Designing and Building Secure Software

With material from Dave Levin, Mike Hicks
• P1 grading is underway for on-time people
• P2 will be released hopefully Friday
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

• Many development processes; **four common phases:**
  • Requirements
  • Design
  • Implementation
  • Testing/assurance
  • Apply to: whole project, individual components, iterations

• Where does **security engineering** fit in?
  • All phases!
Security engineering

Phases

- Requirements
- Design
- Implementation
- Testing/assurance

Activities

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*
Learn more!

Hardware Security

Part of the "Cybersecurity" Specialization »

In this course, we will study security and trust from the hardware perspective. Upon completing the course, students will understand the vulnerabilities in current digital system design flow and the physical attacks to these systems. They will learn that security starts from hardware design and be familiar with the tools and skills to build secure and trusted hardware.

Gang Qu
Associate Professor
Electrical and Computer Engineering
University of Maryland, College Park
Running Example: Online banking

Bob’s:

Alice’s:

Bob

Alice

✓

✗
Threat Modeling
(Architectural Risk Analysis)
Threat Model

• Make adversary’s assumed powers explicit
  • Must match reality, otherwise risk analysis of the system will be wrong

• The threat model is critically important
  • If you don’t know what the attacker can (and can’t) do, how can you know whether your design will repel that attacker?

• This is part of architectural risk analysis
Example: **Network User**

- Can connect to a service via the network
  - May be anonymous

- Can:
  - **Measure** size, timing of requests, responses
  - Run *parallel sessions*
  - Provide **malformed** inputs or messages
  - **Drop** or **send extra** messages

- **Example attacks**: SQL injection, XSS, CSRF, buffer overrun
Example: **Snooping User**

- Attacker on **same network** as other users
  - e.g., Unencrypted Wi-Fi at coffee shop
- Can **also**
  - **Read/measure** others’ messages
  - **Intercept, duplicate, and modify**
- **Example attacks**: Session hijacking, other data theft, side-channel attack, denial of service
Example: 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
  - Read/write the user’s display (e.g., to spoof)
- Example attacks: Password theft (and other credentials/secrets)
Threat-driven Design

- Different threat models will elicit different responses

- **Network-only attackers** implies message traffic is **safe**
  - No need to encrypt communications
  - This is what `telnet` remote login software assumed

- **Snooping attackers** means **message traffic is visible**
  - So use encrypted wifi (link layer), encrypted network layer (IPsec), or encrypted application layer (SSL)
    - Which is most appropriate for your system?

- **Co-located attacker** can **access local files, memory**
  - Cannot store unencrypted secrets, like passwords
  - Worry about keyloggers as well (2nd factor?)
Bad Model = Bad Security

- **Assumptions** you make are potential **holes the attacker 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 eventually reveal an 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 assumption is false?
    • 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?
Security engineering

Phases
- Requirements
- Design
- Implementation
- Testing/assurance

Activities
- Security Requirements
- Abuse Cases
- Architectural Risk Analysis
- Security-oriented Design
- Code Review (with tools)
- Risk-based Security Tests
- Penetration Testing

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

• Software requirements: typically about what software should do

• We also want 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)
  • Mechanisms for enforcing them
    • Example: Users identify themselves using passwords, passwords are “strong,” password database 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 unauthorized
  • 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 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 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:* Withdraw from the account yourself 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.
Supporting mechanisms

- Leslie Lamport’s **Gold Standard** defines mechanisms provided by a system to enforce its requirements
  - **Authentication**
  - **Authorization**
  - **Audit**

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

• Who/what is the **subject** of security policies?
  • Need *notion of identity* and a way to *connect action with identity*
    • a.k.a. a **principal**

• **How can system tell a user is who she says she is?**
  • What (only) she **knows** (e.g., password)
  • What 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
Authorization

• Defines **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.
Audit

- Retain enough information to determine the circumstances of a breach or misbehavior (or establish one did not occur)
  - Often stored in log files
  - Must be protected from tampering
  - 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

• Many processes for deciding security requirements

• Example: General policy concerns
  • Due to regulations/standards (HIPAA, SOX, etc.)
  • Due organizational values (e.g., valuing privacy)

• 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

• Illustrate security requirements
  • Describe what system \textit{should not do}

• Example \textit{use case}: System allows bank managers to modify an account’s interest rate

• Example \textit{abuse case}: 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
  • 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 properly hashed, salted

• Example: Snooping attacker replays a captured message, effecting a bank withdrawal
  • Possible if messages have no nonce
Design Flaws
Design Defects = Flaws

• Recall: Software defects = 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
Design vs. Implementation?

