Computer Systems Security

> Author: Guanzhou (Jose) Hu 胡冠洲 @ MIT 6.858
>
> Teachers: Nickolai Zeldovich & Frans Kaashoek

Overview of Security

The main goal of a *secure* computer system is to protect it from all potential adversary.

Security Concerns

Security involves:

- **Goal**: what your system is trying to achieve, e.g., only Alice should read file `F`
- **Policy**: rules that enforce the goal to be achieved (confidentiality, integrity, availability, ...)
  - Good practice is to be *conservative*
  - Hard to state precisely
- **Threat model**: assumptions about bad guys, e.g.,
  - Can guess passwd, but cannot physically steal our server
  - Computing power, e.g., 56 bits keys are easy to enumerate brute-forcely nowadays
  - Centralized authority that is easy to attack against, e.g., any one of SSL/TLS CAs
  - Human factor $\rightarrow$ the weakest link & hard to prevent
  - ...
- **Mechanism**s: SW/HW to ensure our policies are satisfied (i.e., implementation of the policy) given threat model
  - Tiny bugs matter
  - Often *layered*: mechanism of one layer is policy of next level down

> Attacks to insecure systems (especially imperfect mechanisms) are often called *Exploits*.

Modern computer systems should involve the idea of building a complete **security architecture**. Example: Google infrastructure security architecture.

Why Security is Hard?

Security is very hard in the senses that:

1. Testing whether a positive function works is trivial, but defending yourself from all potential kinds of adversary is **hard to be comprehensive**. Almost every system has a *breaking point*, but having that does not mean your system is useless.

   ⇒ Theoretically, hard to formalize a good thread model & design a good set of policies against it.

2. Any **tiny mistakes matter**. Any bug in the system might give adversary the chance to break the whole system's security policy. Every detail has a chance of being really mattering. Many famous *mechanism failures* originate in a tiny (but essential) bug.

   ⇒ Even if we have a strong theory fundation, practically, the implementation is likely to be imperfect.

Workarounds to overcome such difficulty:

- Encourage people to report vulnerabilities & Keep iterating
- Recovery plan backups
- Perfect security is rarely needed: making cost of attack greater than the reward is usually enough
- Secure defaults: default to "private" instead of "public"
- ...

> See *social engineering*!

User Authentication

Registration - Authentication - Recovery

Passwords

A **passwd** is essentially a secret setted up during registration and shared between user & server. Server stores:

- Simple: user $\mapsto$ passwd
- Improved: user $\mapsto hash(\text{passwd})$

Example attack: using a *Rainbow table*. Example defense: including a *Salt* in the hash: $hash(\text{salt}, \text{passwd})$.

Weaknesses of Passwords

Passwords scheme is quite vulnerable to various attacks:

- Hack into the server to steal clear text
- Eavesdrop the communication network
- Brute-force guessing - Passwords in real world have very skewed distribution, making it easier for attackers to attack by trying those top choices
- *Human factors*: passwds are often very weak
- Password recovery questions: lower the *entropy* of passwds to $\min\left(E_{\text{passwd}}, E_{\text{recovery\_Q}}\right)$

Defenses

Various defenses are put up against passwd attacks:

1. Avoid clear text: *cryptographic* defenses, e.g., hashes, salt, ...
2. Defend against network eavesdropping: *challenge/response* protocol:

```
|client| -----  Hi, I'm Mike.  ----> |server|
         <----        C        -----   (challenge)
         ----- hash(C, passwd) ---->   (response specific to that challenge)
```

3. *Anti-hammering* defenses:
   - Limit # password guesses
   - Add time-outs
4. Reduce the risk of *MITM* ('man in the middle") & *phishing* attacks: *multi-factor authentication* (MFA), often 2-factor nowadays

Separation & Isolation

Big idea: Avoiding software bugs in user-written code is very hard. Instead, we design a system to have certain-level of security when assuming user-written bugs *exist*.

