

6. Secure Design Principles

Prof. Dr. Marc Rennhard
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema@zhaw.ch

Marc Rennhard, 29.05.2014, SSI_DesignPrinciples.pptx 1

This topic is covered, e.g., in the following books:

- *J. Viega and G. McGraw, Building Secure Software, ISBN-13: 978-0201721522* (Chapter 5: Guiding Principles for Software Security)
- *N. Dasawani, C. Kern and A. Kesavan, Foundations of Security, ISBN-10: 1590597842* (Chapter 2: Secure Systems Design, Chapter 3: Secure Design Principles)

The slides in this chapter partly contain information from these books.

Content

- **Secure Design Principles** you should keep in mind when thinking about security during software development
- A brief exercise to **apply the principles** to an e-banking scenario

Goals

- You **know and understand** the secure design principles and can provide examples for each of the principles
- You can **apply the principles to a given scenario** and assess whether a principle is met or violated

Secure Design Principles

Secure Design Principles

- The **technology** we are using continuously advances
 - New platforms, programming languages, protocols, frameworks...
- Similarly, **new threats** arise, sometimes driven by new technology
 - This requires adapting secure development practices over time
 - Considering new attack vectors, employing new countermeasures, integrating new security technologies, and adapting security testing
- However, there are also some **fundamental secure design principles** that have established themselves over time
 - Fundamental in the sense that they are technology-independent and likely to be true also in the (far) future
- When developing secure systems, it's important to **keep these principles in mind**
 - Using them appropriately in your system should help you avoiding many security problems

Marc Rennhard, 29.05.2014, SSI_DesignPrinciples.pptx 5

Threats driven by new Technologies

Very often, new technologies mean new threats, e.g.:

- The introduction of Javascript that can be executed in browsers enabled several different attacks
- The possibility to install and use apps on smartphones introduced a large number of attacks in the mobile world that were simply no issue previously

Secure Design Principles Overview

- Secure the Weakest Link
- Defense in Depth
- Fail Securely
- Principle of Least Privilege
- Secure by Default
- Keep it Simple
- Hiding Secrets is Hard

Secure the Weakest Link (1)

- An information system usually consists of several **different components**
 - Browser, client computer, network, server, database...
 - But also users, administrators, support personnel, processes...
- The overall **security is determined by the weakest component**
 - Attackers try to identify the weakest component of a system as this is where they will most likely succeed
- Example: Assume the communication between client and server is secured with TLS that uses strong ciphers
 - Attackers will most likely **not try to break the crypto**, as the success probability is virtually zero
 - The attacker will focus their attacks on other "**more promising**" areas, e.g. the users or vulnerable client or server components
 - This does not mean breaking crypto has never succeeded (due to e.g. poor key generators or implementation flaws), but it's simply less likely

Secure the Weakest Link (2)

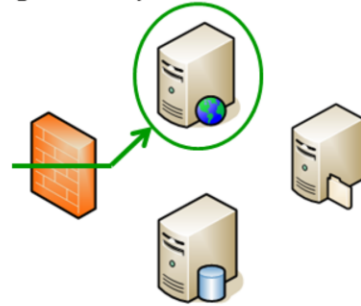
- Weak links are usually identified by performing **risk analysis**
 - Good risk management should always address the highest risks first and not the ones that are easy to mitigate
 - This guarantees that security investments are made at the right places
- Typical weak links:
 - **Weak passwords**, especially if users are not forced to fulfill minimum password strength requirements (length, character mix etc.)
 - **Insecure support processes**, e.g. related to password recovery (mother's maiden name...)
 - **People**, e.g. social engineering attacks on end users (phishing) or support personnel ("I'm the boss and need a new password NOW")
 - **Implementation defects**, e.g. web applications using easy-to-guess session IDs or failing to perform proper input validation

Defense in Depth (1)

- Defense in Depth means that risks should be managed with multiple **diverse defensive strategies**
 - If one layer of defense fails, another layer may prevent a successful attack

Example: A **web application consisting of multiple server components**

- The development team decides that the following security measurements should be taken
 - **Harden the component reachable** from the outside (e.g. the web application server)
 - On the router, use a **firewall** so only the web application server can be accessed from the outside and only via HTTPS
 - This should “guarantee” that no other servers can be accessed from the outside and that the web application server cannot be compromised



Marc Rennhard, 29.05.2014, SSI_DesignPrinciples.pptx 9

Defense in Depth and Buffer Overflows

We have already talked about defense in depth when discussing buffer overflows: The first layer of defense should be secure programming, i.e. making sure no buffer overflow vulnerabilities are in the code. If a vulnerability still occurs, we then can still hope that the various technical defenses (stack canaries, ASLR etc.) help to prevent exploitation of the vulnerability.

