Lecture 19: Software Security

1. Introduction: Why Security Matters

- Security must be considered from the very beginning of software design—not as an afterthought.
- Integrate security into initial specs, requirements, and all stages of development.
- Clients/customers assume developers are building secure systems, even if they don't specify requirements.
- Ignoring security leads to wasted time, insecure architectures, and potentially catastrophic vulnerabilities.
- Security is a major field in practical computing (jobs, internships, real-world impact), second only to AI in scope.
- Even if you don't specialize in security, you must understand the basics to avoid critical mistakes and communicate with specialists.
- **Regulatory/Compliance:** Many industries require security by law (e.g., GDPR for privacy in Europe, HIPAA for healthcare in the US).
- **Real-World Example:**
  - In 2017, the Equifax breach exposed sensitive data of 147 million people due to an unpatched vulnerability, leading to billions in damages and loss of trust.

---

2. Step Zero: Security Mindset

Before you write code, develop two models:

2.1 Security Model

- What are you defending? ("Crown jewels"/assets)
- Model the application and its valuable data/resources.
- **CIA Triad:**
  - **Confidentiality (Privacy):** Prevent unauthorized data access/leakage.
  - **Integrity:** Prevent unauthorized modification/tampering.
  - **Availability (Service):** Ensure systems are usable and accessible.

```
+-------------------+
|   Application     |
|-------------------|
| Confidential Data |
+-------------------+
      |   |   |
      v   v   v
 C (no leaks)  I (no tampering)  A (always up)
```

Example Security Model: Online Banking

- Assets: Account balances, transaction history, personal info
- Confidentiality: Only account owner and authorized staff can view balances
- Integrity: Only valid transactions can modify balances
- Availability: Users must be able to access accounts 24/7

2.2 Threat Model

- Who might attack the system, and how?
- Prioritize defense strategies based on realistic threats.
- Models are never perfect—improve them continuously.

Threat Modeling Frameworks

Framework	Focus	Example Use
STRIDE	Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege	Web apps, APIs
DREAD	Damage, Reproducibility, Exploitability, Affected Users, Discoverability	Risk ranking
PASTA	Process for Attack Simulation and Threat Analysis	Enterprise systems

Checklist for Security Modeling

Category	Purpose	Example
Assets	What needs protection	Database, config files, credentials
Vulnerabilities	Weaknesses/channels for attack	Open ports, input flaws, misconfig

Category	Purpose	Example
Threats	Potential attackers/scenarios	Ex-employees, bots, phishing, insiders

3. Key Security Functions

3.1 Authentication

- Verifies user identity.
- Examples: Passwords, two-factor authentication (2FA), USB keys, biometrics (retinal scan, fingerprint).
- Multi-factor authentication (MFA):
  - Something you know (password)
  - Something you have (device, app, USB key)
  - Something you are (biometric)
- **Password Best Practices:**
  - Use long, random passwords (consider passphrases)
  - Never reuse passwords across sites
  - Use a password manager
  - Avoid common pitfalls: security questions, SMS-based 2FA (can be intercepted)

Method Type	Description	Example
Something you know	Knowledge-based	Password, PIN
Something you have	Possession-based	USB key, Duo app, TOTP
Something you are	Biometric	Fingerprint, retina

3.2 Authorization

- Dictates what authenticated users can do.
- Example: Instructor can modify grades in their own classes, not others.
- Uses Access Control Lists (ACLs):

User	Read	Write	Execute
Eggert	Yes	Yes	Yes
Milstein	Yes	No	No
Frank	No	No	No

- **Role-Based Access Control (RBAC):**
  - Users are assigned roles (e.g., admin, user, guest), and roles have permissions.
  - Easier to manage than per-user ACLs in large systems.

Role	Permissions
Admin	Read, Write, Execute
User	Read, Write
Guest	Read

- **Principle of Least Privilege:**
  - Users/processes should have the minimum access necessary to perform their tasks.
- **Fail-Safe Defaults:**
  - Deny access by default; only allow what is explicitly permitted.

3.3 Integrity Mechanisms

- Detect and recover from unauthorized changes.
- Examples: Checksums, secure backups, digital signatures.