Privilege Separation (Lab 2)

Chopping the system into multiple small pieces (boxes) with *least priviledge*. Benefits include:

1. Limit the damage of attacks ⇒ make compromises "partial"
2. Shrink access to buggy code ⇒ Reduce the attack surface

To achieve privilege separation, we must face the following challenges:

- How to separate
- How to isolate different pieces: `chroot` jails, containers, VMs, physical isolation, ...
  - Left → right - cost ↑
  - Left → right - guaranteed isolation ↑
- How to share: use limited RPCs interface, instead of a common big database
- Maintain performance

> Linux containers (LXCs) are essentially processes with separate FS/`pid`/NET namespace.

A good & thorough example of doing privilege separation for a web server is OKWS, Figure 1. It designs a good architecture of infrastructure + client-written services, makes attack surface of core parts very narrow, and limits the harm of compromising vulnerable client-written services.

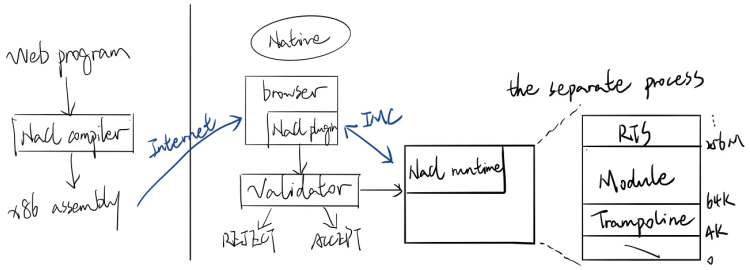
Box	Harm if Compromised	Attack Surface
`okld` (su)	everything	very narrow (no direct interface from external)
`okd`	HTTP traffic	first line of HTTP
`oklogd`	trash the log	logging RPCs
SVC_i	in-memory user states used only by this service	relatively larger
`dbproxy`	database injection	database proxy RPCs

Software Isolation

Sometimes we need compatible (protable on different OS) & high-performance (native execution) isolation. Here we cannot rely on OS support, so we have to implement our own *sandbox*. We need to think about "how to safely run untrusted code natively". Approaches:

- Language specific, e.g., JavaScript safe interpreter
- Language independent (but HW/ISA specific), e.g., static analysis on x86 machine code directly

A very good example of doing software native isolation for web apps is Google Native Client. It introduces static analysis (compiler enhancement + validator) on x86 machine code.



Challenges (attack surface) we need to overcome:

- Variable-length instructions in x86 ⇒ Static validator phases before running:
  1. Force alignment
  2. Build target table (address of every instruction)
  3. Check that all `jmp`s actually jump to a valid target
- Guarding harmful instructions ⇒
  - Compiler adds guard instructions around `mov` / `jmp`
  - Validator disallows `int 0x80` (`syscall`), `ret`
  - Validator double checks compiler's work

- Entering / Leaving the module ⇒ Using the *trampoline* code

> When `nacljmp` is not treated as a whole pseudo-instruction, two vulnerabilities:
>
> 1. The `jmp` in `nacljmp` can be placed at the 32-byte boundary
> 2. A direct jump can jump to the `jmp` in `nacljmp`

Enclaves Technique

Assume a stronger threat model that our operating system is untrusted. This is reasonble as OS kernels might be compromised by:

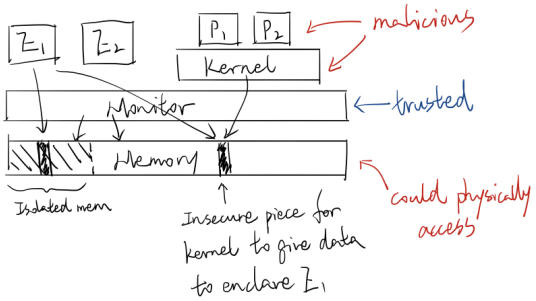
- User / Admin operations
- Physical attacks
- Kernel bugs
- Malware

