (UID 0) can call these system calls (to first approximation).

Where does the user ID, group ID list come from?

- On a typical Unix system, login program runs as root (UID 0)
- Checks supplied user password against /etc/shadow.

- Finds user's UID based on /etc/passwd.
- Finds user's groups based on /etc/group.
- Calls setuid(), setgid(), setgroups() before running user's shell

How do you regain privileges after switching to a non-root user?

- Could use file descriptor passing (but have to write specialized code)
- Kernel mechanism: setuid/setgid binaries.
    - When the binary is executed, set process UID or GID to binary owner.
    - Specified with a special bit in the file's permissions.
    - For example, su / sudo binaries are typically setuid root.
    - Even if your shell is not root, can run "su otheruser"
    - su process will check passwd, run shell as otheruser if OK.
    - Many such programs on Unix, since root privileges often needed.
- Why might setuid-binaries be a bad idea, security-wise?
    - Many ways for adversary (caller of binary) to manipulate process.
    - In Unix, exec'ed process inherits environment vars, file descriptors, ..
    - Libraries that a setuid program might use not sufficiently paranoid
    - Historically, many vulnerabilities (e.g. pass $LD_PRELOAD, ..)

How to prevent a malicious program from exploiting setuid-root binaries?

- Kernel mechanism: chroot
    - Changes what '/' means when opening files by path name.
    - Cannot name files (e.g. setuid binaries) outside chroot tree.
- For example, OKWS uses chroot to restrict programs to /var/okws/run, ..
- Kernel also ensures that '/../' does not allow escape from chroot.
- Why chroot only allowed for root?
    - setuid binaries (like su) can get confused about what's /etc/passwd.
    - many kernel implementations (inadvertently?) allow recursive calls to chroot() to escape from chroot jail, so chroot is not an effective security mechanism for a process running as root.
- Why hasn't chroot been fixed to confine a root process in that dir?
    - Root can write kern mem, load kern modules, access disk sectors, ..

Background: traditional web server architecture (Apache).

- Apache runs N identical processes, handling HTTP requests.
- All processes run as user 'www'.
- Application code (e.g. PHP) typically runs inside each of N apache processes.
- Any accesses to OS state (files, processes, ...) performed by www's UID.
- Storage: SQL database, typically one connection with full access to DB.
    - Database principal is the entire application.
- Problem: if any component is compromised, adversary gets all the data.
- What kind of attacks might occur in a web application?
    - Unintended data disclosure (getting page source code, hidden files, ..)
    - Remote code execution (e.g., buffer overflow in Apache)
    - Buggy application code (hard to write secure PHP code), e.g. SQL inj.
    - Attacks on web browsers (cross-site scripting attacks)

Back to OKWS: what's their application / motivation?

- Dating web site: worried about data secrecy.
- Not so worried about adversary breaking in and sending spam.
- Lots of server-side code execution: matching, profile updates, ...
- Must have sharing between users (e.g. matching) -- cannot just partition.
- Good summary of overall plan: "aspects most vulnerable to attack are least useful to attackers"

Why is this hard?

- Unix makes it tricky to reduce privileges (chroot, UIDs, ..)
- Applications need to share state in complicated ways.
- Unix and SQL databases don't have fine-grained sharing control mechanisms.

How does OKWS partition the web server? (Figure 1 in paper)

- How does a request flow in this web server?

```
okd -> oklogd
    -> pubd
    -> svc -> dbproxy
           -> oklogd
```

- How does this design map onto physical machines?
    - Probably many front-end machines (okld, okd, pubd, oklogd, svc)
    - Several DB machines (dbproxy, DB)

How do these components interact?

- okld sets up socketpairs (bidirectional pipes) for each service.
    - One socketpair for control RPC requests (e.g., "get a new log socketpair").
    - One socketpair for logging (okld has to get it from oklogd first via RPC).
    - For HTTP services: one socketpair for forwarding HTTP connections.
    - For okd: the server-side FDs for HTTP services' socketpairs (HTTP+RPC).
- okd listens on a separate socket for control requests (repub, relaunch).
    - Seems to be port 11277 in Figure 1, but a Unix domain socket in OKWS code.
    - For repub, okd talks to pubd to generate new templates, then sends generated templates to each service via RPC control channel.
- Services talk to DB proxy over TCP (connect by port number).

How does OKWS enforce isolation between components in Figure 1?

- Each service runs as a separate UID and GID.
- chroot used to confine each process to a separate directory (almost).
- Components communicate via pipes (or rather, Unix domain socket pairs).
- File descriptor passing used to pass around HTTP connections.
- What's the point of okld?
- Why isn't okld the same as okd?

