Designing and Building **Secure Software**

Making secure software

- **Flawed approach**: Design and build software, and ***ignore security at first***
  - Add security once the functional requirements are satisfied

- **Better approach**: ***Build security in*** from the start
  - Incorporate security-minded thinking into all phases of the development process

Threat Modeling
(Architectural Risk Analysis)

Threat Model

- The **threat model** **makes explicit the adversary's assumed powers**
  - Consequence: The threat model must match reality, otherwise the risk analysis of the system will be wrong

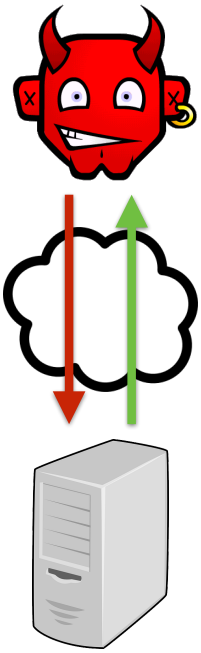
- The threat model is **critically important**
  - If you are not explicit about what the attacker can do, how can you assess whether your design will repel that attacker?

- This is part of **architectural risk analysis**

Example: **Network User**

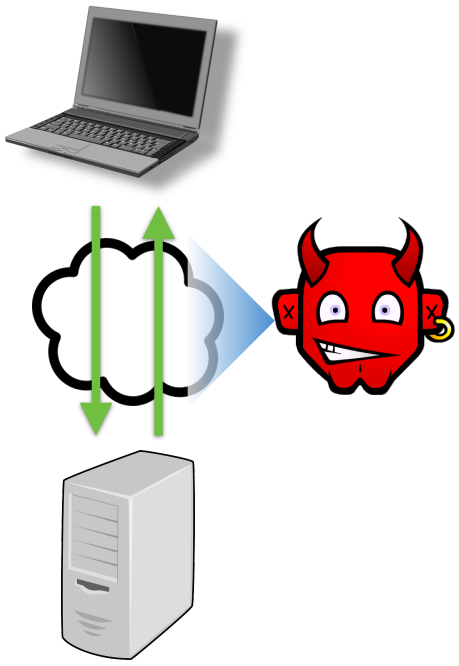
- An (anonymous) user that can connect to a service via the network
- Can:
  - **measure** the size and timing of requests and responses
  - run **parallel sessions**
  - provide **malformed inputs**, **malformed messages**
  - **drop or send extra messages**

- **Example attacks**: SQL injection, XSS, CSRF, buffer overrun/ROP payloads, …



Example: **Snooping User**

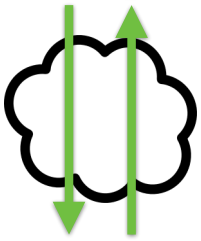
- Internet user **on the same network** as other users of some service
  - For example, someone connected to an unencrypted Wi-Fi network at a coffee shop
- Thus, can additionally
  - **Read/measure** others' **messages**,
  - **Intercept, duplicate,** and **modify messages**
- **Example attacks**: **Session hijacking** (and other data theft), **privacy-violating side-channel attack**, **denial of service**



Example: **Co-located User**

- Internet user **on the same machine** as other users of some service
  - E.g., **malware** installed on a user's laptop

- Thus, can additionally
  - **Read/write** user's **files** (e.g., cookies) and **memory**
  - **Snoop keypresses** and other events
  - Read/write the user's **display** (e.g., to **spoof**)



- **Example attacks**: **Password theft** (and other credentials/secrets)

Bad Model = Bad Security

- Any **assumptions** you make in your model are potential **holes that the adversary can exploit**

- E.g.: Assuming no snooping users **no longer valid**
  - *Prevalence of wi-fi networks in most deployments*
- Other mistaken assumptions
  - **Assumption**: Encrypted traffic carries no information
    - Not true! By analyzing the size and distribution of messages, you can infer application state
  - **Assumption**: Timing channels carry little information
    - Not true! Timing measurements of previous RSA implementations could be used eventually reveal a remote SSL secret key

Finding a good model

- **Compare against similar systems**
  - What attacks does their design contend with?
- **Understand past attacks and attack patterns**
  - How do they apply to your system?
- **Challenge assumptions in your design**
  - What happens if an assumption is untrue?
    - What would a breach potentially cost you?
  - How hard would it be to get rid of an assumption, allowing for a stronger adversary?
    - What would that development cost?

Quiz 1

- When worrying about the malicious capabilities of Javascript code in Web 2.0 (e.g., XSS), what sort of attacker are we considering?

A. Network User
B. Snooping User
C. Co-located User
D. Sleeping User

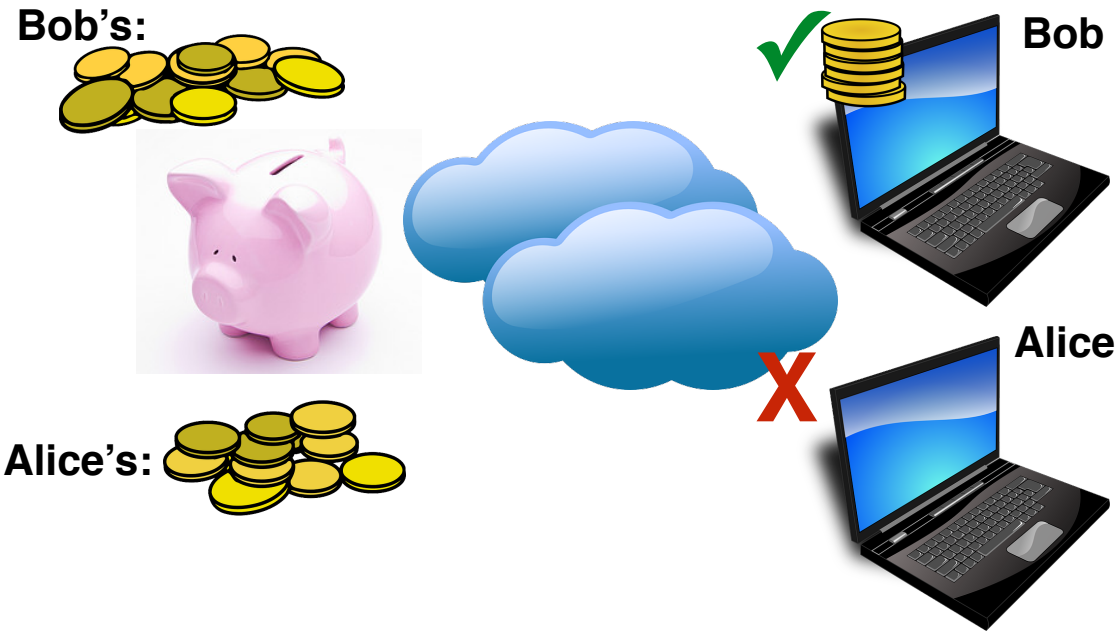
Quiz 1

- When worrying about the malicious capabilities of Javascript code in Web 2.0 (e.g., XSS), what sort of attacker are we considering?

