CITS3007 Secure Coding
Secure software development

Unit coordinator: Arran Stewart

Highlights

- Risk management
- Design processes
- Design principles
- Security testing

Risk management

Security is about managing risks.

No system can be *perfectly* secure (except perhaps one that is never actually used).

But we can try to ensure that we bring the *risk* of serious security problems occurring down to a tolerable level.

Risk management

Steps in risk management:

- *Identify* risks
- *Assess* their likelihood and impact
- *Rank* them
- For all risks above our level of tolerance:
  - Avoid/resolve, mitigate, transfer or accept

Risk management

Identifying risks:

- Can reduce to "filling in forms"
- But *proper* risk identification requires creativity, brainstorming, communication with stakeholders.
- Needs to overcome positivity bias/groupthink
  - Pre-mortem: Imagine we're in the future and the system has *already* failed catastrophically. Ask yourselves, how did this arise?

Risk management

- Avoid/resolve the risk: completely eliminate it
- Mitigate the risk: reduce the likelihood or impact
- Transfer the risk: assign or move the risk to a third-party (e.g. outsource, insure)
- Accept the risk: acknowledge the risk, and decide not to resolve, transfer or mitigate

Incorporating security

Approaching security as something you can simply "add on" to existing systems or processes as an extra phase or step is doomed to failure.

The aim should be to *incorporate* security into existing processes, at all stages of the software development life cycle:

- analysis/requirements elicitation
- design
- implementation and testing
- maintenance/operation
- disposal

Incorporating security

Most of these elements of secure development assume you're already applying (non-security) best practices – version control, testing, etc.

If your other processes are bad, then adding on (e.g.) "secure testing" isn't going to make them any better.

Incorporating security

Requirements stage:

- Identify security goals
- Identify essential threats

Incorporating security

Design stage:

- Risk analysis
- Plan for secure implementation and secure testing
- Design review

Incorporating security

Implementation/testing stage:

- Code review
- Risk analysis for libraries used
- Security testing
  - (Possibly) penetration testing

Incorporating security

Maintenance/operation:

- Handling reported vulnerabilities
- Regression testing

Incorporating security

Disposal:

- If the product is being disposed of – what happens to any sensitive data?

Design processes and principles

Make design assumptions explicit

- What budget, resource, and time constraints limit the design space?
- Is the system is likely to be a target of attack?
- Are there non-negotiable requirements (e.g. compatibility with legacy systems)
- What are the expectations about the level of security the system must adhere to?
- How sensitive are different sorts of data? How important is it to protect the data?
- Are there any anticipated needs for future change to the system?
- What performance or efficiency benchmarks must the system achieve?

Ensure there are security requirements

These can be user stories, or more traditional requirements.

But they should set out:

- what the security goals for the system are
- whether there are competing stakeholder needs
- whether there are acceptable costs or trade-offs to be made
- any unusual requirements

The goals and requirements should be *achievable*!

Threat modeling

Identify essential threats to a system's security.

For example:

- Do we transmit customer data between client and server? Then one threat is that it could be intercepted.
- Do we store sensitive customer data in a database? Then one threat is that confidentiality of the database could be breached.

Threat modeling

STRIDE is a Microsoft-developed method for identifying and reasoning about threats to a system.

The name is a mnemonic for categories of threats:

- **Spoofing:** attacker pretends to be someone else
- **Tampering:** attacker alters data or settings
- **Repudiation:** user can deny making attack
- **Information disclosure:** loss of personal info
- **Denial of service:** preventing proper site operation
- **Elevation of privilege:** user gains power of root use

Threat modeling

STRIDE uses data flow diagrams to follow the path of data through the system.

- For each flow / transformation / storage:
  - Are there vulnerabilities to **S T R I D E**?
  - Can this route be attacked? What is the attack surface?
- Design mitigations/countermeasures

STRIDE is intended to be *developer*-friendly

- doesn't assume we know about the end-user's risk appetite
- doesn't emphasise risk/impact assessment (developers may not be able to do so)

Security reviews

The software development process should incorporate *reviews*.

- A *security design review* involves someone assessing and critiquing the software design for possible problems.
- A *security code review* involves the same, but for code that is being submitted / amended.

Security reviews

When to conduct secure design reviews?

Once the design is reasonably stable.

Kohnfelder's advice is to separate *security* design reviews from other reviews (e.g. of functionality).

Security reviews

If a security review is to be useful, it has to be done carefully.

Reviewers need to

- study the design and supporting documents
- clarify where necessary and investigate further
- identify the *highest-risk*, most security-critical parts of the system to give special attention to
- write up and document their findings and recommendations

The organization needs to

- have a process in place to ensure reviewing findings and recommendations are followed up on and signed off.

Security reviews – four questions

1. What are we working on?
   - What are the high-level goals of the design?
2. What can go wrong?
3. What are we going to do about it?
4. Did we do a good job?

Some principles of secure software design

- Redundancy
  - Defence in depth
- Exposure minimization
  - Principle of least privilege
  - Separation of Privilege
  - Secure by default
- Economy of design

Saltzer and Schroeder

Saltzer and Schroeder (1975)'s classic principles:[1]

- **Economy of mechanism:** keep it simple
- **Fail-safe defaults**
- **Complete mediation:** check everything, every time
- **Open design:** assume attackers get the source and spec
- **Separation of privilege**
- **Least privilege:** no more privilege than needed
- **Least common mechanism:** beware shared resources
- **Psychological acceptability:** are security ops usable?