Previous isolation techniques no longer work because they all assume a trusted OS kernel to provide at least process-level isolation. We now need something that runs alongside with the OS kernel to provide a secure environment. Existing defenses include:

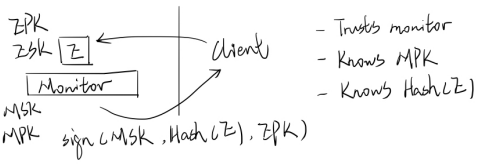
- Secure boot: ensuring that we are booting the right kernel, but does nothing more
- Hypervisor/VMs: hypervisors themselves are complicated pieces of software which are as hard to verify as OS kernels
- Separate CPU (e.g., *Apple Enclaves* for TouchID service): totally independent chipset to hold confidentiality, but costly & not flexible to upgrade

Intel *Software Guard Extensions* (SGX) provide another solution that the CPU has a special set of secure instructions, and holds isolated, encrypted piece of memory only for executing *enclaves*. This solution still suffers from inflexibility to maintain and upgrade.

Microsoft Komodo follows SGX's design, but further decouples enclave management into a separate piece of software called a *monitor*:



Attestation process is used in Komodo for a client to safely verify that the enclave service it is talking to is correct (we do not even trust the running enclave code):



Analysis to Find Bugs

Big idea: though bugs do not necessarily mean they are exploitable, we'd better fix all the bugs.

Symbolic Execution (Lab 3)

Symbolic execution is a novel method/approach tring to find deep bugs comprehensively. Ideas:

1. We specify what variables are inputs - mark them *symbolic*
2. Compute on symbolic values, e.g., `x =` x `=` $y \cdot z$, instead of concrete values, e.g., `x = 10`
3. Whenever branching, create path conditions (i.e., *constraints*)
4. Use *solver* to see if a branch has solution (is possible). If not, then no need to explore that branch
5. (Ideally), Exhausively explore all possible paths until it succeeds or triggers/raises exception
6. For every explored path, give the set of inputs that can trigger it

This idea is brought up by the EXE paper, whose design is as the following:

- Compile-time: use a C-to-C compiler to instrument the code
  - Mark input variables symbolic, and get their possible value range
  - Instrument all branches (`if` statements) to call the run-time solver
- Run-time: solver listens for constraint solving and give solutions; scheduler adds unexplored branches in queue

Example:

```
// Example skipped. Check the paper ;)
```

EXE is not guaranteed to find all bugs:

- There are constraints that are very hard to solve, so we set a timeout bound in solver
- For dereferencing symbolic pointers, we collapse downto one concrete value
- Floating-point numbers are not modeled
- Non-deterministic code (random numbers) are not modeled
- We are always at the risk of exponential *path explosion*

Symbolic values are in enssence *bitarrays* of length of their datatype. *Concolic* execution (concrete + symbolic) is used in practice to overcome the difficulty that some operations are very hard to formalize & solve in a symbolic way, e.g., $x \cdot x = 900$, or $\text{sha256}(x) = 10$.

###Other Bug-Finding Techniques

Other testing / analysis techniques:

- Test cases & Unit testing: can only go for intended cases (known bugs)
- Static analysis
- Formal methods
- Fuzzing tests: can find some unintended bugs, but low chance to find "deep" bugs

> Open-source software community is very open to bugs and exploits, so that we can learn from them and build better software.
>
> Related: Zero-day attacks.

Web Security

Browser Security (Lab 4)

Same *browser* talking to many different sources (*sites*), running tangled HTML, JS, CSS, ... We should tease different sites apart. Web security sits in between desktop security (old design, very complex) & mobile security (novel design, well-structured).

We consider the specific threat model: the attacker hosts a website `a.com` and the victim visits `a.com` meanwhile also browsing other important sites. Assume the browser is trusted and bug-free.