- Many levels of system design decisions
  - *Highest level*: main actors (processes), interactions, and programming language(s) to use
  - *Next level*: decomposition of an actor into modules/components, identifying the core functionalities and how they work together
  - *Next level*: how to implement data types and functions, e.g., purely functionally, or using parallelism, etc.
- Last two could be implementation or design, or both
  - The distinction is a bit fuzzy
Secure Software Design

Design software architecture with good **principles** and **rules**

**Risk-based analysis** of software architecture's design
Principles and Rules

- **Principle**: high-level design goal, many possible manifestations

- **Rule**: specific practice consistent with sound principles
  - Difference can be fuzzy, just like design vs. implementation
  - Principles often overlap

- Software design phase tends to **focus on principles** for avoiding flaws
Categories of Principles

- **Prevention**: Eliminate software defects entirely
  - **Example**: Heartbleed bug 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
Some Principles

• Favor simplicity
  • Use fail-safe defaults
  • Do not expect expert users

• Trust with reluctance
  • Minimize trusted computing base
  • Grant the least privilege possible; compartmentalize

• Defend in Depth
  • If one fails, maybe the next will succeed
  • Use community resources to stack defenses

• Monitor and trace
The classic reference on principles of secure design is *The Protection of Information in Computer Systems*, by Saltzer and Schroeder (in 1975)

- Economy of Mechanism
- Fail-safe Defaults
- Complete mediation
- Open design
- Psychological acceptability
- Separation of privilege
- Least privilege
- Least common mechanism
- Trust with reluctance

Add monitoring — these were focused on prevention!

Comparing to our list

- Several principles reorganized/renamed
  - *Separation of privilege* has elements of our *compartmentalization, defend in depth*
  - *Open design* is like *use community resources*, but did not anticipate open-source code

- **Monitoring** is added
  - Their focus on prevention of attack, rather than recovery

- “Principle” of *complete mediation* dropped
  - CM not a *design* principle, but a rather an implementation requirement
Principle: Favor Simplicity
Some Principles

- Favor simplicity
  - Use fail-safe defaults
  - Do not expect expert users

- Trust with reluctance

- Defend in Depth

- Monitor and trace
Favor Simplicity

• Keep it **so simple** it is **obviously correct**
• Applies to external interface, internal design, 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

FS: Use fail-safe defaults

- Some **configuration** or **usage choices** affect security
  - Length of cryptographic keys
  - 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 system without creating
  - **Whitelist** valid objects, rather than blacklist invalid ones
    - e.g., don’t render images from unknown sources
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.
Simple Math: It Always Costs Less to Avoid a Breach Than to Suffer One

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 whitelist and 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.

Home Depot was relying primarily on anti-viral software, as required by the PCI DSS regime, but reports say even internal Home Depot security staff knew it was not sufficient. Since no AV software will recognize and stop custom attack code, the attackers were able to load and run malicious software on Home Depot's self service registers.

How Much Was at Risk?

“... whitelisting on servers and single function servers or appliances has proven to cause near zero business or IT administration disruption”
FS: Don’t expect expert users

[Only] Computer scientists and drug dealers have users
—R. David Lankes

- Consider how **goals, habits, abilities** of unsophisticated users affect security
- Avoid security choices whenever possible
  - Does the user actually know more than you about making the choice?
  - Avoid fatigue, habituation
- When choices are necessary
  - Make natural/obvious/default the secure choice
  - Help users explore ramifications of choices
    - E.g., allow admin to explore user view of set access control policy
- Cannot allow security to interfere with the user’s primary task!
• Name sometime recently when you saw a security warning or made a security decision

• What did you do?

• Why?
Passwords

• **Goal:** easy to remember but hard to guess
  • Turns out to be **wrong** in many cases!
    • Hard to guess = Hard to remember!
  • **Compounding problem:** repeated password use

• **Password cracking tools** train on released data to quickly guess common passwords
  • Project Rainbow, [http://project-rainbowcrack.com/](http://project-rainbowcrack.com/)
  • many more …

• **Top 10 worst passwords of 2013:** 123456, password, 12345678, qwerty, abc123, 123456789, 111111, 1234567, iloveyou, adobe123 [from SplashData]
Password Manager

• A password manager (PM) stores a **database of passwords, indexed by site**
  • Encrypted by a **single, master password** chosen (and remembered) by the user, used as a key
  • **PM generates complicated per-site passwords**
    - Hard to guess, hard to remember, but the latter doesn’t matter!