A. Network User
B. Snooping User
C. Co-located User
D. Sleeping User

Security Requirements

Running Example: **On-line banking**



Security Requirements

- **Software requirements** typically about **what** the **software should do**

- We also want to have **security requirements**
  - **Security-related goals** (or **policies**)
    - **Example**: One user's bank account balance should not be learned by, or modified by, another user, unless authorized
  - **Required mechanisms for enforcing them**
    - **Example**:
      1. Users identify themselves using passwords,
      2. Passwords must be "strong," and
      3. The password database is only accessible to login program.

Typical *Kinds* of Requirements

- **Policies**
  - **Confidentiality** (and Privacy and Anonymity)
  - **Integrity**
  - **Availability**

- Supporting **mechanisms**
  - **Authentication**
  - **Authorization**
  - **Auditability**

Privacy and Confidentiality

- *Definition*: **Sensitive information not leaked** to unauthorized parties
  - Called *privacy* for individuals, *confidentiality* for data
- **Example** policy: bank account status (including balance) known only to the account owner
- Leaking **directly** or via **side channels**
  - **Example**: manipulating the system to directly display Bob's bank balance to Alice
  - **Example**: determining Bob has an account at Bank A according to shorter delay on login failure

Secrecy vs. **Privacy**? https://www.youtube.com/watch?v=Nlf7YM71k5U

Anonymity

- A specific **kind of privacy**

- **Example**: Non-account holders should be able to browse the bank informational site without being tracked
  - Here *the adversary is the bank*
  - The previous examples considered other account holders as possible adversaries

Integrity

- *Definition*: **Sensitive information not damaged** by (computations acting on behalf of) unauthorized parties

- **Example**: Only the account owner can authorize withdrawals from her account

- Violations of integrity can also be **direct** or **indirect**
  - **Example**: Being able specifically withdraw from the account vs. confusing the system into doing it

Availability

- *Definition*: A system is **responsive to requests**
- **Example**: a user may always access her account for balance queries or withdrawals
- **Denial of Service (DoS)** attacks attempt to **compromise availability**
  - by busying a system with useless work
  - or cutting off network access

Quiz 2

- An attacker gaining access to your account and defacing your web page is a violation of what policy?

A. Confidentiality
B. Anonymity
C. Integrity
D. Availability

Quiz 2

- An attacker gaining access to your account and defacing your web page is a violation of what policy?

A. Confidentiality
B. Anonymity
C. Integrity
D. Availability

Supporting mechanisms

- Leslie Lamport's **Gold Standard** defines mechanisms provided by a system to enforce its requirements
  - **Au**thentication
  - **Au**thorization
  - **Au**dit

- The gold standard is **both requirement and design**
  - The *sorts of policies* that are authorized *determines* the *authorization mechanism*
  - The *sorts of users* a system has *determines* how they should be *authenticated*

Authentication

- What is the **subject of security policies**?
  - Need to define a ***notion of identity*** and a way to ***connect an action with an identity***
    - *a.k.a. a* **principal**

- **How can system tell a user is who he says he is?**
  - What (only) he **knows** (e.g., password)
  - What he **is** (e.g., biometric)
  - What he **has** (e.g., smartphone)
  - Authentication mechanisms that employ more than one of these factors are called **multi-factor authentication**
    - E.g., bank may employ passwords and text of a special code to a user's smart phone

Authorization

- Defines **when a principal may perform an action**
- **Example**: Bob is authorized to access his own account, but not Alice's account
- There are a wide variety of **policies** that define what actions might be authorized
  - E.g., access control policies, which could be originator based, role-based, user-based, etc.

Audit

- Retain enough information to be able to **determine the circumstances of a breach or misbehavior** (or *establish one did not occur*)
  - Such information, often stored in **log files**, must be **protected from tampering**, and from access that might violate other policies

- **Example**: Every account-related action is logged locally and mirrored at a separate site

Defining Abuse Cases

- Using attack patterns and likely scenarios, construct cases in which an **adversary's exercise of power** could **violate a security requirement**
  - Based on the threat model
  - What might occur if a security measure was removed?

- **Example**: *Co-located attacker* steals password file and learns all user passwords
  - Possible if password file is not encrypted
- **Example**: *Snooping attacker* replays a captured message, effecting a bank withdrawal
  - Possible if messages are have no *nonce*

Quiz 3

- Video cameras at a bank enable what kind of security mechanism?

A. Authentication
B. Authorization
C. Audit
D. Auror

Quiz 3

- Video cameras at a bank enable what kind of security mechanism?

A. Authentication
B. Authorization
C. Audit
D. Auror

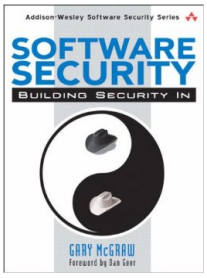
Design Flaws

Design Defects = Flaws

- Recall that software defects consist of both flaws and bugs
  - **Flaws** are problems in the **design**
  - **Bugs** are problems in the **implementation**

- **We avoid flaws during the design phase**

- According to Gary McGraw, **50% of security problems are flaws**
  - So this phase is very important



Secure Software Design

Design software architecture according to good **principles** and **rules**

Risk-based analysis of software architecture's design

Principles and Rules

- A **principle** is a high-level design goal with many possible manifestations
- A **rule** is a specific practice that is consonant with sound design principles
  - The **difference between these two can be fuzzy**, just as design vs. implementation is fuzzy.
    - For example, there is often a *principle underlying specific practices*
  - **Principles often overlap**

- The **software design phase** tends to **focus on principles** for avoiding flaws

Categories of Principles

- **Prevention**
  - **Goal**: Eliminate software defects entirely
  - **Example**: Heartbleed bug would have been prevented by using a type-safe language, like Java
- **Mitigation**
  - **Goal**: Reduce the harm from exploitation of unknown defects
  - **Example**: Run each browser tab in a separate process, so exploitation of one tab does not yield access to data in another
- **Detection** (and **Recovery**)
  - **Goal**: Identify and understand an attack (and undo damage)
  - **Example**: Monitoring (e.g., expected invariants), snapshotting

The Principles

- **Favor simplicity**
  - Use fail-safe defaults
  - Do not expect expert users
- **Trust with reluctance**
  - Employ a small trusted computing base
  - Grant the least privilege possible
    - Promote privacy
    - Compartmentalize
- **Defend in Depth**
  - Use community resources - no security by obscurity
- **Monitor and trace**

Design Category:
Favor Simplicity

Favor Simplicity

- Keep it **so simple** it is **obviously correct**
  - Applies to the external interface, the internal design, and the implementation
    - Classically referred to as **economy of mechanism**
  - **Category**: Prevention

"We've seen **security bugs in almost everything**: operating systems, applications programs, network hardware and software, and security products themselves. **This is a direct result of the complexity of these systems.** The more complex a system is--the more options it has, the more functionality it has, the more interfaces it has, the more interactions it has--the harder it is to analyze [its security]". *—Bruce Schneier*

https://www.schneier.com/essays/archives/1999/11/a_plea_for_simplicit.html

FS: Use fail-safe defaults

- Some **configuration** or **usage choices affect** a system's **security**
  - The length of cryptographic keys
  - The choice of a password
  - Which inputs are deemed valid

- **The default choice should be a secure one**
  - **Default key length is secure** (e.g., 2048-bit RSA keys)
  - **No default password**: cannot run the system without picking one
  - **Whitelist** valid objects, rather than blacklist invalid ones
    - E.g., don't render images from unknown sources

U.S.