Browser identifies a server by HTTPS (TLS + certificates), and server identifies a client through *cookies*. By design, all neeeded cookies are sent by browser automatically. Example cookie table maintained by the browser:

Domain Suffix	Key	Value
mit.edu	Athena	`josehu`
google.com	Session ID	ABCDEFGxxx
...	...	...

The core defense strategy against attacks is the *same-origin policy* (SOP):

- Define the *origin* for all scripts & resources to be protocol + domain + port where it came from, e.g., https://mit.edu:443
- Permit a script to touch a piece of resource, including pages, JS variables, and cookies, if from the same origin
- Several special exceptions, mostly embedded `href`, `img`s, scripts, and `iframe`s ⇒ CSRF
- Browser side channel attack tricks to bypass SOP: summary

Some other common vulnerabilities which, when exploited, completely bypasses SOP:

- Carelessly escaped user input (through GET, POST, ...) put in page template ⇒ XSS
- Use atop invisible frames to steal user inputs to framed pages; Screenshotting children frames
- Phishing attacks can often fool users

- Serious XSS vulnerabilities can sometimes make *worms* (*viruses*) possible

Many details, such as cross-site cookies, cookies overwriting, *cross-site scripting* (XSS), *cross-site request forgery* (CSRF), *clickjacking*, *phishing*, and top-level domains listing, are discussed in the "The Tangled Web" book. Also check Mozilla's page.

Internet Protocols Security

The network protocol layers, including TCP/IP, DNS, BGP, Telnet/SSH, and FTP, have several security weaknesses that might greatly influence the rapidly growing Internet. (Note that we are now interested in the protocol design, not specific implementation stack bugs like Linux kernel bugs or *Heartbleed*.)

Security was not a No.1 concern when people were designing these fundational protocols widely used today. Also, *active* attacks such as MITM attacks are very plausible nowadays. And, *liveness* is at core of the Internet today, so *DoS* attacks matter.

> Also see *secure messaging* and *secure emails*.

Things around TCP/IP

- Telnet: no crypto; got extremely bad when *Ethernet* got popular ⇒ `Rlogin`: do not send passwords, with the basic assumption that it is hard to fake source address; long time later people realized that IP addresses should not be used as authentication
- TCP/IP: see this paper, *sequence number guessing* attack and its extensions, ... ⇒ do not use IP address-based authentication, use end-to-end (E2E) strong cryptos instead; we can also harden TCP, e.g., randomize per conn seq number increment as $ISN_s = ISN_{old} + \text{SHA1}(src, dst, secret)$
- DoS attacks: SYN flooding, ICMP broadcast amplification & DNS amplification (reflection) for DDoS, ...
- Routing protocols - DHCP (ARP), BGP (RIP): attacker fakes itself (broadcasts itself) as another identity ⇒ quite hard to get rid of

> Check the *Internet Engineering Task Force* (IETF) stardard organization.

SSL/TLS Secure Channel

SSL/TLS secure channels: see this paper; Encryption ↦ Confidentiality, Signature ↦ Authenticity & Integrity, Tagging ↦ Integrity

- Asymmetric (strong but slow):
  - $\text{keygen}() = (pk, sk)$, $\text{Encrypt}(pk, m) = c$, $\text{Decrypt}(sk, c) = m$
  - $\text{sign}(sk, h(m)) = sig$, $\text{verify}(pk, m, sig) = T/F$
- Symmetric (weaker but fast):
  - $\text{keygen}() = k$, $\text{Encrypt}(k, m) = c$, $\text{Decrypt}(k, c) = m$
  - $\text{MAC}(k, m) = tag$, then checks the tag; AES, ...

