CIS 447/544: Computer and Network Security

Anys Bacha

Slides from U. Shankar, M. Hicks, K. Du, D. Boneh, N. Zeldovich, A. Rahmati

Principles for Secure Design

Making Secure Software

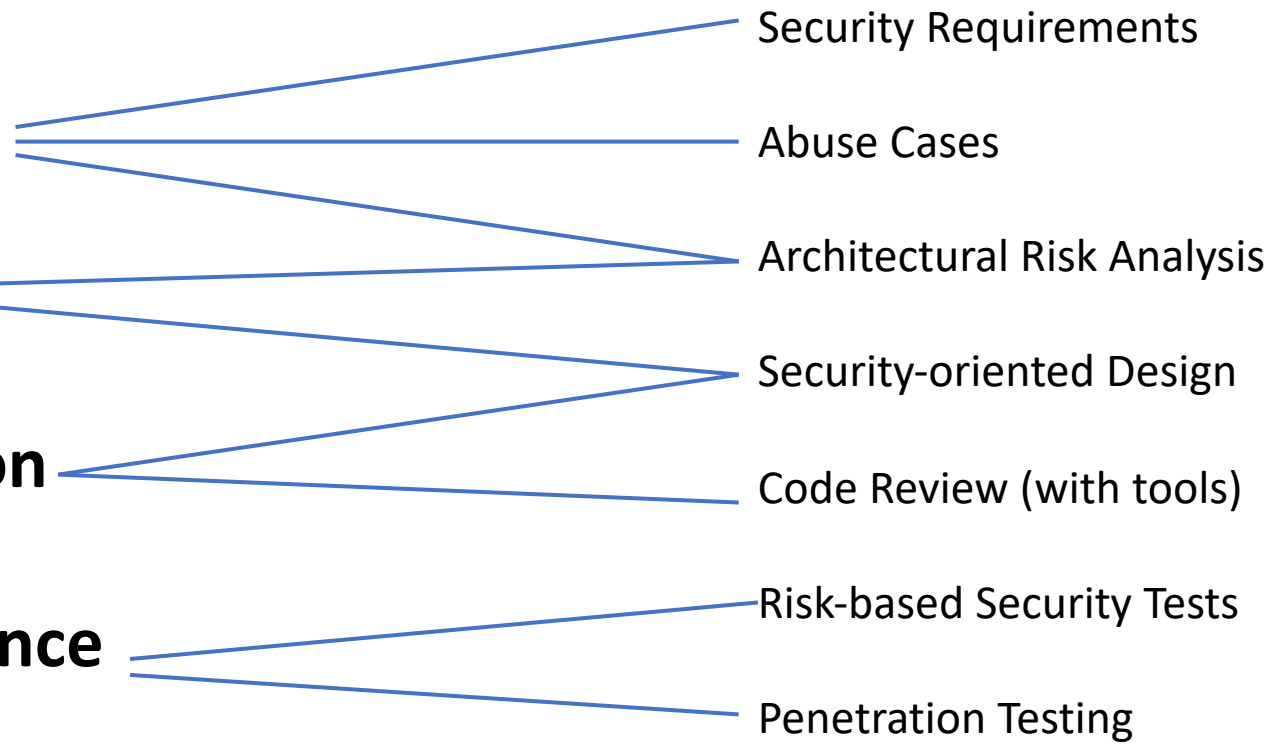
- **Flawed approach**: Design and build software, ignore security at first
  - Add security once the functional requirements are satisfied
- **Better approach**: Build security in from the start
  - Incorporate security-minded thinking into all phases of the development process

Development process

Phases

- **Requirements**
- **Design**
- **Implementation**
- **Testing/assurance**

Security Engineering (Pertains to all phases)



Note that different processes have different phases and artifacts, but all involve the basics above. We'll keep it simple and refer to these

Software vs Hardware

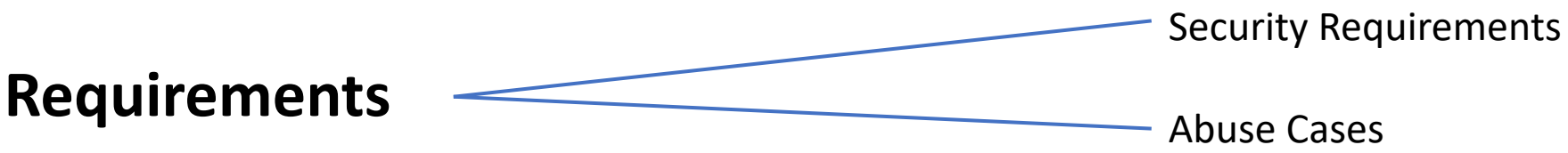
- System design contains ***software and hardware***
  - Mostly, we are focusing on the software

- **Software is malleable** and easily changed
  - Advantageous to core functionality
  - **Harmful to security** (and performance)

- **Hardware is fast**, but hard to change
  - Disadvantageous to evolution
  - **Advantage to security**
    - Can’t be exploited easily, or changed by an attack