The New York Times

Hackers Breach Security of HealthCare.gov

By **ROBERT PEAR** and **NICOLE PERLROTH** SEPT. 4, 2014

WASHINGTON — Hackers breached security at the website of the government's health insurance marketplace, HealthCare.gov, but did not steal any personal information on consumers, Obama administration officials said Thursday.

. . .

Mr. Albright said the hacking was made possible by several security weaknesses. The test server should not have been connected to the Internet, he said, and it came from the manufacturer with a default password that had not been changed.

Blog: SANS Security Trend Line

Simple Math: It Always Costs Less to Avoid a Breach Than to Suffer One

23 sep 2014

Posted by **John Pescatore**
Filed under **Uncategorized**

0 comments

The Home Depot breach is the latest "largest ever," but it is really just another example of "you can pay me now, or you can pay me a **lot** more later" proving out once again as the details come out.

The root cause of the breach can be traced to Home Depot's failure to implement the first subcontrol under Critical Security Control 2:

> **Deploy application whitelisting technology that allows systems to run software only if it is included on the whitelistand prevents execution of all other software on the system.**
>
> **The whitelist may be very extensive (as is available from commercial whitelist vendors), so that users are not inconvenienced when using common software. Or, for some special-purpose systems (which require only a small number of programs to achieve their needed business functionality), the whitelist may be quite narrow.**

"… whitelisting on servers and single function servers or appliances has proven to cause near zero business or IT administration disruption"

Design Category:
Trust with Reluctance

Trust with Reluctance (TwR)

- **Whole system security** depends on the **secure operation of its parts**
  - These parts are **trusted**

- So: **Improve security by reducing the need to trust**
  - By using a better design
  - By using a better implementation process
  - By not making unnecessary assumptions
    - If you use third party code, how do you know what it does?
    - If you are not a crypto expert, why do you think you can design/ implement your own crypto algorithm?

- **Categories**: Prevention and mitigation

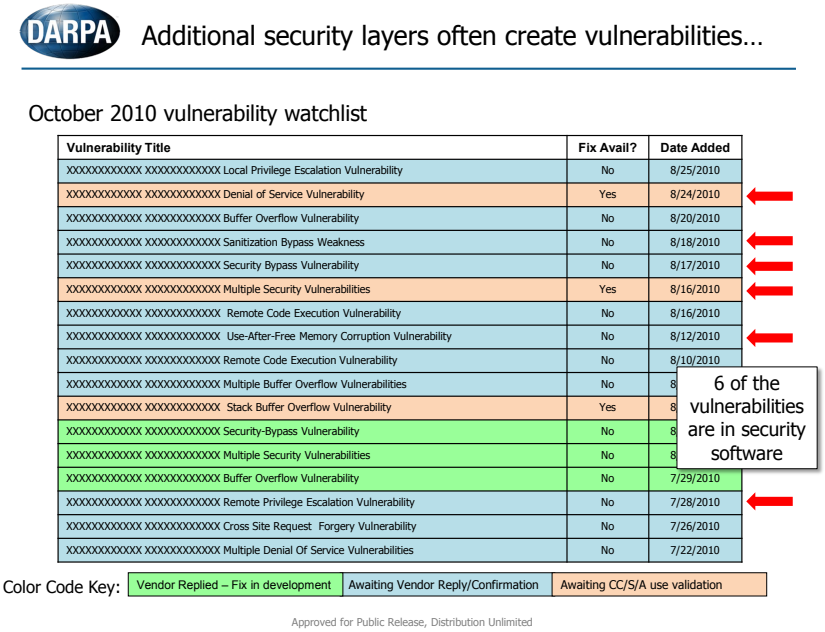
TwR: Small TCB

- Keep the **TCB small** (and simple) to **reduce overall susceptibility to compromise**
  - The trusted computing base (TCB) comprises the system components that *must* work correctly to ensure security
  - **Category**: Prevention

- **Example**: **Operating system kernels**
  - Kernels enforce security policies, but are often millions of lines of code
    - Compromise in a device driver compromises security overall
  - Better: Minimize size of kernel to reduce trusted components
    - Device drivers moved outside of kernel in micro-kernel designs

Failure: Large TCB

- **Security software** is part of the TCB
- But as it grows in size and complexity, it becomes vulnerable itself, and can be bypassed



http://www.darpa.mil/WorkArea/DownloadAsset.aspx?id=2147484449

TwR: Least Privilege

- Don't give a part of the system more privileges than it needs to do its job ("*need to know*")
  - **Category**: Mitigation

- **Example**: Attenuate delegations
  - Mail program delegates to editor for authoring mails
    - `vi, emacs`
  - But many editors permit escaping to a command shell to run arbitrary programs: too much privilege!
  - Better Design: Use a restricted editor (pico)

Lesson: Trust is Transitive

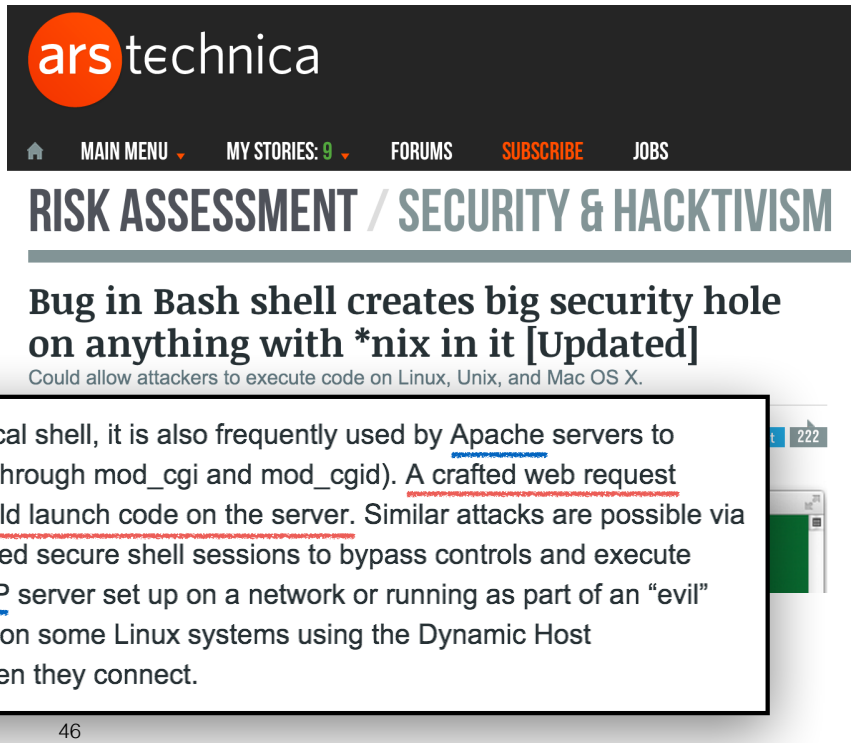
- **If you trust something, you trust what it trusts**
  - *This trust can be misplaced*