• Benefits
  • Only a single password for user to remember
  • User’s password at any given site is hard to guess
  • Compromise of password at one site does not permit immediate compromise at other sites

• But:
  • Must still **protect** and **remember** strong **master password**
Password Strength Meter

- Gives user feedback on the strength of the password
- Intended to measure guessability
- Research shows that these can work, but the design must be stringent (e.g., forcing unusual characters)
Better together

- Password manager
  - One security decision, not many

- Password meter
  - Users can explore ramifications of various choices by visualizing quality and reasoning of password
  - Do not permit poor choices (or reduce the chances of them) by enforcing a minimum score
Phishing

• **User is tricked** into thinking that a **site** or **e-mail is legitimate**, rather than a **scam**
  • And is then tricked into installing malware or performing other harmful actions

---

Dear valued customer of TrustedBank,

We have received notice that you have recently attempted to withdraw the following amount from your checking account while in another country: $135.25.

If this information is not correct, someone unknown may have access to your account. As a safety measure, please visit our website via the link below to verify your personal information:


Once you have done this, our fraud department will work to resolve this discrepancy. We are happy you have chosen us to do business with.

Thank you,

TrustedBank

Member FDIC © 2005 TrustedBank, Inc.
Phishing

- **Failure:** Site or e-mail not (really) authenticated
  - Internet e-mail and web protocols **not originally designed for remote authentication**
  - Solution is **hard to deploy**
    - Use hard-to-fake notions of identity
    - But which system? How to upgrade
Learn more!

Usable Security

Part of the "Cybersecurity" Specialization »

This course focuses on how to design and build secure systems with a human-centric focus. We will look at basic principles of human-computer interaction, and apply these insights to the design of secure systems with the goal of developing security measures that respect human performance and their goals within a system.

Jennifer Golbeck
Director
Human-Computer Interaction Lab
University of Maryland, College Park
Principle: Trust with Reluctance
Some Principles

• Favor simplicity

• Trust with reluctance
  • Minimize trusted computing base
  • Grant the least privilege possible; compartmentalize

• Defend in Depth

• Monitor and trace
Trust with Reluctance (TwR)

• **Whole-system security** depends on secure operation of **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 TCB **small** (and simple) to **reduce overall susceptibility**
  - Trusted computing base (TCB): components that *must* work correctly to ensure security
  - **Category**: Prevention

- **Example**: Operating system kernels
  - Kernels enforce security policies, often millions of LoC
    - Compromise in a device driver compromises security overall
  - Better: **Minimize size of kernel** to reduce trust
    - Device drivers moved outside of kernel in micro-kernel designs
Failure: Large TCB

Additional security layers often create vulnerabilities...

October 2010 vulnerability watchlist

<table>
<thead>
<tr>
<th>Vulnerability Title</th>
<th>Fix Avail?</th>
<th>Date Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Privilege Escalation Vulnerability</td>
<td>No</td>
<td>8/25/2010</td>
</tr>
<tr>
<td>Denial of Service Vulnerability</td>
<td>Yes</td>
<td>8/24/2010</td>
</tr>
<tr>
<td>Buffer Overflow Vulnerability</td>
<td>No</td>
<td>8/20/2010</td>
</tr>
<tr>
<td>Sanitization Bypass Weakness</td>
<td>No</td>
<td>8/18/2010</td>
</tr>
<tr>
<td>Security Bypass Vulnerability</td>
<td>No</td>
<td>8/17/2010</td>
</tr>
<tr>
<td>Multiple Security Vulnerabilities</td>
<td>Yes</td>
<td>8/16/2010</td>
</tr>
<tr>
<td>Remote Code Execution Vulnerability</td>
<td>No</td>
<td>8/16/2010</td>
</tr>
<tr>
<td>Use-After-Free Memory Corruption Vulnerability</td>
<td>No</td>
<td>8/12/2010</td>
</tr>
<tr>
<td>Multiple Buffer Overflow Vulnerabilities</td>
<td>No</td>
<td>8/10/2010</td>
</tr>
<tr>
<td>Stack Buffer Overflow Vulnerability</td>
<td>Yes</td>
<td>8/9/2010</td>
</tr>
<tr>
<td>Security Bypass Vulnerability</td>
<td>No</td>
<td>8/6/2010</td>
</tr>
<tr>
<td>Multiple Security Vulnerabilities</td>
<td>No</td>
<td>8/5/2010</td>
</tr>
<tr>
<td>Buffer Overflow Vulnerability</td>
<td>No</td>
<td>7/29/2010</td>
</tr>
<tr>
<td>Remote Privilege Escalation Vulnerability</td>
<td>No</td>
<td>7/28/2010</td>
</tr>
<tr>
<td>Cross Site Request Forgery Vulnerability</td>
<td>No</td>
<td>7/26/2010</td>
</tr>
<tr>
<td>Multiple Denial Of Service Vulnerabilities</td>
<td>No</td>
<td>7/22/2010</td>
</tr>
</tbody>
</table>

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

6 of the vulnerabilities are in security software

TwR: Least Privilege

• Don’t give any piece 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