Security by design & defensive programming (Lecture 9)

Prof. Mathy Vanhoef

DistriNet – KU Leuven – Belgium

Agenda

- Security by design: secure design principles
- Defensive programming
- Security mechanisms
- (Assessing the Secure Development Lifecycle: SAMM)

Security by design

What is security by design?

“Security by design” can have two interpretations:

1. **Software design**, where security is explicitly considered
   - E.g., security requirements / patterns / principles
2. Framework to cover the **whole development lifecycle**
   - A synonym for “Security Development Lifecycle”

Today we use the 1st interpretation

Sources:
- Waidner, M., Backes, M., & Müller-Quade, J. (Ed4s.). (2014). *Development of secure software with security by design*. Fraunhofer-Verlag.
- Kreitz, M. (2019). Security by design in software engineering. *ACM SIGSOFT Software Engineering Notes*, *44*(3), 23-23.

Secure design principles

There are *many* secure design principles:

- Secure design principles by Saltzer and Schroeder
- Principles listed by US CISA and the Microsoft DSL
- OWASP overview and overview by Cavoukian and Dixon

We cover the most common ones

- Note: you don't need to know *where* each principles was listed. You *do* need to know all principles and their meaning.

- OWASP: https://wiki.owasp.org/index.php/Security_by_Design_Principles
- Cavoukian and Dixon: "Privacy and Security by Design: An Enterprise Architecture Approach" (2013)

Selected principles of Saltzer & Schroeder:

- **Economy of mechanism**: keep the system simple
- **Complete mediation**: every access to every object must be checked for authorization.
- **Open design**: assume the code is public or will be reverse engineered. Security should depend on keys or passwords.
- **Least common mechanism**: don’t share mechanisms to access resources, can lead to side-channels.
- **Psychological acceptability**: security mechanisms shouldn’t hinder the usability of accessibility of resources.

Selected principles of Saltzer & Schroeder:

Separation of privilege: separate users and programs based on different levels of trust, needs, and privilege requirements:

- **Segmentation of user privileges** across accounts.
  - E.g., account for privileged operations (managing servers, installing software) and another for normal activities (sending email, browsing).
- **Split privileges across different programs** or components
  - E.g., a low-privileged component parses incoming packets, and a higher-privileged component processes the parsed packets.

Additional principles listed by US CISA

- **Secure the weakest link**: bad people will attack the weakest part of a system. Spend your security budget there.
- **Promoting privacy**: be diligent in protecting personal info. Store as little data as needed.