Secure Hardware

- Security functionality in hardware
  - Intel's AES-NI implements cryptography instructions
  - Intel SGX: per-process encrypted enclave
    - Protect application data from the OS
- **Hardware primitives for security**
  - **Physically uncloneable functions (PUFs)**
    - Source of unpredictable, but repeatable, randomness, useful for authentication
  - Intel MPX - **primitives for fast memory safety**



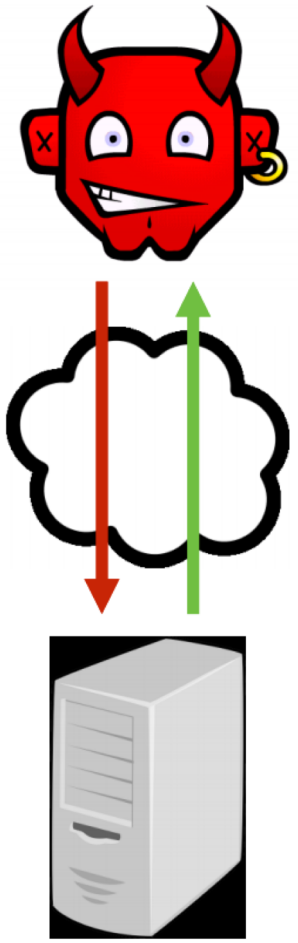
Threat Modeling

Threat Model

- The threat model makes explicit the adversary's assumed powers
  - Must match reality, otherwise risk analysis of the system will be wrong
- **The threat model is critically important**: without the threat model
  - Cannot asses whether your design will repel that attacker
  - Saying "This system is secure" means nothing

Example network threat model: Malicious user

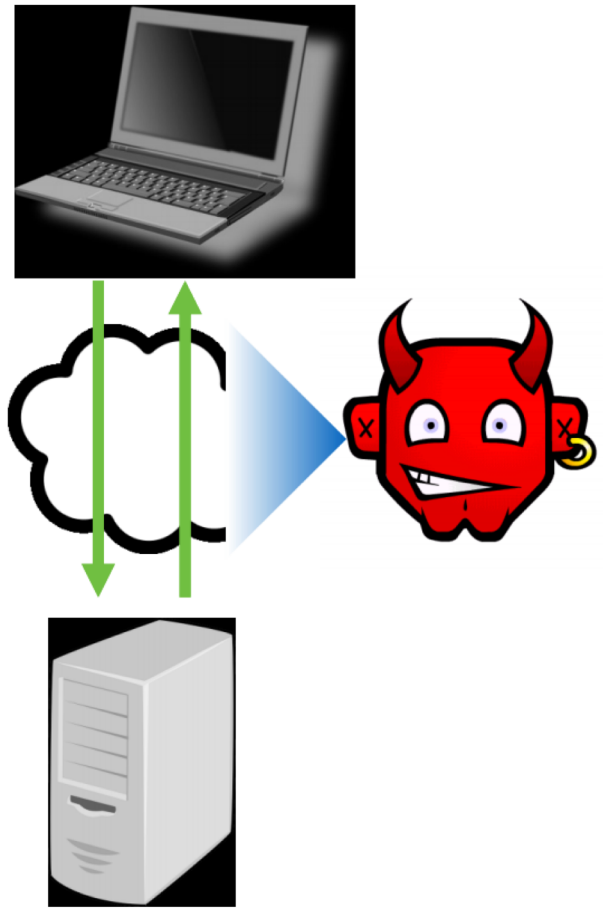
- Is a user who can connect to a service via the network
  - May be anonymous
- Can:
  - **Measure** the size and timing of requests and responses
  - Run **parallel sessions**
  - Provide **malformed** inputs or messages
  - **Drop** or **send extra** messages
- Design: No need to encrypt communications **(what about telnet?)**
- Example attacks a malicious user can perform
  - SQL injection, XSS, CSRF, buffer overrun



Example network threat model: Snooping User

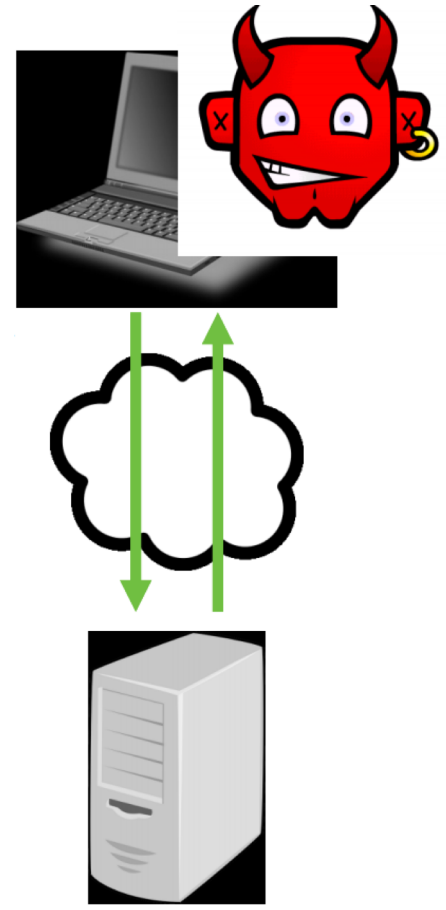
- Attacker on **same network** as other users
  - e.g., Unencrypted Wi-Fi at coffee shop
- Can **read/measure** others' messages
  - **May also intercept, duplicate**, and **modify**