- **Previous e-mail client example**
  - Mailer delegates to an arbitrary editor
  - The editor permits running arbitrary code
  - Hence the mailer permits running arbitrary code

Failure: Ignore Attack Surface of External Components

- **Attack surface**: Elements of a system that an adversary can attack, or use in an attack

- **Do third-party components do *only* what I want?**
  - **Shellshock** Failure: "Bourne ag
    we
    oth
    po
    the
    - T
      n



Quiz 4

- Extensive code reuse (e.g., using Linux on your baby monitor) is a violation of what principle?

A. Small Trusted Computing Base
B. Least Privilege
C. Favor Simplicity
D. Trust is Transitive

Quiz 4

- Extensive code reuse (e.g., using Linux on your baby monitor) is a violation of what principle?

A. Small Trusted Computing Base

B. Least Privilege

C. Favor Simplicity

D. Trust is Transitive

Rule: Input validation

- Input validation is a **kind of least privilege**
  - You are **trusting a subsystem** only **under certain circumstances**
  - *Validate that those circumstances hold*

- Several **examples** so far:
  - Trust a given function *if* the range of its parameters is limited (e.g., within the length of a buffer)
  - Trust a client form field *if* it contains no `<script>` tags (and other code-interpretable strings)

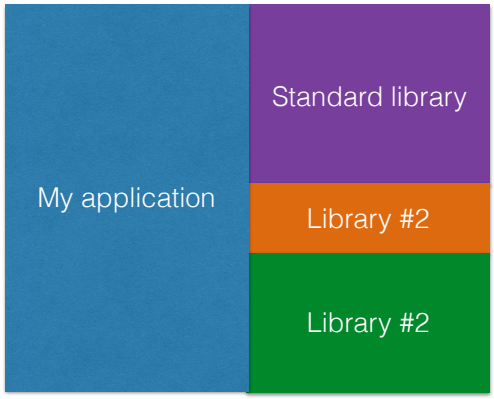
Libraries

- One never writes an entire application on its own
  - We delegate to **libraries**

Dangers:

- The library will **do the wrong thing** with certain inputs
- The library will **delegate to code that does the wrong thing**

- Need to justify trust



Validating library inputs

- Philosophy of **C** library: **I trust my inputs**
  - `fwrite(fp,"foo")` — the `fwrite` code assumes that `fp` will be a valid file pointer
    - Not closed, not NULL, etc.
  - `strcpy(dst,src)` — the `strcpy` code assumes that `dst` has enough space to contain `src`'s contents, and that `src` is **nul**-terminated
  - Etc.
- **Many libraries trust their inputs**
- So it's the **client's responsibility to check them**

Library validating its input

- Library routines can also **check their own input**
  - This is what our `catwrapper.rb` program was doing — didn't expect the web server to do it
  - Likewise `fwrite` *could* have been written to check that its `fp` argument is valid (e.g., non-`NULL`).
- In general: **validation in the library is safer**
  - Programmers forget, and are lazy
  - One library, many clients: Fewer chances for mistakes
- But **validation may be context dependent**
  - and is less efficient when client already knows it's valid

TwR: Compartmentalization

- **Isolate a system component** in a compartment, or **sandbox**, reducing its privilege by making certain interactions impossible
  - **Category**: Prevention and Mitigation

- **Example**: Disconnect student records database from the Internet
  - Grant access only be direct terminals

- **Example**: `Seccomp` system call in Linux
  - Enables compartments for untrusted code

Minimize privilege

Example:

- Our `catwrapper.rb` program was able to read, write, and delete all files.
- We could reduce the permissions
  - on the directory to forbid deletions, and
  - on the file to forbid writing
- Puts less trust in `catwrapper.rb` to do the right thing — *defense in depth*

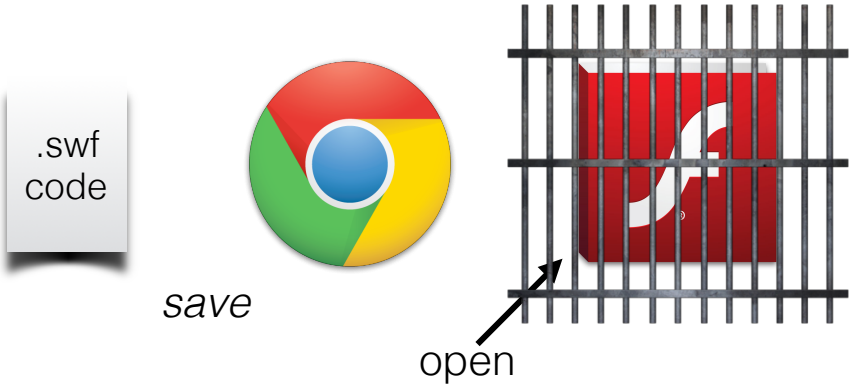
SecComp

- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform `read`, `write`, `exit`, and `sigreturn` system calls**
    - No support for `open` call: Can only use already-open file descriptors
  - **Isolates a process by limiting possible interactions**

- Follow-on work produced **seccomp-bpf**
  - **Limit process to policy-specific set of system calls**, subject to a policy handled by the kernel
    - Policy akin to *Berkeley Packet Filters (BPF)*
  - Used by *Chrome*, *OpenSSH*, *vsftpd*, and others

Idea: Isolate Flash Player

- Receive .swf code, save it
- Call **`fork`** to create a new process
- In the new process, open the file
- Call **`exec`** to run Flash player
- Call **`seccomp-bpf`** to compartmentalize