Possible attacks & their defenses in secure channel communication:

1. Authenticating the server (avoid MITM attacks):
   - *Certificate Authority* (CA) signing $\{name, pk\}_{sk_{CA}}$, see more below
   - *Trust-on-First-Use* (TOFU) like what SSH does
2. Message Integrity: $E(k, m) \mid\mid \text{MAC}(k, m)$; *replay* attacks can be avoided by message sequence numbers
3. *Forward Secrecy*: use a short-lived session key for symmetric encryption key exchange and delete the session private key immediately after common knowledge has been established, so that leaking a main server private key does not leak old
4. communications

> SSL 3.0 is actually considered *deprecated* & *forbidden* starting from 2015. Nailing weak point: the *Poodle* attack. Now we are at TLS 1.2-1.3.

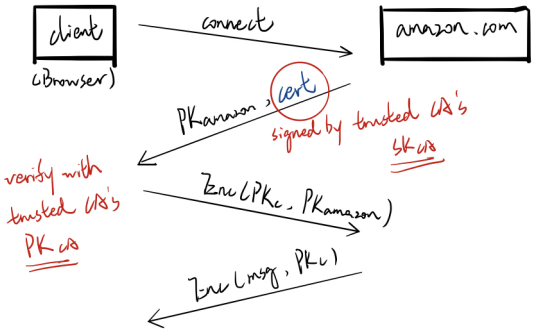
HTTPS CA/Browser Model

Once we have channels secured, how can we be confident that we are talking to the right server, not some MITM attacker? In other words, how can I verify the publick key from server pk_s is actually the public key of that service?

People are relying on *Certificate Authorities* (CAs). A trusted CA signs with its secret key the mapping from a domain name to a public key (called a *certificate*):

$$\text{cert} = \{\text{amazon.com}, pk_{\text{amazon}}, \text{expiration date}\}_{sk_{CA}}$$

The message flow between client and server now becomes:



The client (browser) checks:

- CA is trusted; Can verify the cert with that CA's public key
- Domain name matches
- Expiry information
- Server's public key matches

Such a model puts CAs outside of the common path of browsing, thus better performance, availability, and privacy. However, several things this model does not effectively defend against:

- Phishing - users fooled to type in wrong domain
- Users tricked to use plain HTTP instead of HTTPS
- Using CDNs
- Links, scripts, ...
- How CAs operate & Compromised CAs ⇒ Key pinning? DANE? Certificate revocation?

> Certificate Transparency (CT): Certificate log as a Mekrle tree, Log server (might be malicious) & gossiping among browsers.

Privacy & Anonymity

Anonymous communication hides the source and destination of flow traffic. One famous example is Onion Routing & *Tor*.

- Statistical traffic monitoring attacks
- Exit node as a weak point
- DoS attacks on entry & exit node

Side-Channel Attacks

A **Side Channel** is some indirect signals / side effects / shared-state changes produced by the processing of the hidden secret key which may leak information about the secret. Side channels are NOT part of the designed threat model, so can be very sneaky and dangerous. Side channels may include:

- Electrical magnetic waves
- Power usage
- Physical vibrations
- Various timing
- ... (many more)

CPU Timing Channel

The most famous side channel attacks proposed on CPU architecture are Meltdown & Spectre at 2018. They exploit *speculative execution* (i.e., *out-of-order execution*) features of modern CPUs + timing side channels like *cache*.

The whole kernel is mapped somewhere in each user process's addreass space and normally we cannot access a kernel page in user mode. These two attacks try to read some secret bytes from the kernel memory region.