- Design: Use encrypted communications
  - application (SSL), network (IPsec), link (wifi)
- Example attacks a snooping user can conduct:
  - Session hijacking (read session key of a user and re-use it), side-channel attack (violate privacy by reading unencrypted data), DoS



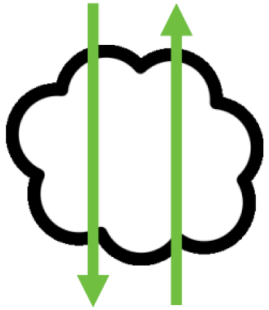
Example network threat model: Co-located user

- Attacker on same machine as other users
  - 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
- Design:
  - Encrypt all stored (sensitive) information
  - Sandbox applications to isolate keylogger
  - Example attacks: Password theft (and other credentials/secrets) through keylogging

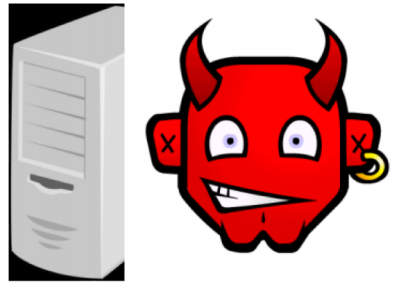


Example network threat model: Compromised server

- The attacker is on the **server machine** (think Cloud/AWS)



- Like an attacker co-located with user BUT WORSE



Bad Model = Bad Security

- The assumptions you make are potential **holes the attacker can exploit**
- E.g.: Assuming there are no snooping users is **no longer valid**
  - wi-fi networks are widespread in most deployments

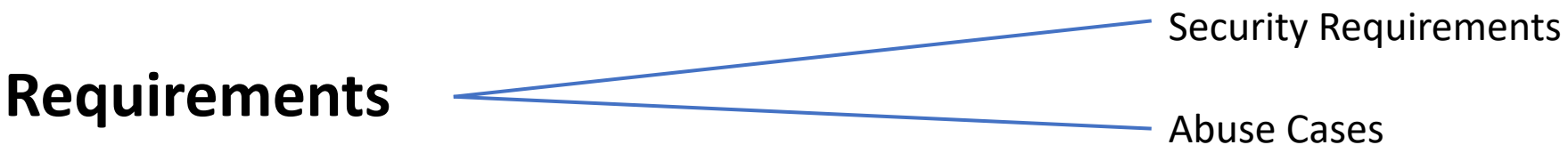
- Other mistaken assumptions
  - **Assumption**: Encrypted traffic carries no information
    - Not true! By analyzing the size and distribution of messages, you can infer application state
  - **Assumption**: Timing channels carry little information
    - Not true! Timing measurements of previous RSA implementations could eventually reveal an SSL secret key

Finding a good model

- One way to find a good model is to compare against similar systems
  - What attacks does their design contend with?
- Understand past attacks
  - How do they apply to your system?

Finding a good model

- It is important to **challenge assumptions** in your design
- Don’t settle for the status quo and ask yourself these questions:
  - What happens if an assumption is false?
    - What would a breach potentially cost you?
  - How hard would it be to get rid of an assumption in order to model a stronger adversary?
    - At the same time, be realistic
      - What would the development cost be?



Security Requirements, Abuse Cases

Security Requirements

- Software requirements are typically about what software should do

- But we also want **security requirements**
  - This is in the form of: Security-related **goals** or **policies**
    - Example: One user's bank account balance should not be learned by, or modified by, another user (unless authorized)

Security Requirements

- **Mechanisms** for enforcing policies:
  - Example:
    - Users identify themselves using passwords
    - passwords are “strong”
    - password databases are only accessible to a login program.

Typical Kinds of Requirements

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

- **Supporting mechanisms include:**
  - **Authentication**
  - **Authorization**
  - **Auditability**

Policy: Confidentiality

- **Confidentiality** implies that sensitive information is not leaked to unauthorized users
  - **An example policy:**
    - A bank account status (including balance) is known only to the account owner

- **Privacy** implies to provide confidentiality for individuals

- **Anonymity** is a special kind of privacy
  - **Example:** Non-account holders should be able to browse the bank site without being tracked (Here the adversary is the bank)

Where do Google and Facebook stand on this?

Policy: Confidentiality

- Example violations could be done **directly or indirectly via side channels**
  - For example:
    - Manipulating the system to directly display Bob's bank balance to Alice
    - Or determining Bob has an account at Bank A according to a shorter delay on login failure (don't exhaust the entire database before returning)

Policy: Integrity

- **Definition:** Sensitive information is **not changed** by unauthorized parties or computations
- **Example**: Only the account owner can authorize withdrawals from her account
- Violations of integrity can also be **direct** or **indirect**
  - **Example**: Withdrawing from an account yourself vs. confusing the system into doing it (think CSRF attack)

Policy: Availability

- **Definition**: A system is **responsive to requests**

- **Example:** A user may want to 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

How to support a mechanism: Authentication

- Who/what is the **subject** of security policies?
  - We need a **notion of identity** and a way to **connect the action** with an identity
  - This is also known as a **principal**

How to support a mechanism: Authentication

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

Supporting mechanism: Authorization

- It is also important to define **when** a principal may perform an action
- **Example:** Bob is authorized to access his own account, but not Alice's account
- **Access-control policies** define what actions might be authorized
  - May be role-based, user-based, etc.