Design Categories: **Defense in Depth** and **Monitoring/Traceability**

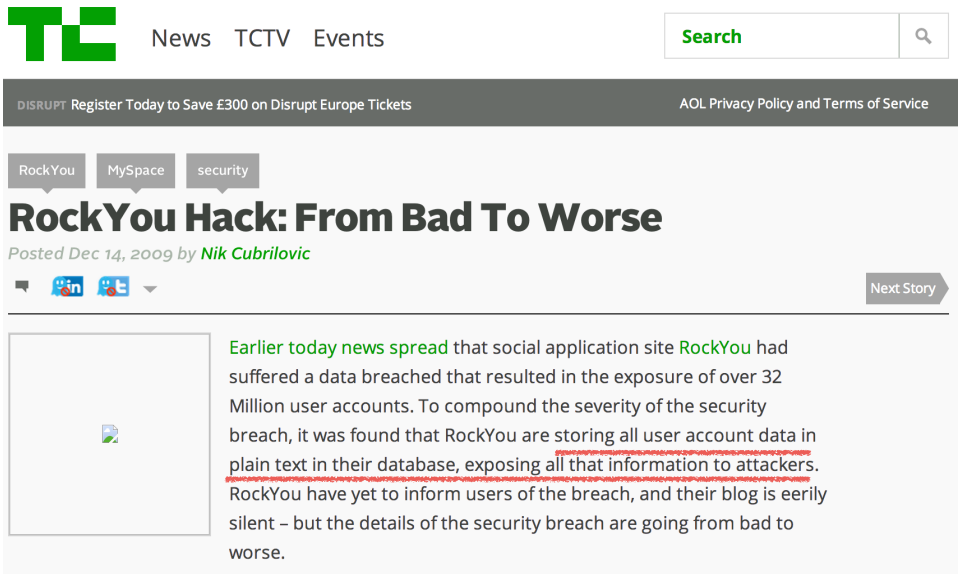
Defense in Depth (DiD)

- **Security by diversity**
  - If one layer is broken, there is another of a materially different character that needs to be bypassed
  - **Categories**: Prevention/Mitigation

- **Example**: Do *all of the following*, not just one
  - Use a firewall for preventing access via non-web ports
  - Encrypt account data at rest
  - Use a safe language for avoiding low-level vulnerabilities

Failure: Authentication Bypass

- **(Poor) passwords can be guessed**
  - bypassing authentication process intent
- **Passwords can be stolen**
  - **Defense in depth**: Should encrypt the password database
    - Assumes that compromise is possible, and thus requires additional defense



DiD: Use community resources

- **Use hardened code**, perhaps from other projects
  - E.g., **crypto libraries**
  - But make sure it meets your needs (*test it*; cf. Heartbleed!)

- **Vet designs publicly**: *No security by obscurity!*

- **Stay up on recent threats and research**
  - **NIST** for **standards**
  - **OWASP**, **CERT**, **Bugtraq** for **vulnerability reports**
  - **SANS** *Newsbites* for **latest top threats**
  - Academic and industry **conferences and journals** for **longer term trends**, **technology**, and **risks**

Failure: Broken Crypto Impl.

Getting crypto right is hard

Use vetted implementations and algorithms

Remote Timing Attacks are Practical

David Brumley
Stanford University
dbrumley@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

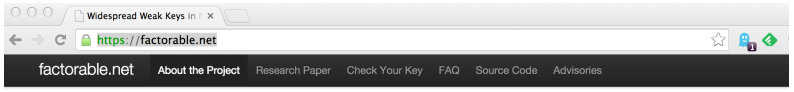
Abstract

Timing attacks are usually used to attack weak computing devices such as smartcards. We show that timing attacks apply to general software systems. Specifically, we devise a timing attack against OpenSSL. Our experiments show that we can extract private keys from an OpenSSL-based web server running on a machine in the local network. Our results demonstrate that timing attacks against network servers are practical and therefore security systems should defend against them.

The attacking machine and the server were in different buildings with three routers and multiple switches between them. With this setup we were able to extract the SSL private key from common SSL applications such as a web server (Apache+mod_SSL) and a SSL-tunnel.

Interprocess. We successfully mounted the attack between two processes running on the same machine. A hosting center that hosts two domains on the same machine might give management access to the admins of each domain. Since both domain are hosted on the same machine, one admin could use the attack to extract the secret key belonging to the

Timing channel *USENIX Security'03*



Widespread Weak Keys in Network Devices

We performed a large-scale study of RSA and DSA cryptographic keys in use on the Internet and discovered that significant numbers of keys are insecure due to insufficient randomness. These keys are being used to secure TLS (HTTPS) and SSH connections for hundreds of thousands of hosts.

- We found that 5.57% of TLS hosts and 9.60% of SSH hosts share public keys in an apparently vulnerable manner, due to either insufficient randomness during key generation or device default keys.
- We were able to remotely obtain the RSA private keys for 0.50% of TLS hosts and 0.03% of SSH hosts because their public keys shared nontrivial common factors due to poor randomness.
- We were able to remotely obtain the DSA private keys for 1.03% of SSH hosts due to repeated signature randomness.

Nearly all the vulnerable hosts are headless and embedded network devices, such as routers, firewalls, and server management cards. These types of devices often generate keys automatically on first boot, and lack many of the physical sources of randomness used by traditional PCs to generate random numbers. We identified apparently vulnerable devices and software from 54 manufacturers and notified these companies about the problems.

In experiments with several popular open-source software components, we were able to reproduce these vulnerabilities and show how such weak keys can arise in practice. Most critically, we found that the Linux random number generator can produce predictable output at boot under certain conditions, although we also observed compromised keys on BSD and Windows-based systems.

Poor randomness *USENIX Security'12*

Monitoring and Traceability

- **If you are attacked, how will you know it?**
  - Once you learn, **how will you discern the cause?**

- Software must be designed to **log relevant operational information**
  - What to log? E.g., events handled, packets processed, requests satisfied, …
  - **Category:** Detection and Recovery

- **Log aggregation**: Correlate activities of multiple applications when diagnosing a breach
  - E.g., `splunk` log aggregator

Quiz 5

- Extensive code reuse (e.g., using Linux on your baby monitor) can actually **help**, by supporting what principle?

A. Small Trusted Computing Base
B. Least Privilege
C. Defense in Depth
D. Compartmentalization

Quiz 5

- Extensive code reuse (e.g., using Linux on your baby monitor) can actually **help**, by supporting what principle?