Additional principles listed by US CISA

Promote privacy of both users *and* systems:

```
HTTP/1.1 200 OK
Date: Thu, 12 Jun 2014 14:15:01 GMT
Server: Apache/2.2.21 (Win32) PHP/5.4.7
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

- Don’t leak version numbers, configuration info, etc.
- Don’t leak sensitive error messages, e.g., SQL/PHP errors.

Additional SAFECode secure design principle

- **Compromise recording**: sometimes reliably recording a compromise can be used instead of more elaborate defenses
  - Example: use surveillance cameras to protect a building.
  - Example: log all accesses to files instead of complex permissions
- This is similar to the **break glass principle**: create methods to perform otherwise-disallowed actions under emergencies:
  - Example: accessing patient info during emergency. Access is logged and later audited. Abuse can be detected during audits.

Secure design principles

Minimize attack surface (part one)

- Network attack surfaces
  - Reduce open ports, protocols, services, devices & their interfaces
- Software attack surfaces
  - Remove unnecessary or unused code (might be exploited)
  - Remove untested code
  - Check/audit 3rd party software
- Human attack surfaces
  - Think about phishing attacks and how they can be minimized

Secure design principles

Minimize attack surface (part two)

- Reduce area & exposure of attack surface
  - = Principle of least functionality (turn off features you don’t need)
  - Deprecate unsafe functions
  - Eliminate APIs vulnerable to attacks
- Reduce accessibility of attack surface
  - Example: limit physical access to a service
  - Example: restrict direct access to databases
  - Example: limit the search feature of a website to authorized users

Secure design principles

Minimize attack surface: other examples

- Automatically log off user after n minutes
- Automatically lock screen after n minutes
- Unplug network connection if you don’t use it
  - Similar: turn of Wi-Fi or Bluetooth when you don’t use it
- Switch off computer if you don’t use it
- Limiting installable software and functionality of that software
- Disable services in a router, disable APIs for debugging, etc.

Secure design principles

Establish secure defaults (≈ fail-safe defaults)

- By default, security should be enabled, e.g., allow no access.
- Protection scheme identifies conditions when access is permitted.
- It should be up to the user to reduce their security (if allowed)

- Example: by default a web framework should enforce a minimum password length and complexity.
- Counterexample: when Bluetooth is by default turned on, any vulnerability in the implementation can always be exploited.

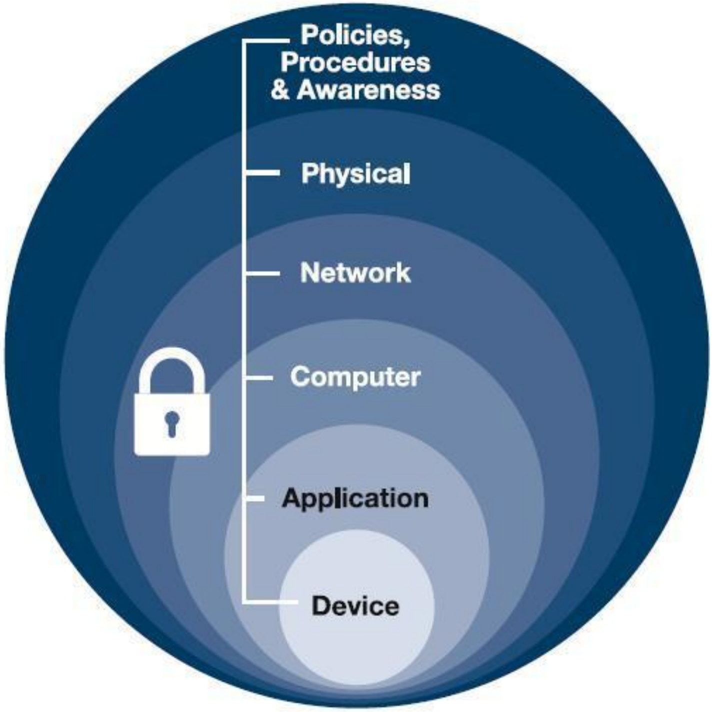
Secure design principles

- **Principle of least privilege**
  - Programs and users should operate using the least set of privileges necessary to perform a task
  - Example: don’t run all services as the root user.

Secure design principles

Defense in depth

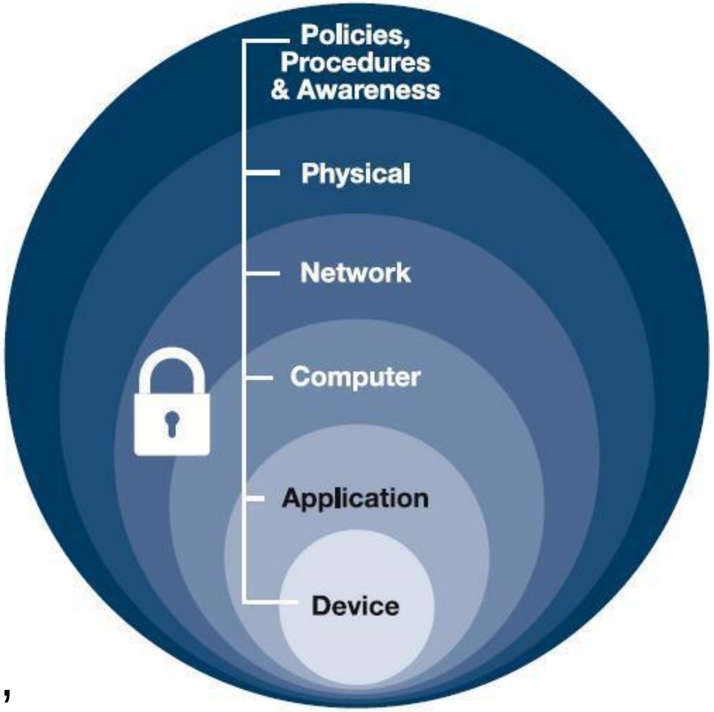
- Multiple layers of security controls
  - If one control fails, another is still in place
- Example: protecting customer data:
  - The (web)service runs with low privileges
  - The wider network requires authentication
  - Internal network is protected by a firewall
  - Physical security to protect access to the server
  - Logging of accesses (in software and cameras)



Secure design principles

Defense in depth

- Counterexample: old /etc/passwd
  - Used to contain hashed password along with other user data.
  - This file was world readable. Dictionary attacks are possible, so this is not ideal.
  - Better: separate user data from passwords, and put passwords in a different file
  - These days passwords are stored in /etc/shadow instead, this file can only be read by root



Secure design principles

Ensure confidentiality

› Encrypt sensitive data

› Use standardize algorithms

Ensure integrity

› Use secure boot (requires specific hardware components)

› Design secure update process

› Verify integrity of your system/app

Secure design principles

Ensure authenticity & non-repudiation

- Use standardized protocols & algorithms

Secure communication

- Never communicate over insecure channels
- Verify authenticity of data
- Use standardized protocols

Secure design principles

Input validation

- Still one of the most common issues!
- Consider allow-lists, deny-lists, etc.
- Proper testing of any input supplied by a user / application
- Use **allow-list** and not deny-list when validating input

Secure design principles

Programming language

- Languages running inside VMs (Java, C#) reduce risk of buffer overflows. Or use languages like Go and Rust.
- Picking the language can be considered a design technique

Use existing security protocols and libraries

- Don’t invent your own! Use well-known existing libraries.
- Don’t implement (an existing protocol) yourself!
- More general: for any task, see if there’s *trustworthy* libraries

Secure design principles

Be reluctant to trust

- Having to trust something is *not* a good thing
  - “Trusted” ≠ “trustworthy”
  - Trust is transitive!
- Minimize the Trusted Computing Base (TCB). That is, minimize the code/libraries/… that must be trusted
- Treat input as untrusted, assume users can get social engineered, etc.

Be reluctant to trust

You verified **all** source code. Can everything now be trusted?

- Ken Thompson's *"Reflections on Trusting Trust"*: no!
- **Required & highly recommended reading** (only 3 pages)

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

Be reluctant to trust

You verified **all** source code. Can everything now be trusted?

› Ken Thompson's *"Reflections on Trusting Trust"*: no!

Summarized, a backdoor may still exist in the compiler:

1. When login.c is compiled, the compiler adds a backdoor:

```
if (name == "ken") { don't check password;
                     log in as root; }
```

2. When the compiler compiles its own source code, it adds the backdoor in (1) and (2) to the compiled binary!

Be reluctant to trust

Conclusion: you always have to trust something/someone

- The goal is to reduce how much things you need to trust

Application of principles

Note that the principles can be applied at many levels:

- In the source code
- At the operating system level
- At the network level
- Within an organization
- …

Defensive programming

Defensive programming

> […] a form of defensive design intended to **ensure the continuing function** of a piece of software **under unforeseen circumstances**. [..] often used where high availability, safety, or security is needed.
>
> \- [Wikipedia]

→ More on the coding level (not architectural or design)

→ Core idea: defend against the impossible, because the seemingly impossible might happen!

Defensive programming: basic stuff

- Source code should be readable and understandable
- Software should behave predictable on unexpected input
- Assume the software will be attacked. Use safe functions.

Several coding standards exists, we cover example principles

- OWASP Secure Coding Practices, Quick Reference Guide
- Oracle Secure Coding Guidelines for Java SE
- CERT Secure Coding Standards

Key coding principles

DRY: Don't Repeat Yourself

- Duplication in logic should be eliminated via abstraction
- Duplication in processes should be eliminated via automation

SOLID principles for object-oriented code

- Not necessary security-related, but results in better code
- Code will also be easier to test, increasing security
- See: github.com/vishalMalvi/SOLID-Principles-in-Swift or codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp

Some key principles: SOLID (object oriented)

- **S**: Each class has a single responsibility
  - Example: one class shouldn’t process data *and* write/output it.
- **O**: Open (for extension) – closed (for modification) principle
  - Class should be extendable without modifying the class itself
- **L**: Liskov (sub-type) substitution principle: child (derived) class should be usable in place of parent class
  - Every subclass or derived class should be substitutable for their base or parent class

Some key principles: SOLID (object oriented)

- **I**: Interface segregation principle: use specific interfaces instead of general-purpose interfaces
  - = large interfaces should be split into smaller ones
- **D**: Dependency inversion principle: high-level modules should not depend on low-level modules
  - Bad example: using a MySQLConnection class to communicate with the database.
  - Good example: using a DBConnectionInterface class instead.

Properly handling errors I

Incorrect handling of unexpected errors is a major cause of security breaches

- Example: fallback to insecure crypto protocol for backwards compatibility
- Example: crashing on failure, leading to DoS attack

Having exceptions in a programming language has a big impact, they are harder to ignore.

Properly handling errors II

- Detect errors & handle them appropriately
- A failure (exception or return value) should follow the same execution path as disallowing the operation.
- Log errors & investigate errors

Variants of failing insecurely

- **Information leakage**: error message or code can leak sensitive information or data that was being processed
- **Ignoring errors**: easier in a programming language with exceptions such as C
- **Misinterpreting** errors
- **Handling wrong exceptions**
- **Handling all exceptions**
- …

Failing insecurely: an example