Defense in Depth (2)

- Based on this assumption, they decide the following:
 - Internal communication between the server components can be done **in the clear**
 - Not directly exposed servers **don't require much security attention**
- But what if an attacker **compromises the firewall**?
 - He gets access to all non-encrypted traffic in the network (ARP spoofing)
 - He "sees" the other server components and may likely find vulnerabilities that can be exploited to compromise them
 - → The attacker has to overcome **only one layer of defense** to succeed
- In this case, a defense in depth approach would therefore include:
 - **Encrypting the traffic** also between the server components, which prevents the attacker from reading and manipulating it
 - **Harden** not only the exposed, but also the apparently "hidden" server components to avoid compromise



Defense in Depth (3)

- Defense in Depth also means thinking **beyond preventive measures**
 - Consider the case when your preventive measures have failed and the attack takes or took place
- A powerful strategy should include mechanisms to **prevent, detect, contain, and recover from attacks**
- Example: how do banks defend against robbery?
 - Security guards outside the bank and cameras can **prevent** attacks
 - **Detecting** the attack is usually quite simple in the brick-and-mortar world
 - Having bank tellers stationed behind bulletproof glass may help **containing** the attack if violence breaks out (protecting the lives of the tellers)
 - Another **containment** measure is that two people – who are rarely at the bank at the same time – must each provide a key to the vault with the really valuable stuff
 - Handing a specially prepared briefcase that emits dye when the robbers open it may help **recovering** from the attack
 - Appropriate insurance is also a method to **recover** from robbery

Defense in Depth (4)

Prevent / Detect / Contain / Recover **in the cyberworld** using a password guessing attack as example:

- Password-guessing attacks can be **prevented** by requiring users to pick passwords with certain requirements
- Password-guessing attempts can be **detected** by monitoring server logs for large number of failed login attempts from the same or a few IP addresses (an IDS can do this)
- Assuming an attacker manages to capture username/password pairs, one can **contain** the attack by denying all logins from suspicious IP addresses (assuming this can be detected) and changing the passwords
- Finally, to **recover** from the attack, one could do more detailed monitoring of the accounts and source IP addresses involved in the attack

Marc Rennhard, 29.05.2014, SSI_DesignPrinciples.pptx 12

Diversity in Defense

“Diversity in Defense” is often mentioned along with Defense in Depth. The idea is to use multiple heterogeneous systems that do the same thing. For instance, using different operating systems to store multiple backup copies will likely reduce the risk of a full backup loss in case a particular operating system type is infected by malware.

Of course, Diversity in Defense comes at a cost, as you need experts in your IT staff for multiple systems and the complexity increases (e.g. you have to monitor and apply patches to different types of systems).

Fail Securely (1)

- Fail securely means that a failure (which will happen in complex systems) **must not compromise the security** of the system
 - One reason why this principle is often violated is because in the case of failure, the focus is very often on "keep things running at all costs"
- Example: A company uses correctly configured **firewalls** to protect access to its various systems
 - The company also has a **spare firewall** in case one of the firewalls becomes inoperable
 - The firewall is configured as "**let through everything**" so it can easily replace a malfunctioning firewall with minimal service interruption
 - An attacker who knows this can try to disable a firewall (by exploiting a vulnerability), which causes the firewall to be replaced with the spare one
 - Until the new firewall is correctly configured, this gives the attacker a significant **advantage**

Fail Securely (2)

- Another possibility to violate the fail securely principle happens when a system supports also **earlier insecure versions of a protocol**
 - E.g. because older clients supporting only an older version are still used
 - But this means that **if using the secure protocol version fails, the system gets less secure** → this should be avoided
- Even worse, this may be exploited in a **"version downgrading" attack**
 - In this attack, the attacker forces both endpoints to use the earlier, insecure version although both would support the new one
 - To do so, the attacker acts as a man-in-the-middle and modifies some protocol messages to convince both client and server that the other side only supports an older protocol version
- Examples:
 - Downgrading of the **NTLM** protocol to its predecessor LM, which allows cracking even complex user passwords
 - Downgrading to **SSLv2**, which allows manipulating the cipher suites offered by the client