Supporting mechanism: Auditability

- Retain enough information to **determine the circumstances of a breach or misbehavior**
  - Often stored in **log files**
  - Must be **protected from tampering (blockchain?)**,
  - Disallow access that might violate other policies

- **Example**: Every account-related action is logged locally and mirrored at a separate site
  - Only authorized bank employees can view log

Defining Security Requirements

- There are many processes for deciding security requirements

- Example: **General policy concerns**
  - Due to regulations/standards (HIPAA, etc.)

Defining Security Requirements

- Example: **Policy arising from threat modeling**
  - Which **attacks** cause the **greatest concern**?
    - Who are likely attackers, what are their goals and methods?
  - Which **attacks** have **already occurred**?
    - Within the organization, or elsewhere on related systems?

Abuse Cases

- Must be in line with the security requirements
- Must describe what a system **should not do**
- Example **use case**: A system allows bank managers to modify an account's interest rate
- Example **abuse case**: A user can spoof being a manager and modify account interest rates

Defining Abuse Cases

- Use attack patterns and likely scenarios to consider how an **attacker's power** could **violate a security requirement**
  - Based on the threat model you put together
  - What might occur if a security measure was removed?
  - This allows you to build some fault tolerance in your system and strengthen your mechanisms you put in place

Examples of Defining Abuse Cases

- **Example**: A co-located attacker steals a password file and learns all user passwords
  - This is possible if password file is not properly hashed, salted
- **Example**: Snooping where an attacker replays a captured message, effecting a bank withdrawal
  - Possible if messages have no nonce

Security design principles

Design Defects = Flaws

- Recall: Software defects = both flaws and bugs
  - Here we classify
    - **Flaws** as 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

Categories of Principles

- **Prevention:** Eliminate software defects entirely
  - **Example:** Heartbleed bug (buffer overflow) would have been prevented by using a type-safe language, like Java

- **Mitigation**: Reduce harm from exploitation of unknown defects
  - **Example**: Run each browser tab in a separate process, so exploiting one tab does not give access to data in another

- **Detection/Recovery**: Identify, understand an attack; undo damage
  - **Examples**: Monitoring, snapshotting

Principles for building secure systems

The general rule of thumb is that when principles are neglected they result in design flaws

- Security is economics
- Principle of least privilege
- Use fail-safe defaults
- Use separation of responsibility
- Defend in depth
- Account for human factors
- Ensure complete mediation
- Kerkhoff's principle
- Accept that threat models change over time
- If you can't prevent, detect
- Design security from the ground up
- Prefer conservative designs
- Proactively study attacks

“Security is economics”

You can’t afford to secure against everything, so what do you defend against?

Answer: That which has the greatest “return on investment”

THERE ARE NO SECURE SYSTEMS, ONLY DEGREES OF INSECURITY

- In practice, need to **resist a certain level of attack**
  - Example: Safes come with security level ratings
  - “Safe against safecracking tools & 30 min time limit”
- Corollary: Focus energy & time on **weakest link**
- Corollary: Attackers follow the path of least resistance

“Principle of least privilege”

Give a program the access it legitimately needs to do its job.
NOTHING MORE

- **Example**: Unix does a BAD JOB:
  - Every program gets all the privileges of the user who invoked it
  - vim as root: it can do anything -- should just get access to file

- **Example**: Windows JUST AS BAD, MAYBE WORSE
  - Many users run as Administrator,
  - Many tools require running as Administrator

“Principle of least privilege”

Give a program the access it legitimately needs to do its job.
NOTHING MORE

- **Example**: Smartphones:
    - Apps tend to require unnecessary resources and sensors in order to be installed

“Use fail-safe defaults”

Things are going to break. So break safely.

- **Default-deny policies**
  - Start by denying all access
  - Then allow only that which has been explicitly permitted

- **Crash => fail to a secure behavior**
  - Example: firewalls are explicitly setup to forward
  - Failure => packets don’t get through

“Use separation of responsibility”

Split up privilege so no **one** person or program has total power.

- **Example**: US government
  - Checks and balances among different branches

- **Example**: Movie theater
  - One employee sells tickets, and another person tears the tickets

- **Example**: Nuclear weapons…

“Defend in depth”

Use multiple, redundant protections

- Only in the event that all of them have been breached should security be endangered.

- Example: Multi-factor authentication:
  - Some combination of password, image selection, USB dongle, fingerprint, iris scanner,…

- Example: “You can recognize a security guru who is particularly cautious if you see someone wearing both...