Meltdown's core idea: speculative execution on raising exceptions + timing channels:

- Relies on the page fault / permission exception when reading a kernel memory address
- Restricted to accessing kernel memory

```
/**
 * Meltdown demo.
 * Wanna steal the byte at `arr[x]`.
 */

// Attacker's snippet.
```

```
raise_exception();  // Exception raised, but CPU speculatively runs following few instructions.
read(probe[arr[x]*4096]);   // The `arr[x]`-th page of `probe` might be loaded in cache.

// Speculative execution might have ended here, but cache is not cleared.

// Timing.
for (b = 0; b < 256; b++)
    timing(read(probe[b*4096]));    // One cache-hit `b` will finish fast.
```

Spectre's core idea: train a mispredicted branch + timing channels:

- Does not rely on exceptions, so can be used to access hidden user-region data as well
- Requires a piece of victim gadget in target code

```
/**
 * Spectre demo.
 * Wanna steal the byte at `arr[x]`.
 */

// Victim gadget.
if (x < arr_length)     // Give in-range x's several times to train branch predictor to believe cond
is true.
    y = probe[arr[x]*4096];     // Then give a malicious offset x where `arr+x` points to the byte we
want.

// Now the `arr[x]`-th page of `probe` might be in cache. Speculative execution might have
// ended here, but cache is not cleared.

// Attacker's timing.
for (b = 0; b < 256; b++)
    timing(read(probe[b*4096]));    // One cache-hit `b` will finish fast.
```

Meltdown & Spectre are very new and powerful. Several defenses like *kernel address space layout randomization* (KASLR) will help mitigate these two attacks - making it hard for the attacker to know where the desired secret lies on the address space, so cannot pick a target `x`.

Other Case Studies

Buffer Overflow (Lab 1)

Buffer overflowing is a famous example of mechanism failure with primitive C programs. Read:

- How buffer overflows are exploited: old article on IA32
- Up-to-date article: Smashing the Stack in the 21st Century
- GDB usage reference: official & mine
- x86_64 cheatsheet: synatx Wikibook, PDF (from Brown), syscall reference

Defense ideas against *stack* buffer overflows:

- Non-executable stacks
- Stack *canaries*, i.e., stack-smashing protector
- *ASLR* (address space layout randomization)

Defense against other buffer overflows (maybe heap, bss, ...) - *Bounds checking*:

- Fat pointers: store alongside the pointer the start & end boundary
- Separate bounds table + OOB bit, e.g., *Baggy Bounds* checking

However, OOB checking is not perfect (section 7 of the Baggy Bounds paper). For example, it cannot catch arithmetics where programmer casts pointers back to integers. Baggy Bounds notes:

1. Pay attention to `struct` layout: low $\rightarrow$ high
2. Bound checking is variable (object) based

Android / iOS Security

Mobile platforms security are also interesting topics.

- Apple iOS security design: white paper explains low-level mobile secure hardware design well

- Android security design: paper explains high-level Apps access controls design well
  - Desktops: Users are principals; Apps all take user's privileges - sharing is easy
  - Mobile: Apps represent different UIDs and each of them has separate privilege

Secure Network FS (Lab 5)

Enlarges the threat model exposed to networking file systems that, FS servers might be malicious as well.

- One possible mechanism: SUNDR
- Achieves the probably strongest guarantee in such scenario - *fork consistency*: once the server lies to a client, it must continue that "fork" on that client and any attempt of merging two different forks will be detected

(Distributed) Denial-of-Service (DDoS)

Often utilizes UDP *source obfuscation* + *amplifications*. Includes DNS, SNMP, and NTP protocols. Possible mitigations:

- DNS over CDNs
- BGP control
- Proper ACLs

Keybase

Trying to bring back convenient trust, authenticity, and key management (user identity $\rightarrow$ public key mapping).

- Legacy solutions typically involve a single private key per user on a single device (USB dongle, phone, ...) and assume that the user keeps that private key secure, remembers it, and uses it for all communications
- Keybase's approach is to think about *devices* instead of a single private key. Each device of a user is equally powerful, and their addition & revocation form a hashed blockchain (kinda like a Bitcoin ledger). To verify a user's identity, just download its last block in chain and check
  - A user is a chain of device additions & removals
  - A team is a chain of user additions & removals
- Uses other media, e.g., a Tweet, as a proof of correct public key
- Similar attempt: Blockstack, using a global decentralized blockchain to re-build the naming infrastructure