Marc Rennhard, 29.05.2014, SSI_DesignPrinciples.pptx 14

Version Downgrading Attack on NTLM

In case you visited the course Internet-Sicherheit (ISI) you may remember the lab "Network Attacks", where you performed the version downgrading attack on NTLM yourself (using the tool Cain) to crack the password used in the challenge response protocol using rainbow tables. With Windows XP clients, this worked but since Windows Vista, it is prevented per default (although the LM protocol can still be enabled via registry settings)

Fail Securely (3)

Another example: Credit card payment authorization

- To prevent credit card fraud, vendors use terminals to check a card before payment is authorized
 - The terminal contacts the credit card company to make sure the card is not reported stolen or that the credit limit has not yet been reached
 - In addition, the transaction is denied if any "suspicious spending pattern" is detected
- That's great, but what if the terminal or the phone line is down?
 - Vendors still sometimes use the old machines that make an imprint of the card, but none of the checks above takes place (and no PIN is required either)
- This means that if the system fails, it gets less secure
 - This can be exploited in a "physical version downgrading attack"
 - To execute the attack, try to "cut the phone line" before shopping



Principle of Least Privilege (1)

- A user or program should be given the **least amount of privileges necessary to accomplish a task**
 - So a malicious user and a compromised/malicious program can cause only limited damage
- One reason why this principle is often violated is **laziness**
 - It's usually simpler to write or install a program when it's run with full access rights as one does not have to think about access rights to low level system resources (e.g. files)
- Example of good practice: **Postfix**
 - Postfix is a popular *ix mail server
 - It is heavily **modularized** and each component (e.g. delivery to local mailboxes, sending of outbound messages) gets minimal access rights to do its job
 - There's just one (small) component running as root (the master daemon) as binding to port 25 is only possible by a process that runs as root

Marc Rennhard, 29.05.2014, SSI_DesignPrinciples.pptx 16

Postfix

Communication between the Postfix components, which are all individual processes, takes place via interprocess communication techniques that are offered by every modern operating system. On *ix systems, this happens via Unix Domain Sockets.

An overview of the anatomy of Postfix can be found here: <http://www.linuxjournal.com/article/9454>

Principle of Least Privilege (2)

Examples of **bad practice**:

- Up to Windows XP, many programs required administrator privileges to function correctly and consequently most users worked with **full administrator rights**
- Many server programs still run with **root privileges** under *ix per default, which has repeatedly caused security problems in the past (e.g. sendmail)
- Web applications accessing a database often do this with a database user that has **full access rights** (sometimes including rights to modify the database schema)

- Programs and operating systems should be delivered/installed with **secure default configurations**
- This includes, e.g., the following:
 - **Turn off or don't install features** that are not required by most users
 - E.g. don't enable file sharing by default as most users won't need it
 - Personal firewall: **block** all incoming connections by default
 - Enable automated **software updates** by default (usually good for end-user computers, more questionable for servers)
 - The standard configuration of **TLS** should not use insecure cipher suites
- Example of a positive development:
 - Years ago, **Windows operating systems** were installed with many services enabled (e.g. IIS was turned on by default)
 - This has significantly changed during recent years and Windows operating systems are installed with reasonable default settings today

Keep it Simple (1)

- Complexity is the enemy of security
 - It's much more difficult to **develop and maintain** a complex system in a secure way
 - Similarly, it's much more difficult to **analyze and test** a complex system with respect to security – the likelihood you are going to miss something important is very high
- Simplicity also means that "**it should be easy to use a system in a secure way**"
 - Don't expect users to read manuals – they won't
 - Don't expect your users to understand security and don't expect them to configure it correctly (that's why "Secure by Default" is important)
 - Don't give the users the option to circumvent security as they likely will
 - Especially when security is perceived as a usability hindrance

Keep it Simple (2)

Some **good practices** to follow:

- Select a **decent, as simple as possible** (only as complex as necessary) **software design**
 - It's the basis also for sound security design
- **Re-use proven software components or technologies** when appropriate
 - This is especially important with security functions such as secure communication protocols and cryptographic algorithms
- **Implement security-critical functions only once** and place them in easily identifiable program components (e.g. in a separate package)
 - This will make it easier to use security-critical functions in your software
 - For instance, use a single `checkPassword()` or `checkAccess()` method in your software and take care to review that method for correctness

Marc Rennhard, 29.05.2014, SSI_DesignPrinciples.pptx 20

Spreading Security Functions in the Code

One often sees that security code is spread throughout a program. This can mean that access control is implemented at different places in different ways, which naturally significantly increases the probability for security vulnerabilities.