A. Small Trusted Computing Base
B. Least Privilege
C. Defense in Depth (e.g,. crypto library)
D. Compartmentalization

Case study: VSFTPD

Very Secure FTPD

- **FTP**: File Transfer Protocol
  - More popular before the rise of HTTP, but still in use
  - 90's and 00's: **FTP daemon compromises were frequent and costly**, e.g., in Wu-FTPD, ProFTPd, …

- **Very thoughtful design** aimed to **prevent** and **mitigate security defects**

- But also to **achieve good performance**
  - Written in C

- Written and maintained by Chris Evans since 2002
  - **No security breaches that I know of**

https://security.appspot.com/vsftpd.html

VSFTPD Threat model

- **Clients untrusted, until authenticated**

- Once authenticated, **limited** trust:
  - According to user's **file access control policy**
  - For the files being served FTP (and not others)

- Possible attack goals
  - **Steal** or **corrupt resources** (e.g., files, malware)
  - **Remote code injection**

- Circumstances:
  - **Client attacks server**
  - **Client attacks** another **client**

Defense: Secure Strings

```
struct mystr
{
  char* PRIVATE_HANDS_OFF_p_buf;
  unsigned int PRIVATE_HANDS_OFF_len;
  unsigned int PRIVATE_HANDS_OFF_alloc_bytes;
};
```

Normal (zero-terminated) C string

The actual length (i.e., `strlen(PRIVATE_HANDS_OFF_p_buf)`)

Size of buffer returned by `malloc`

```
void
private_str_alloc_memchunk(struct mystr* p_str, const char* p_src,
                           unsigned int len)
{
  ...
}


void
str_copy(struct mystr* p_dest, const struct mystr* p_src)
{
  private_str_alloc_memchunk(p_dest, p_src->p_buf, p_src->len);
}
```

```
struct mystr
{
  char* p_buf;
  unsigned int len;
  unsigned int alloc_bytes;
};
```

replace uses of `char*` with `struct mystr*`
and uses of `strcpy` with `str_copy`

```
void
private_str_alloc_memchunk(struct mystr* p_str, const char* p_src,
                           unsigned int len)
{
  /* Make sure this will fit in the buffer */
  unsigned int buf_needed;
  if (len + 1 < len)
  {
    bug("integer overflow");
  }
  buf_needed = len + 1;
  if (buf_needed > p_str->alloc_bytes)
  {
    str_free(p_str);
    s_setbuf(p_str, vsf_sysutil_malloc(buf_needed));
    p_str->alloc_bytes = buf_needed;
  }
  vsf_sysutil_memcpy(p_str->p_buf, p_src, len);
  p_str->p_buf[len] = '\0';
  p_str->len = len;
}
```

consider NUL terminator when computing space

allocate space, if needed

copy in p_src contents

```
struct mystr
{
  char* p_buf;
  unsigned int len;
  unsigned int alloc_bytes;
};
```

Copy in at most `len` bytes from `p_src` into `p_str`

Defense: Secure Stdcalls

- Common problem: **error handling**
  - Libraries **assume** that **arguments are well-formed**
  - Clients **assume** that library **calls always succeed**

- Example: `malloc()`
  - What if argument is non-positive?
    - We saw earlier that integer overflows can induce this behavior
    - Leads to buffer overruns
  - What if returned value is NULL?
    - Oftentimes, a deference means a crash
    - On platforms without memory protection, a dereference can cause corruption

fails if it receives malformed argument or runs out of memory

```
void*
vsf_sysutil_malloc(unsigned int size)
{
  void* p_ret;
  /* Paranoia - what if we got an integer overflow/underflow? */
  if (size == 0 || size > INT_MAX)
  {
    bug("zero or big size in vsf_sysutil_malloc");
  }
  p_ret = malloc(size);
  if (p_ret == NULL)
  {
    die("malloc");
  }
  return p_ret;
}
```

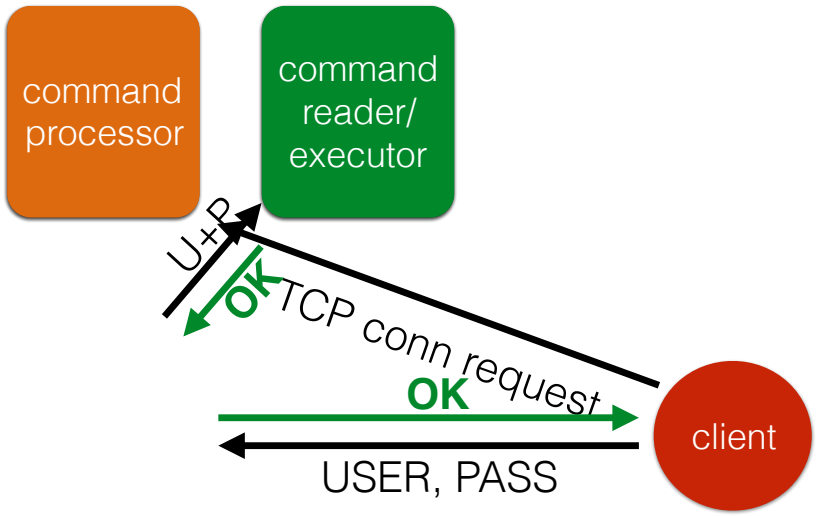
Defense: Minimal Privilege

- **Untrusted input** always handled by **non-`root` process**
  - Uses IPC to delegate high-privilege actions
    - Very little code runs as `root`
- **Reduce privileges** as much as possible
  - Run as particular (unprivileged) user
    - File system access control enforced by OS
  - Use capabilities and/or SecComp on Linux
    - Reduces the system calls a process can make
- **`chroot` to hide all directories** but the current one
  - Keeps visible only those files served by FTP