…a belt and suspenders

“Defend in depth”

…a belt and suspenders

“Ensure complete mediation”

Make sure your reference monitor sees every access to every object

- Any **access control system** has a resource that it needs to enforce
  - For example: Who is allowed to access a file (specific group, user, etc.)
  - Another example: Who is allowed to post to a message board…
- **Reference Monitor:** The piece of code that checks for permission to access a resource

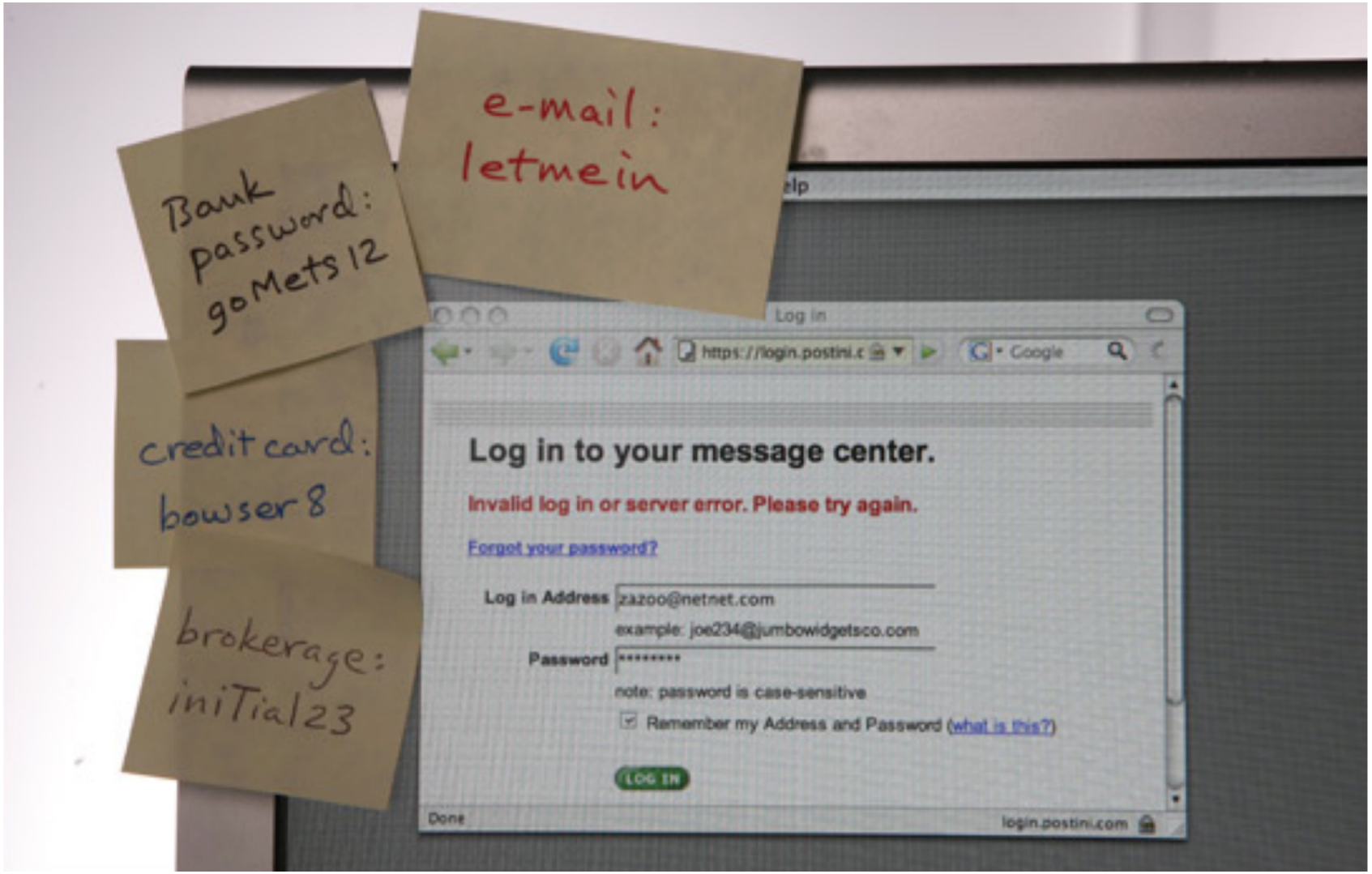
Ensure complete mediation!

“Account for human factors”

(1) “Psychological acceptability”:

In other words, users must buy into the security model

- The security of your system ultimately lies in the hands of those who use it.
- If they don’t believe in the system or the cost it takes to secure it, then they won’t do it.
- **Example**: “All passwords must have 15 characters, 3 numbers, 6 special characters, …”