Keep it Simple (3)

- Don't give the users security dialogues they can ignore
 - As they most likely will – because they want to “get things done”
 - A study in 2009 analyzed how users react to certificate warnings when browsing the web: approximately 50% ignore the warning



Marc Rennhard, 29.05.2014, SSI_DesignPrinciples.pptx 21

Study about Ignoring Certificate Warning

Joshua Sunshine, Serge Egelman, Hazim Almuhiemedi, Neha Atri, and Lorrie Faith Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. Presented at 18th Usenix Symposium, August, 2009. Available at <http://lorrie.cranor.org/pubs/sslwarnings.pdf>

Keep it Simple (4)

- For the average user, the warning more looks like this:



Marc Rennhard, 29.05.2014, SSI_DesignPrinciples.pptx 22

The screenshot above was taken from <http://www.usenix.org/events/sec09/tech/slides/sunshine.pdf>

Hiding Secrets is Hard



- Don't build security on the assumption that the adversary does not know **how your software works or that he cannot find secrets** in your software
 - This is also known as "**Security by Obscurity**"
- There are powerful **reverse engineering** methods (e.g. decompiler) that can be applied to binaries
 - It's very hard to hide the actual functionality of, e.g., a proprietary cryptographic algorithm
 - Hiding a secret such as a cryptographic key in a code binary is virtually impossible (e.g. search for randomness)
 - You can use code obfuscation to make the task for the adversary more difficult, but a determined adversary will most likely find out in detail how your software works
- **Several examples undermine this**
 - DeCSS for the cryptographic "protection" of DVD contents
 - The failure of many software copy protection mechanisms

Marc Rennhard, 29.05.2014, SSI_DesignPrinciples.pptx 23

Software Copy Protection

The difficulty with software protection mechanism is that somewhere in the code, a "check" takes place whether, e.g., a valid license key is available or whether the software was activated online. With enough effort, it is usually possible to find that code section and adapt it (directly in the binary) so that the actual check is circumvented and the software can be used.

Code Obfuscation

The purpose of code obfuscation is making reverse engineering more difficult by making decompiled code more difficult to read. Code obfuscation can be achieved through one or more of the following methods:

- Source or binary structure obfuscation - A source code obfuscator accepts a program source file, and generates another functionally equivalent source file, which is much harder to understand or reverse-engineer. This is useful for technical protection of intellectual property when source code must be delivered for public execution purposes.
- Data Obfuscation - This is aimed at obscuring data and data structures. Techniques used in this method range from splitting variables, promoting scalars to objects, converting static data to procedure, change the encoding, changing the variable lifetime etc.
- Control Flow Obfuscation - This aims at changing the control hierarchy with logic preservation. Here false conditional statements and other misleading constructs are introduced to confuse decompilers, but the logic of the code remains intact.
- Preventive Obfuscation - Here the focus is on protection against decompilers and reverse engineering methods. Renaming metadata to gibberish or less obvious identifiers is one such technique, like defining function InterestCalculation() as x().

Source: <http://palpapers.plynt.com/issues/2004Dec/code-obfuscation/>

Code obfuscation may certainly be a good idea if your software provides, e.g., an ingenious algorithm (that you don't want to publish) to perform a non security-critical operation such as processing an image in an photo editor program, but it shouldn't be used to hide the functionality of security-critical operations from an adversary.

Secure Design Principles – Exercise

Secure Design Principles – Exercise (1)

Assume a [web-based e-banking application](#), which should provide “good” security. For the following statements, think about whether it’s a good or bad choice and to which secure design principle it is associated.

- To support as many users as possible, the server is configured to accept connections from browsers that only support SSLv2
- Users can disable the second authentication factor (login code received by SMS) in their profile settings
- A web application firewall (WAF) that can detect typical web application attacks (SQL injection, Cross-site scripting etc.) is used in front of the web application server

Secure Design Principles – Exercise (2)



- The web application server runs with root privileges, as this is required to write the logs
- To increase security, users get a hardened browser on a read-only USB stick, which must be used to access the e-banking site
- To further increase security, this browser uses a proprietary and secret encryption algorithm developed by the bank for TLS encryption

Summary

- There exist some **fundamental secure design principles** that have established themselves over time
 - Secure the Weakest Link
 - Defense in Depth
 - Fail Securely
 - Principle of Least Privilege
 - Secure by Default
 - Keep it Simple
 - Hiding Secrets is Hard
- They are **technology-independent**, which means they are likely to be true also in the future
- **Keeping these principles in mind** during software development should help you to avoid many security problems