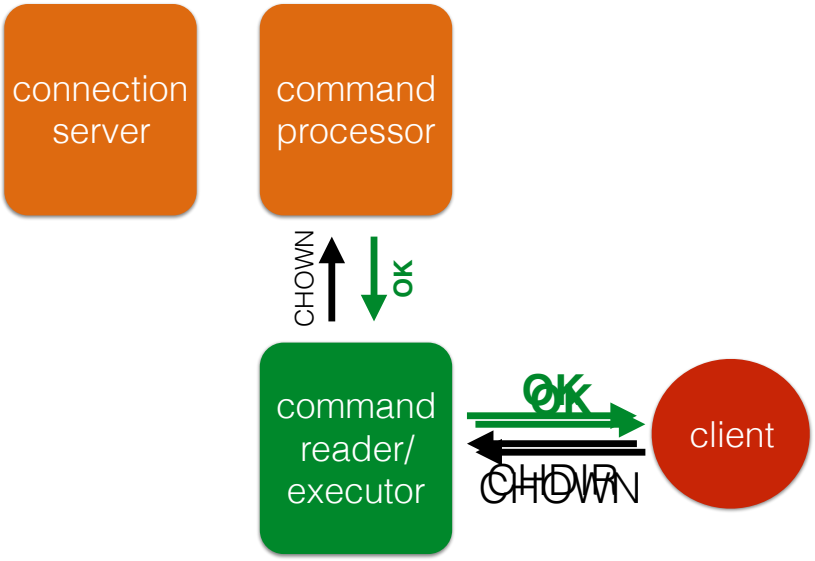
small trusted computing base

principle of least privilege

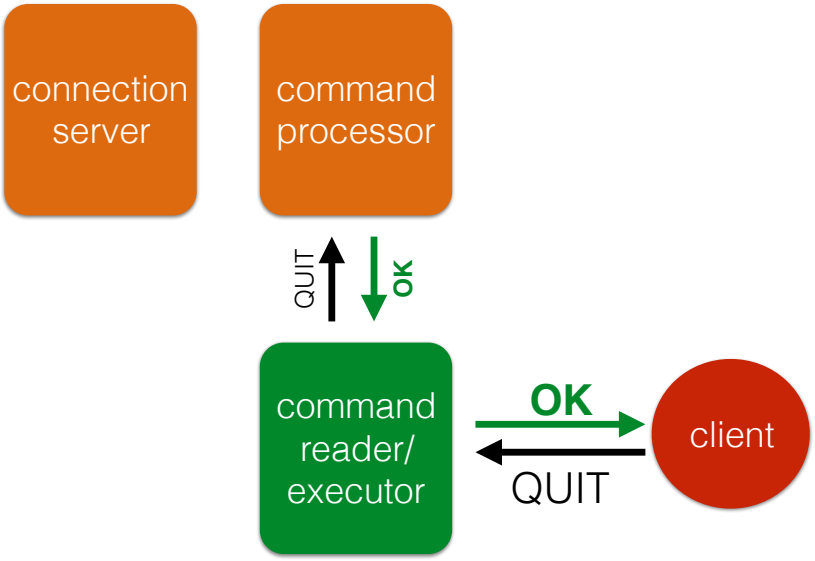
Connection Establishment



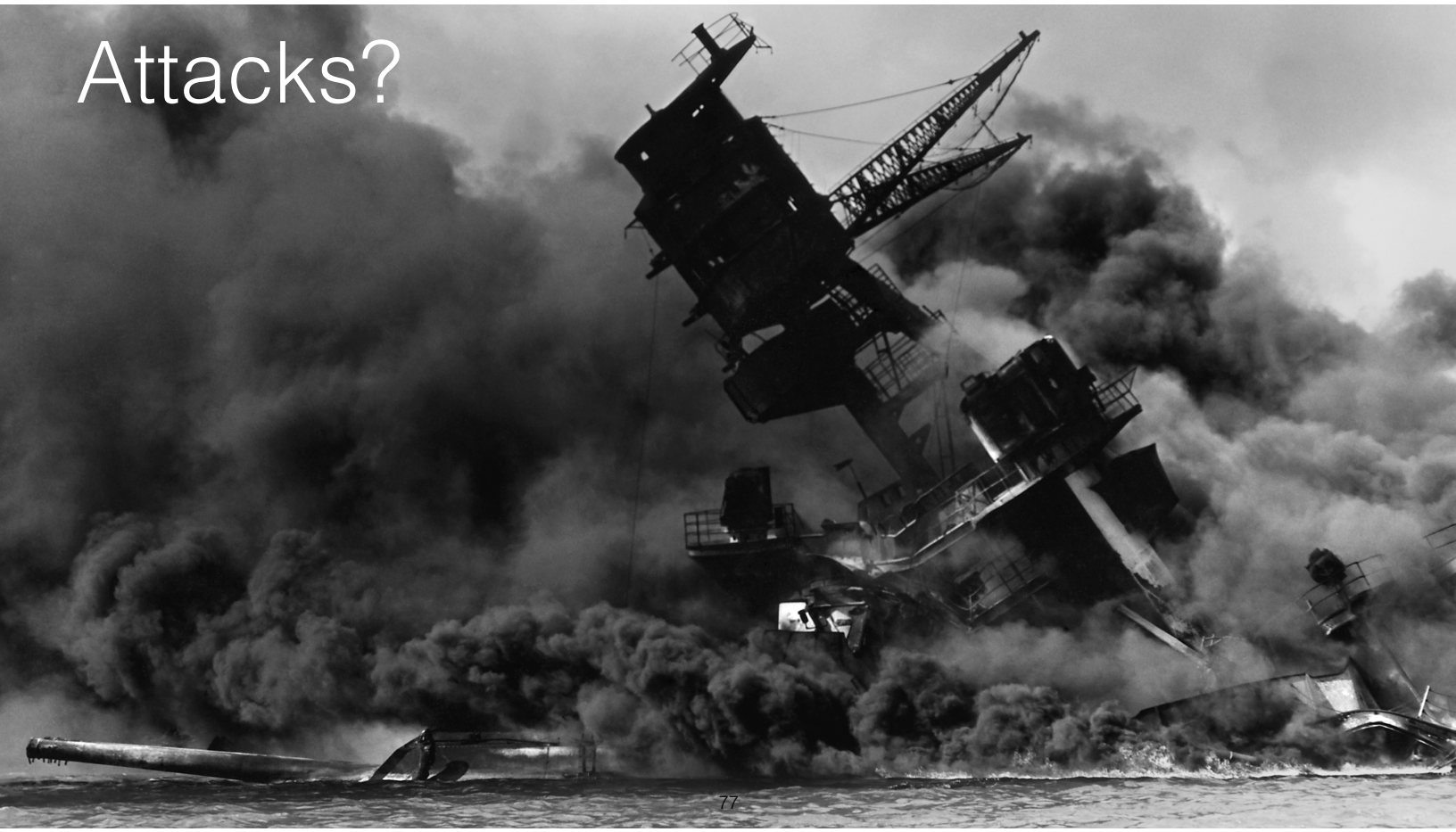
Performing Commands



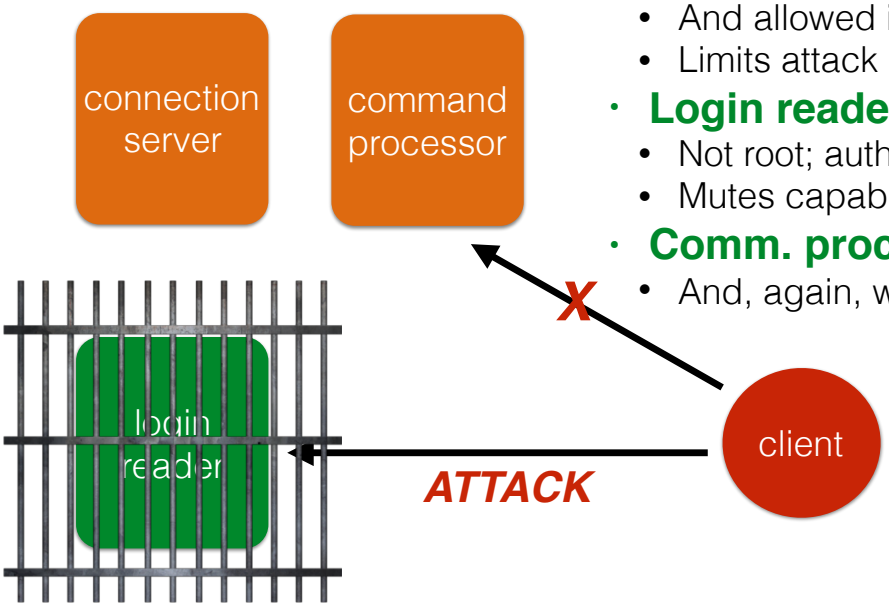
Logging out


connection server
command processor
QUIT
OK
command reader/ executor
OK
QUIT
client

Attacks?