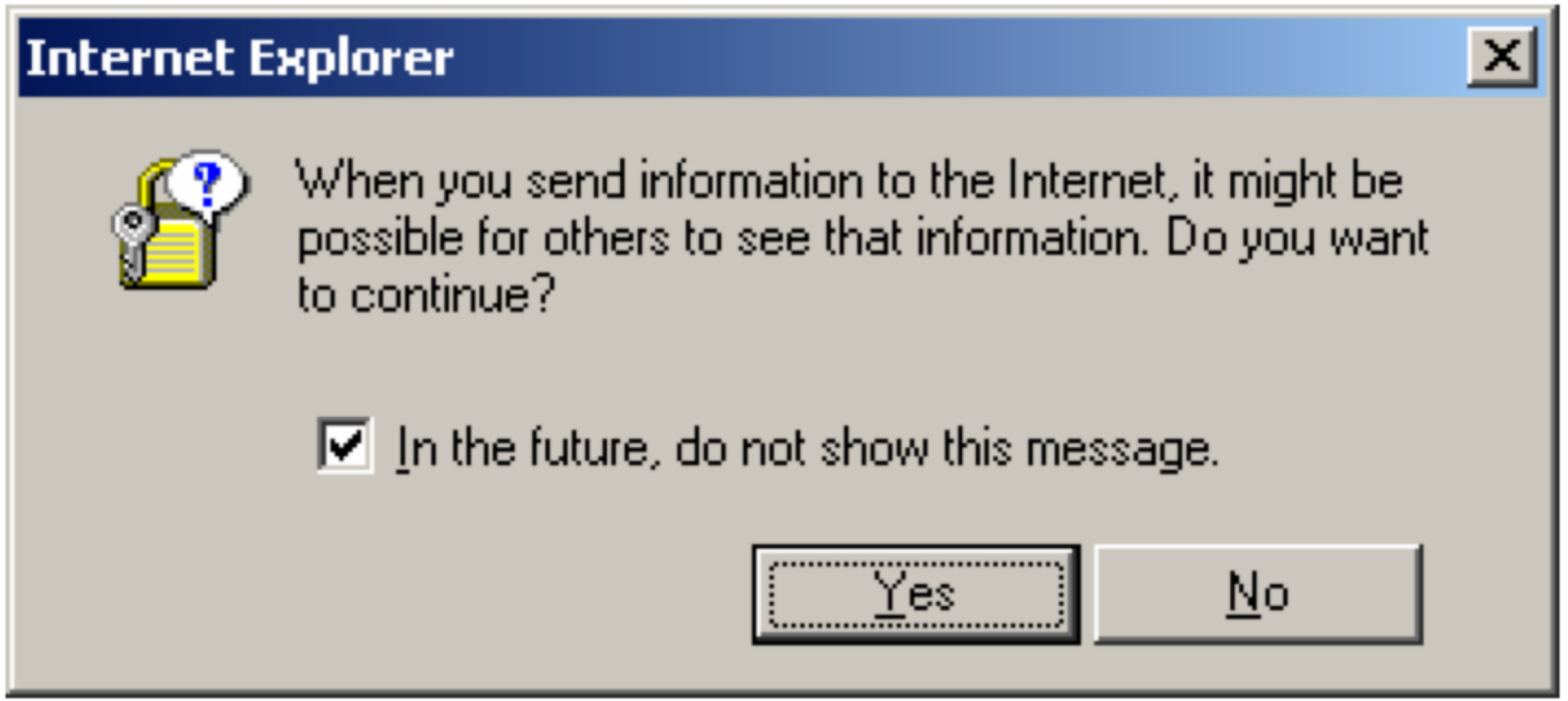
(1) Users must buy into the security

“Account for human factors”

(2) The security system must be usable

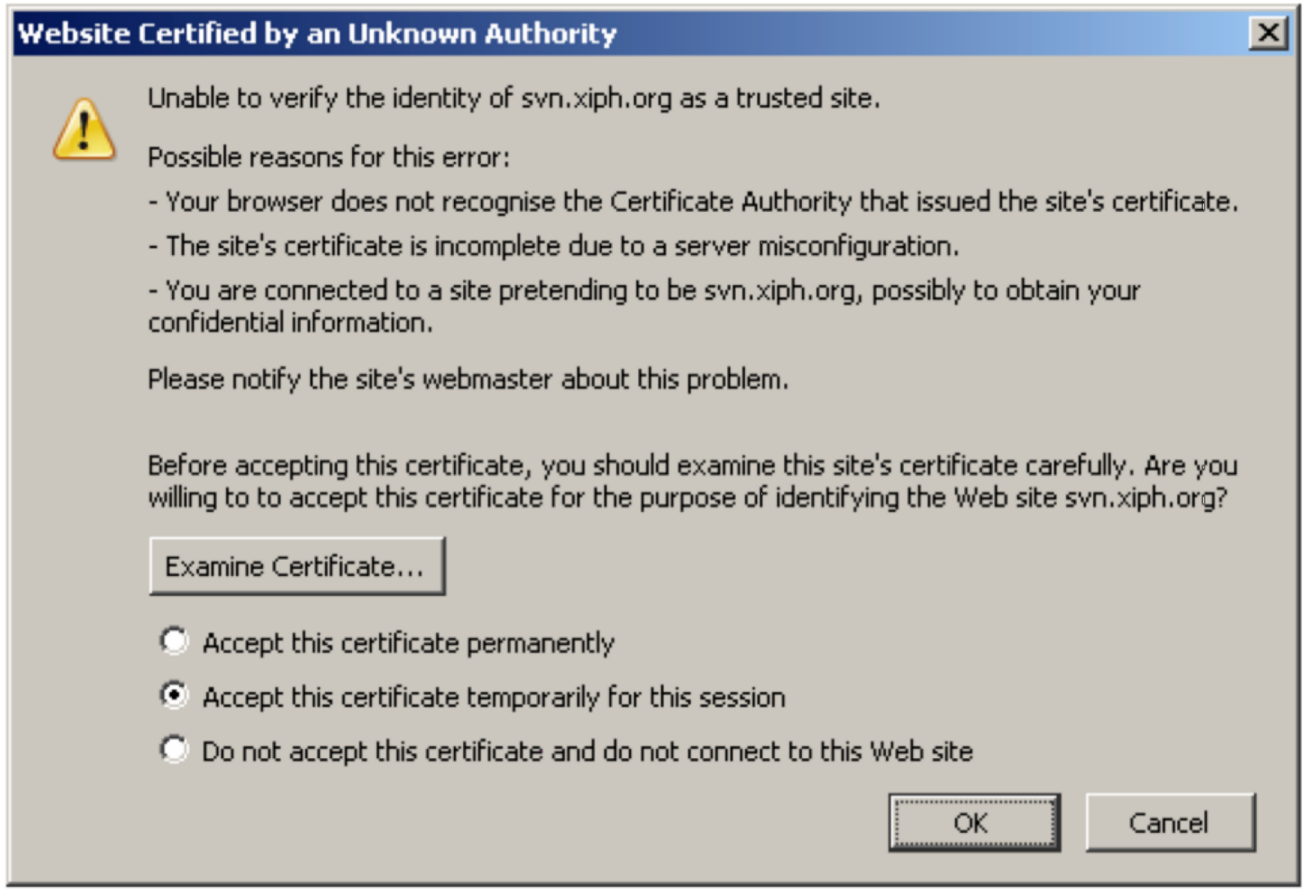
- The security of your system ultimately lies in the hands of those who use it.
- If it is too hard to act in a secure fashion, then they won’t do it.
- **Example:** Popup dialogs