[1] Saltzer, Jerome, and Michael D. Schroeder. "The protection of information in computer systems." *Proceedings of the IEEE* 63.9 (1975): 1278-1308.

Defence in depth

Combine independent layers of protection.

- Then, for something to be insecure/exposed, they *all* need to fail.

Ensure the weakest link is secured.

Defence in depth

Example:

- Sandboxes/VMs
- Run your student assignment-checking code in a Docker sandbox, as a non-root user, in a VM, in Singapore.
  - Even if someone comrpomises a web-server program, there's limited information they have access to.

Principle of least privilege

> Every [component] should operate using the least amount of privilege necessary to complete the job.[1]
>
> – *Jerome Saltzer*

- Functions, programs, processes etc. should be able to access only the information and resources they they need to do their job
- e.g. If they don't need "write" access, they shouldn't have it

[1] Saltzer, Jerome H. (1974). "Protection and the control of information sharing in MULTICS".

Separation of Privilege

A sort of corollary of the Principle of Least Privilege.

- Where possible, split responsibilities between components/processes/systems, so that no one of them has too much power.
- The patterns we looked at for setuid programs are examples of this (e.g. splitting into client/server)

Separation of Privilege

- Separate the system into independent modules
- Limit interaction between modules

Secure by default

Give things secure and/or safe values by default.

- Even if a user/developer does *no* customization, the system shouldn't be unsafe or insecure

Economy of design

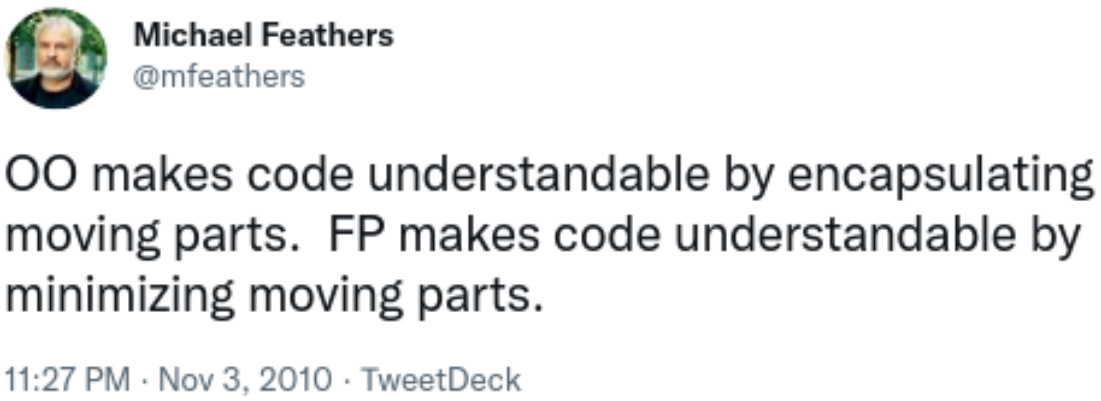
> Keep things as simple as they possibly can be (but no simpler).[1]
> – *Einstein? William of Ockham? Anonymous?*

- The simpler the design, the easier it is to analyse and the fewer places bugs can lurk

[1] https://quoteinvestigator.com/2011/05/13/einstein-simple/

Economy of design

Minimize or hide “moving parts”.



– Michael Feathers,
https://twitter.com/mfeathers/status/29581296216

- Keep exposed interfaces as small as possible (information hiding)
- Keep data as immutable as possible

Implementation quality

Code style

Consistent code style makes it easier to conduct code *reviews*.

Human reviewers shouldn't spend their time checking for issues that can be checked mechanically.

Code reviews

Someone other than the original developer should always sign off on code that's checked into version control/ merged with main branches.

Empirically, code reviews are highly effective at preventing bugs from getting into a software product.

Static and dynamic analysis

In previous lectures, we've looked at how automatic static and dynamic analysis can be incorporated.

Don't "roll your own" crypto

Unless you have a very good understanding of cryptography, it's better to make use of existing cryptography libraries.

It's very easy to make a mistake in implementation that can render the cryptography worthless.

Don't reinvent the wheel

Similar principles apply to most other components, as well – if there's already a trusted and battle-tested implementation of something, it's usually better to use that than write your own.

Security testing

Security testing

Security testing should be *in addition to* normal functional testing.

Systems should have unit tests, integration tests and (sub-)system tests in place.

Security testing

Test for the various things that can go wrong with implementations.

Integer overflows Can they occur? Are they detected/handled?

Memory corruption/problems Does the system handle out of bounds pointers/values? Can the system be overloaded by requesting it to allocate too much memory?

Untrusted inputs Check to make sure bad/blacklisted inputs are rejected.

Exception handling Check that when exceptions or errors occur, the system still behaves robustly.

Security testing – fuzzing

Where possible, use **fuzzing** to see how your program holds up against potential bad data.

Is it robust, or does it crash?

Security regression tests

Whenever a security vulnerability is identified and fixed, tests should be put in place to ensure it doesn't later get reintroduced.

(Ideally – we should improve our tests/practices so that whole *class* of bugs can be avoided.)

Security system tests

Some types of system testing:

- Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Stress testing
  - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
  - test the run-time performance of software within the context of an integrated system

Security system tests

- Recovery testing
- Stress testing
- Performance Testing

We can use these sorts of testing to try and avoid disruptions of *availability*.

When the system is under high load, are excessive resources consumed?

If availability is important, we might also use third party content delivery networks (CDNs).