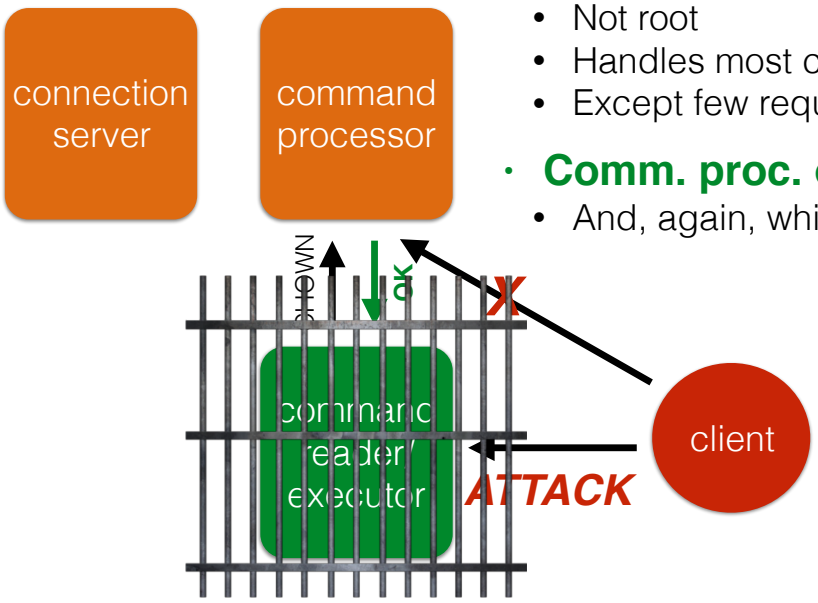


Attack: Login



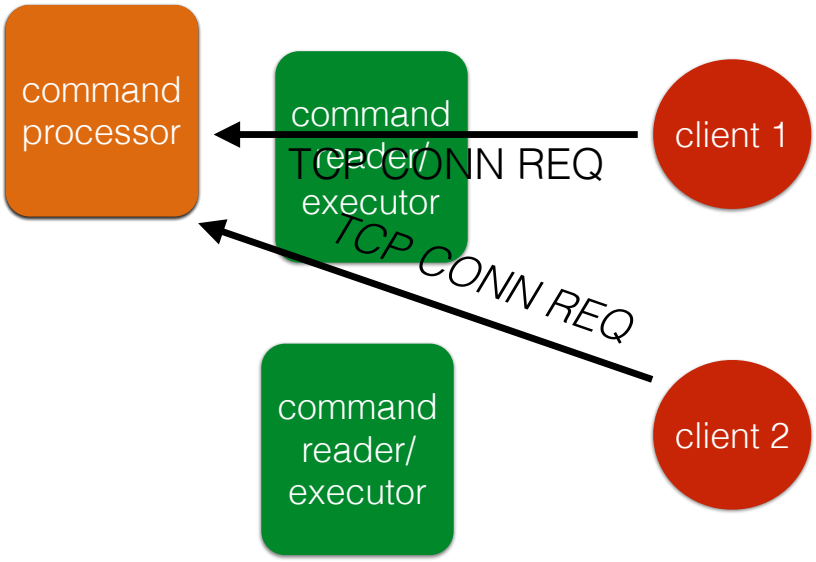
- **Login reader white-lists input**
  - And allowed input very limited
  - Limits attack surface
- **Login reader has limited privilege**
  - Not root; authentication in separate process
  - Mutes capabilities of injected code
- **Comm. proc. only talks to reader**
  - And, again, white-lists its limited input

Attack: Commands

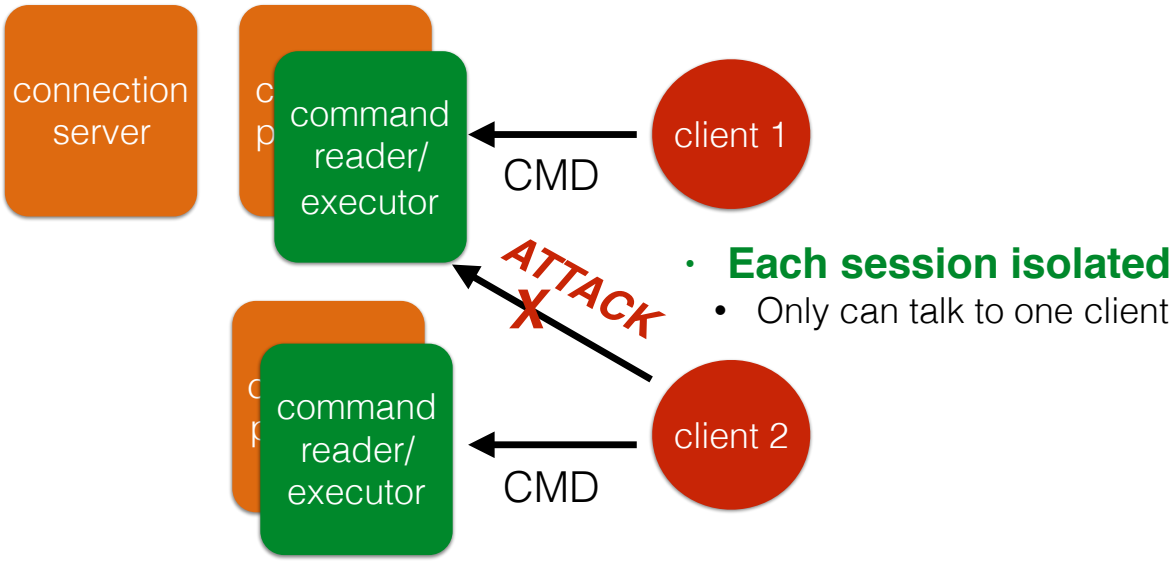


- **Command reader sandboxed**
  - Not root
  - Handles most commands
  - Except few requiring privilege
- **Comm. proc. only talks to reader**
  - And, again, white-lists its limited input

Attack: Cross-session



Attack: Cross-session



Other VSFTPD notables

- **Secure sockets** option, for encrypted connections
  - But **not turned on by default**: "OpenSSL is a massive quantity of code which is essentially parsing complex protocol under the full control of remote malicious clients. SSL / TLS is disabled by default, both at compile time and run time. This forces packagers and administrators to make the decision that they trust the OpenSSL library. I personally haven't yet formed an opinion on whether I consider the OpenSSL code trustworthy."
- **Eschews trusting other executables**
  - Doesn't use `/bin/ls` for directory listings

Conclusion

- **Secure software** requires thinking about security *throughout the development process*
  - Secure **design**
  - Secure **coding**

- Applying **principles** and **rules**
  - To prevent, mitigate, or more quickly become aware of possible harm

- **"Security as an afterthought" probably won't work**
  - Security too much part of design/implementation