“Account for human factors”



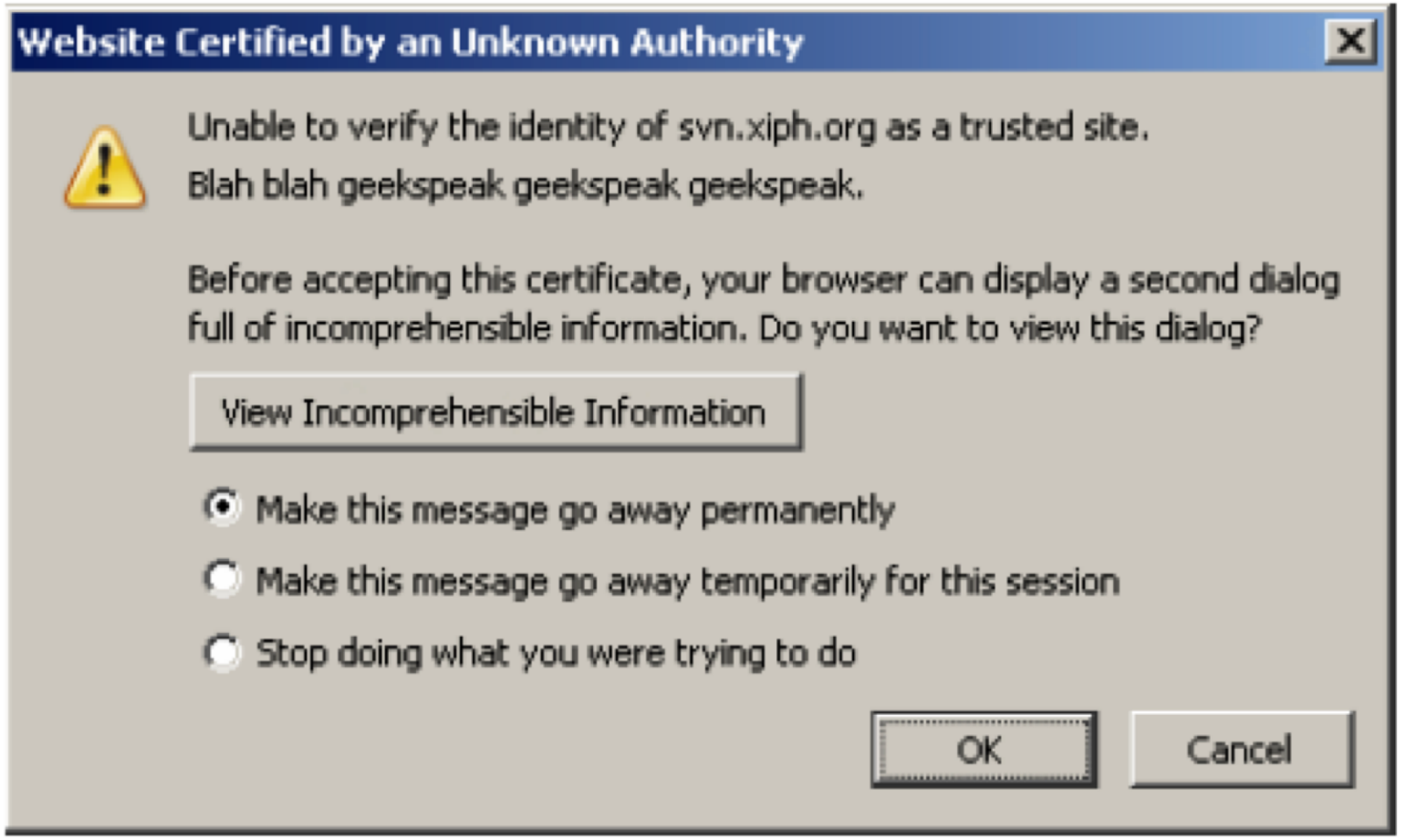
User needs to be onboard

“Account for human factors”



User needs to be onboard

“Account for human factors”



User needs to be onboard

“Kerkhoff’s principle”

Don’t rely on security through obscurity

- Originally defined in the context of crypto systems (encryption, decryption, digital signatures, etc.):
- Crypto systems should remain secure even when an attacker knows all of the internal details
    - It is easier to change a compromised key than to update all code and algorithms

Claude Shannon later redefined this as: “The enemy knows the system”

Kerkhoff's principle??