Mechanism	Description
Checksums	Detect tampering by comparing stored values
Backups	Restore data after compromise/loss
Digital Sig.	Verify authenticity and integrity

3.4 Auditing

- Logs user actions to detect/recover from intrusions.
- Logs must be secure, complete, and regularly reviewed.
- **What to Log:**
  - Logins (success/failure), privilege changes, data access, configuration changes, errors.
  - Protect logs from tampering (write-once, append-only storage).

3.5 Supporting Principles

Principle	Description
Correctness	Security must not break core functionality
Efficiency	Security must not overly degrade performance
Usability	Security should not make the system unusable

---

4. Threat Modeling and Classification

- Classify threats to prioritize and determine effective defenses.

4.1 Network Attacks

Threat	Description	Example
Phishing	Tricking users to click links or enter credentials	Fake bank email asks for password
Drive-by Downloads	Malware triggered by visiting malicious pages	Malicious ad installs spyware
Denial-of-Service	Overloading servers to make them unavailable	Botnet floods login page
Buffer Overruns	Overflowing memory buffers to hijack control flow	See code below
Cross-Site Scripting	Malicious JavaScript in user's browser (e.g., bank theft)	Attacker injects <script> tag
Prototype Pollution	Attacking object prototypes in JavaScript	Overwriting **proto** in JS

Buffer Overrun Example (C):

```c
char buf[8];
gets(buf); // No bounds checking! Attacker can overwrite memory.
```

4.2 Device Attacks

Attack	Description	Example
Bad USB	USB device boots/installs malware, hijacks boot process	Malicious USB stick
Insider Attack	Authorized users misuse privileges	Employee steals data
Supply Chain	Compromise occurs in hardware/software before delivery	Pre-installed malware on device

4.3 Social Engineering

- Convincing users to reveal info or grant access (e.g., fake repairman, lost ID).
- Hard to defend; requires robust integrity and auditing.
- **Example:** Attacker calls pretending to be IT and asks for your password.

4.4 Physical Security

- Physical access can defeat most software security.
- Examples: Stolen laptops, unlocked server rooms, dumpster diving for sensitive documents.

---

5. OWASP Top 10 Application Security Risks (2021)

#	Vulnerability	Description/Examples
1	Broken Access Control	URL/JWT/cookie manipulation, insecure direct object references
2	Cryptographic Failures	HTTP over HTTPS, weak crypto, improper certificate validation
3	Injection Attacks	SQL/NoSQL/command injection via untrusted input

#	Vulnerability	Description/Examples
4	Insecure Design	No threat modeling, poor design practices
5	Security Misconfiguration	Default passwords, open ports, unnecessary services
6	Vulnerable/Outdated Components	Using libraries/OS with known exploits
7	Identification/Authentication Failures	Weak passwords, no rate limiting, brute force attacks
8	Software/Data Integrity Failures	Poor update mechanisms, unverified sources, supply chain attacks
9	Security Logging/Monitoring Failures	Lack of logging, logs not reviewed, filled logs
10	Server-Side Request Forgery (SSRF)	Tricking server to access internal/private network addresses

Examples:

- **1. Broken Access Control:** User changes their user ID in the URL to access another user's data.
- **2. Cryptographic Failures:** Site uses HTTP instead of HTTPS, exposing passwords in transit.
- **3. Injection Attacks:** Attacker enters `'; DROP TABLE users;--` in a login form.
- **4. Insecure Design:** No input validation, no threat modeling, no secure defaults.
- **5. Security Misconfiguration:** Admin interface left open to the internet with default password.
- **6. Vulnerable/Outdated Components:** Using an old version of OpenSSL with known bugs.
- **7. Identification/Authentication Failures:** No account lockout after repeated failed logins.
- **8. Software/Data Integrity Failures:** Application updates from untrusted sources.
- **9. Security Logging/Monitoring Failures:** No alert when admin logs in from a new country.
- **10. SSRF:** Attacker tricks server into fetching internal metadata from AWS.

Clarification:

- **Broken Access Control** is about users being able to access things they shouldn't (authorization failure).
- **Identification/Authentication Failures** are about not being able to reliably tell who a user is (authentication failure).

