Designing and Building Secure Software

With material from Dave Levin, Mike Hicks, Adam Shostack
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
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
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

- Security Requirements
- Abuse Cases
- Threat Modeling
- Security-oriented Design
- Code Review (with tools)
- Risk-based Security Tests
- Penetration Testing
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
Confidentiality (and privacy)

- *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., password and one-time-use code
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 (more later)
  • 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 *should not do*

• Example *use case*: System allows bank managers to modify an account’s interest rate

• Example *abuse case*: User can spoof being a manager and modify account interest rates
Threat Modeling
What is a threat model?

- Structured way of analyzing possible threats/vulns
- What is important to protect?
- What could go wrong?
- What capabilities might an attacker have?
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?
Approaches to threat modeling

- Focus on assets
- Focus on attackers
- Focus on engineering/system components
Focus on assets

• Pro: Prioritize what is important, valuable

• Con: Define asset?
  • What you value? What an attacker values?

• Example: Center of Gravity theory
Focus on attackers

• Pro: Make attacker’s powers explicit
  • Helps identify assumptions

• Pro: Focused on threats

• Con: Do you know everything the attacker knows?
  • Get it wrong, whole model falls down

• Example: Persona Non Grata, attack trees
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 attacker 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?)
Focus on components

- Break system into components to analyze
- Pro: Can be comprehensive, checklist
- Con: Hard to do before you have a design
- Con: Hard to prioritize

- Example: Microsoft STRIDE
• Spoofing identity
• Tampering with data
• Repudiation
• Information disclosure
• Denial of service
• Elevation of privilege
Applying STRIDE

- Break system up into components / model
  - e.g., data flow diagrams
- Go through STRIDE list for each component independently
- Identify threats: who, what, why, how
  - Level of impact
Exercise: Threat Model

- Consider a mobile payments app
  - My phone, tied to my bank account / credit card
  - Send / receive money from contacts

- Work up a (partial) threat model with STRIDE
  - Key components: app, central server, network …
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
Now that we’ve **identified** threats …

What do we do about them?
• Prevent it
• Mitigate it
• Accept it?
• Transfer the risk?
Prevent

- Remove the entire threat
  - Get rid of functionality that has risk?
Mitigate

• Limit effectiveness of attacks
• e.g., tampering: prevent via crypto integrity
• Many standard approaches

• (more on prevent, mitigate later)
<table>
<thead>
<tr>
<th>Threat</th>
<th>Mitigation examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spoofing</td>
<td>Authentication</td>
</tr>
<tr>
<td>Tampering</td>
<td>Integrity, authorization</td>
</tr>
<tr>
<td>Repudiation</td>
<td>Logging, signatures</td>
</tr>
<tr>
<td>Info. Disclosure</td>
<td>Authorization, encryption</td>
</tr>
<tr>
<td>Denial of Service</td>
<td>Availability</td>
</tr>
<tr>
<td>Elevation of Priv.</td>
<td>Authorization, isolation</td>
</tr>
</tbody>
</table>
Accept, transfer

- Organization can accept own risk
  - Don’t “accept” risk for your users/customers?

- Transfer via explicit user acceptance?
  - User interface, license agreement?