Kerkhoff's principle!

Principles for building secure systems

Know these well:

- Security is economics
- Principle of least privilege
- Use fail-safe defaults
- Use separation of responsibility
- Defend in depth
- Account for human factors
- Ensure complete mediation
- Kerkhoff's principle

Self-explanatory:

- Accept that threat models change over time
- If you can't prevent, detect
- Design security from the ground up
- Prefer conservative designs
- Proactively study attacks

Trusted computing base and Code safety

Trusted computing bases

Every system has a TCB

- Your reference monitor
- Compiler
- OS
- CPU
- Memory
- Keyboard…..

Security requires that a TCB be

- Correct
- Complete
- Secure

- Two key principles behind a good TCB:
  - KISS
  - Privilege Separation

KISS: Small TCB

- Keep the **TCB small** (and simple) to **reduce overall susceptibility to compromise**
  - The trusted computing base (TCB) consists of the system components that must work correctly in order to ensure security
- **Example**: Operating system kernels
  - Kernels enforce security policies, but are often millions of lines of code
    - A 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

Additional security layers often create vulnerabilities…

October 2010 vulnerability watchlist

Vulnerability Title	Fix Avail?	Date Added
XXXXXXXXXXX XXXXXXXXXXX Local Privilege Escalation Vulnerability	No	8/25/2010
XXXXXXXXXXX XXXXXXXXXXX Denial of Service Vulnerability	Yes	8/24/2010
XXXXXXXXXXX XXXXXXXXXXX Buffer Overflow Vulnerability	No	8/20/2010
XXXXXXXXXXX XXXXXXXXXXX Sanitization Bypass Weakness	No	8/18/2010
XXXXXXXXXXX XXXXXXXXXXX Security Bypass Vulnerability	No	8/17/2010
XXXXXXXXXXX XXXXXXXXXXX Multiple Security Vulnerabilities	Yes	8/16/2010
XXXXXXXXXXX XXXXXXXXXXX Remote Code Execution Vulnerability	No	8/16/2010
XXXXXXXXXXX XXXXXXXXXXX Use-After-Free Memory Corruption Vulnerability	No	8/12/2010
XXXXXXXXXXX XXXXXXXXXXX Remote Code Execution Vulnerability	No	8/10/2010
XXXXXXXXXXX XXXXXXXXXXX Multiple Buffer Overflow Vulnerabilities	No	8
XXXXXXXXXXX XXXXXXXXXXX Stack Buffer Overflow Vulnerability	Yes	8
XXXXXXXXXXX XXXXXXXXXXX Security-Bypass Vulnerability	No	8
XXXXXXXXXXX XXXXXXXXXXX Multiple Security Vulnerabilities	No	8
XXXXXXXXXXX XXXXXXXXXXX Buffer Overflow Vulnerability	No	7/29/2010
XXXXXXXXXXX XXXXXXXXXXX Remote Privilege Escalation Vulnerability	No	7/28/2010
XXXXXXXXXXX XXXXXXXXXXX Cross Site Request Forgery Vulnerability	No	7/26/2010
XXXXXXXXXXX XXXXXXXXXXX Multiple Denial Of Service Vulnerabilities	No	7/22/2010

6 of the vulnerabilities are in security software

Color Code Key: Vendor Replied – Fix in development | Awaiting Vendor Reply/Confirmation | Awaiting CC/S/A use validation

TCB: Privilege Separation

Isolate privileged operations to as small a module as possible

- Don’t give a part of the system more privileges than it needs to do its job (“need to know”)
  - **Principle of least privilege**
- **Example**: Web server daemon
  - Binding to port 80 requires root
  - YOU DON’T want your whole web server running as root!

- **Example**: Email apps often drop you into an editor
  - vi, emacs
  - But these editors often permit dropping you into a shell

Lesson: Trust is Transitive

Isolate privileged operations to as small a module as possible

- **If you trust something, you trust what it trusts**
  - This trust can be misplaced
  - You have to consider the full set of dependencies of the module you're trusting

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

SecComp

- Linux system call enabled since 2.6.12 (2005)
  - An 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
  - **Essentially, SecComp isolates a process by limiting possible interactions**

SecComp

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

Example: 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 form a sandbox (compartmentalize)

Example VSFTPD

- 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 disclosed so far

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)
  - **Remote code injection** (e.g malware)

- Learn more about the VSFTPD design at:

https://security.appspot.com/vsftpd/DESIGN.txt

END