---

6. Security Testing: Philosophy and Strategies

6.1 Testing Philosophy

Traditional Testing	Security Testing
Inputs: typical users	Inputs: malicious attackers
Failures: random	Failures: deliberate, targeted
Bugs: accidental	Bugs: systematically exploited

6.2 Strategies

- **Static Analysis:** Analyze code without running it (find buffer overflows, races, etc.).
- **Dynamic Analysis:** Analyze code while running (finds runtime issues, e.g., memory leaks, race conditions).
- **Penetration Testing:** Hire trusted "black hats" to simulate real-world attacks.
- **Fuzz Testing:** Automatically generate random/malformed inputs to find crashes and vulnerabilities.

Testing Type	Description	Example Tool
Static Analysis	Examines code without running it	Coverity
Dynamic Analysis	Examines code during execution	Valgrind
Penetration Test	Simulated attack by security experts	Metasploit
Fuzz Testing	Randomized input to find bugs	AFL, libFuzzer

6.3 Side-Channel and Timing Attacks

- Attackers infer internal state/data by measuring timing (e.g., cache timing, Spectre, Meltdown).
- Apple restricts high-res timers to inhibit these attacks; Linux allows nanosecond timing (riskier).
- **Example: Timing Attack on Password Check**

```
// Vulnerable password check
for (int i = 0; i < N; i++) {
    if (input[i] != secret[i]) return 0;
}
```

```
return 1;
// Leaks how many bytes are correct via timing
```

- **Countermeasures:**
    - Use constant-time comparison functions
    - Limit timing information available to untrusted code

6.4 Subtle Abstraction Violations

- Example: Partial string comparison leaks password byte-by-byte via timing.
- Advanced: Manipulate memory layout, exploit page boundaries, cache access patterns.

```
[User Input] ---> [Password Check]
                   |
                   +---> [Timing Difference] ---> [Attacker infers secret]
```

---

7. Trusting Trust: Ken Thompson's Attack

- Described in the Turing Award lecture "Reflections on Trusting Trust."
- Modify the C compiler to insert a backdoor when compiling login.c:

```
if (strcmp(name, "ken") == 0) return true;
```

- Then, modify the compiler to insert this backdoor into any future compiler it compiles.
- Result: Even if you inspect the source code for login.c and cc.c, the executables will still regenerate the backdoor.

```
[Source: login.c] --(buggy cc)--> [login (backdoor)]
[Source: cc.c]    --(buggy cc)--> [cc (backdoor)]
[Source: new cc.c] --(backdoored cc)--> [new cc (backdoor)]
```

- **Modern Implications:**
    - Supply chain attacks in open source: malicious code injected into dependencies, compilers, or build tools.
    - Example: SolarWinds attack (2020) where attackers compromised the build system to insert backdoors.
- **Defense:**
    - Define a **Trusted Computing Base (TCB):** Minimal set of components (compiler, OS, hardware) that must be trusted.
    - Software reproducibility and rigorous review help, but trust must start somewhere.
    - Use reproducible builds and independent verification.

---

8. Summary and Takeaways

- Security must be integrated from the start—never "added later."
- Develop both a security model (what to protect) and a threat model (who/what to defend against).
- The CIA Triad (Confidentiality, Integrity, Availability) is foundational.
- Key functions: authentication, authorization, integrity, auditing, correctness, efficiency.
- Threats include network, device, social engineering, and insider attacks.
- OWASP Top 10 highlights common vulnerabilities—know and avoid them.
- Security testing is adversarial: expect intelligent, targeted attacks.
- Advanced attacks (timing, side-channel, supply chain, trusting trust) require deep awareness.
- Always think like a defender—and sometimes like an attacker—to build robust, secure software.

Checklist for Secure Software Development:

- [ ] Define assets and security requirements
- [ ] Build a security and threat model
- [ ] Apply the principle of least privilege
- [ ] Use secure defaults (fail-safe)
- [ ] Validate and sanitize all inputs
- [ ] Use strong authentication and authorization
- [ ] Encrypt sensitive data in transit and at rest
- [ ] Keep software and dependencies up to date

- [ ] Log and monitor security-relevant events
- [ ] Regularly test for vulnerabilities (static, dynamic, fuzz, pen testing)
- [ ] Plan for incident response and recovery