- Why does okld need to run as root? (Port 80, chroot/setuid.)
- What does it take for okld to launch a service?
  - Create socket pairs
  - Get new socket to oklogd
  - fork, setuid/setgid, exec the service
  - Pass control sockets to okd
- What's the point of oklogd?
- What's the point of pubd?
- Why do we need a database proxy?
  - Ensure that each service cannot fetch other data, if it is compromised.
    - DB proxy protocol defined by app developer, depending on what app requires.
    - One likely-common kind of proxy is a templatized SQL query.
    - Proxy enforces overall query structure (select, update), but allows client to fill in query parameters.
  - Where does the 20-byte token come from? Passed as arguments to service.
  - Who checks the token? DB proxy has list of tokens (& allowed queries?)
  - Who generates token? Not clear; manual by system administrator?
  - What if token disclosed? Compromised component could issue queries.
- Table 1: why are all services and okld in the same chroot? Is it a problem?
  - How would we decide? What are the readable, writable files there?
  - Readable: shared libraries containing service code.
  - Writable: each service can write to its own /cores/<uid>.
  - Where's the config file? /etc/okws_config, kept in memory by okld.
  - oklogd & pubd have separate chroots because they have important state: oklogd's chroot contains the log file, want to ensure it's not modified. pubd's chroot contains the templates, want to avoid disclosing them (?).
- Why does OKWS need a separate GID for every service?
  - Need to execute binary, but file ownership allows chmod.
  - Solution: binaries owned by root, service is group owner, mode 0410.
  - Why 0410 (user read, group execute), and not 0510 (user read & exec)?
- Why not process per user? Is per user strictly better? user X service?
  - Per-service isolation probably made sense for okcupid given their apps. (i.e. perhaps they need a lot of sharing between users anyway?)
  - Per-user isolation requires allocating UIDs per user, complicating okld, and reducing performance (though may still be OK for some use cases).

Does OKWS achieve its goal?

- What attacks from the list of typical web attacks does OKWS solve, and how?
  - Most things other than XSS are addressed.
  - XSS sort-of addressed through using specialized template routines.
- What's the effect of each component being compromised, and "attack surface"?
  - okld: root access to web server machine, but maybe not to DB.
    - attack surface: small (no user input other than svc exit).

- okd: intercept/modify all user HTTP reqs/responses, steal passwords.
    - attack surface: parsing the first line of HTTP request; control requests.
- pubd: corrupt templates, leverage to maybe exploit bug in some service?
    - attack surface: requests to fetch templates from okd.
- oklogd: corrupt/ignore/remove/falsify log entries
    - attack surface: log messages from okd, okld, svcs
- service: send garbage to user, access data for svc (modulo dbproxy)
    - attack surface: HTTP requests from users (+ control msgs from okd)
- dbproxy: access/change all user data in the database it's talking to
    - attack surface: requests from authorized services, requests from unauthorized services (easy to drop)

- OS kernel is part of the attack surface once a single service is compromised.
  - Linux kernel vulnerabilities rare, but still show up several times a year.
- OKWS assumes developer does the right thing at design level (maybe not impl):
  - Split web application into separate services (not clump all into one).
  - Define precise protocols for DB proxy (otherwise any service gets any data).
- Performance?
  - Seems better than most alternatives.
  - Better performance under load (so, resists DoS attacks to some extent)
- How does OKWS compare to Apache?
  - Overall, better design.
  - okld runs as root, vs. nothing in Apache, but probably minor.
  - Neither has a great solution to client-side vulnerabilities (XSS, ..)
- How might an adversary try to compromise a system like OKWS?
  - Exploit buffer overflows or other vulnerabilities in C++ code.
  - Find a SQL injection attack in some dbproxy.
  - Find logic bugs in service code.
  - Find cross-site scripting vulnerabilities.

How successful is OKWS?

- Problems described in the paper are still pretty common.
- okcupid.com still runs OKWS, but doesn't seem to be used by other sites.
- C++ might not be a great choice for writing web applications.
  - For many web applications, getting C++ performance might not be critical.
  - Design should be applicable to other languages too (Python, etc).
  - In fact, zookws for labs in 6.858 is inspired by OKWS, runs Python code.
- DB proxy idea hasn't taken off, for typical web applications.
  - But DB proxy is critical to restrict what data a service can access in OKWS.
  - Why? Requires developers to define these APIs: extra work, gets in the way.

    - Can be hard to precisely define the allowed DB queries ahead of time. (Although if it's hard, might be a flag that security policy is fuzzy.)
- Some work on privilege separation for Apache (though still hard to use).
    - Unix makes it hard for non-root users to manipulate user IDs.
    - Performance is a concern (running a separate process for each request).
- scripts.mit.edu has a similar design, running scripts under different UIDs.
    - Mostly worried about isolating users from one another.
    - Paranoid web app developer can create separate locker for each component.
- Sensitive systems do partitioning at a coarser granularity.
    - Credit card processing companies split credit card data vs. everything else.
    - Use virtual machines or physical machine isolation to split apps, DBs, ..

How could you integrate modern Web application frameworks with OKWS?
- Need to help okd figure out how to route requests to services.
- Need to implement DB proxies, or some variant thereof, to protect data.
    - Depends on how amenable the app code is to static analysis.
    - Or need to ask programmer to annotate services w/ queries they can run.
- Need to ensure app code can run in separate processes (probably OK).

References:

- http://css.csail.mit.edu/6.858/2014/readings/setuid.pdf
- http://httpd.apache.org/docs/trunk/suexec.html

6.858 Computer Systems Security
Fall 2014