```
char dst[19];
char *p = strncpy(dst, src, 19);
if (p) {
    // everything went fine, use dst or p
}
```

- Programmer incorrectly thinks `strncpy` returns `NULL` when `src` is more than 19 characters.
- But `strncpy` never returns an error! Always returns `dst`!
  - Possible risk is that `dst` may not be null-terminated

Failing insecurely: an example

Imagine a Local System ("Admin") service in Windows:

```
ImpersonateNamedClient(someUser); // become someUser
DeleteFile(fileName);
RevertToSelf(); // become Local System (Admin)
```

What's wrong here?

- What happens in `ImpersonateNamedClient` fails?
- Might delete files that user shouldn't be able to delete?

Failing insecurely: suspicious code

```
try { // ...
} catch (Exception ex) {
    // do nothing
}
```

This is suspicious because:

1. Nothing happens in the catch block
2. Catching “Exception” is very broad

→ Even in languages with Exceptions errors can be improperly and/or insecure handled!

Failing insecurely: real-life example

Security Flaw in OpenSSL and OpenSSH

- Got assigned CVE-2000-0535
- Pseudo Random Number Generator (PRNG) in OpenSSH and OpenSSL seeded with /dev/random
  - But failure to check for the presence of /dev/random …
  - … which did not exist on FreeBSD-Alpha

Minimize attack surface (at coding level)

- Zero-out arrays that contain sensitive information as soon as it's no longer needed. This reduces the chance of leaks.
  - E.g. zero-out decrypted data, passwords, encryption keys, etc.
- Your compiler may "optimize away" these operations!
  - Smart compilers may realize that after a "`memset(array, 0, len)`" the array is no longer used, meaning the `memset` can be skipped
- The language, library, or operating system may provide functions to securely zero-out memory
  - Won't be optimized away, clears caches, clear data cached to disk,…

More info: https://crypto.stackexchange.com/questions/9998/should-i-delete-cryptographic-data-from-memory

Use safe functions only

Libraries typically have known unsafe functions:

- For instance, `eval` and `setTimeout`, `strcpy`, and so on.
- Use safe equivalents of unsafe functions
- Use tools to automatically detect (and prevent) the usage of unsafe functions
- Use the latest compiler & toolchains. They can warn you about the usage of unsafe tools.
  - Assure coding conventions are **realistic and enforceable**.

Secure coding

- Use **built-in security features** of frameworks
  - Helps address known classes of issues, e.g., SQL injection or cross-site scripting vulnerabilities
- A framework/library/component should be **loosely coupled** so that it can be easily replaced/upgraded when needed

Security mechanisms

Security Mechanisms

Tools to implement security controls

1. Cryptography
   - Use a well-known standard: NIST, FIPS, IETF, ISO,…
2. Security Protocols
   - TLS, IPsec, Wireguard, SSH, …
3. Hardware security
   - Trusted Platform Module (TPM): performs crypto operations
   - Hardware security module for key generation, storage, and usage
   - Trusted execution zones: run code in secure enclave

Security Mechanisms

4. Authentication
   - Password-based, multi-factor authentication, certificate-based, biometric,…
5. Authorization
   - Attribute-based access control, claim-based approach, group/role based, …
6. Key management
   - Consider short and long-lived keys
   - Consider full lifecycle: generation, distribution, storage, revocation,…

Maturity / assessing

Variants of the secure software lifecycle

There are many secure software development lifecycles

- **OWASP** Software Assurance Maturity Model (**SAMM**)
- Makes software security **measurable** so you can better manage it
- Understand where you are, and what you can still do/improve
- Can do self-assessment and compare result with other companies

References

Required reading: Ken Thompson, "Reflections on Trusting Trust". Turing Award Lecture, Communications of the ACM, 1984.

Optional reading:

- "Security Controls for Computer Systems", commonly called the Ware report. Written in 1970 for the DoD, declassified in 1979.
  - Helped found the security research field. Still has important security design lessons for today!
  - The basis of "the orange book" a.k.a. "Trusted Computer System Evaluation Criteria (TCSEC)"
- Preventing Privilege Escalation by Provos, Frield, and Honeyman at USENIX Security, 2003. This paper introduces privilege separation for OpenSSH.