CITS3007 Secure Coding
Secure software development

Unit coordinator: Arran Stewart

Highlights

- Risk management
- Secure SDLC (software development lifecycle)
- Design processes
- Design principles
- Security testing
- Threat modelling with STRIDE

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

Risk management is basically just asking the question:

- What can go wrong?

So that we can do something about it, *before* things go wrong.

General steps in all risk management processes:

- *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

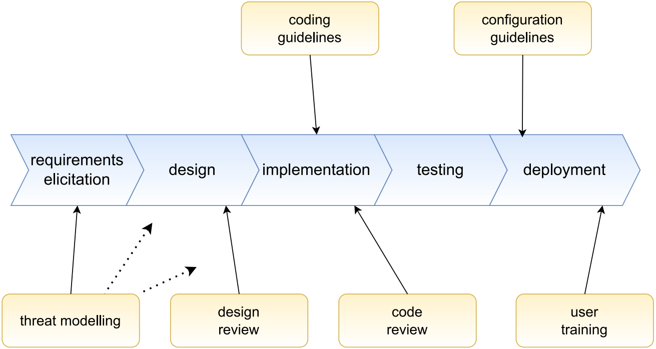
Incorporating security

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



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

Overview

- Make design assumptions explicity
- Ensure there are security requirements
- Perform threat modelling
- Apply principles of secure software design
- Conduct security reviews

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

- Is conducted in the *context* of wanting to protect something of value
- Process whereby potential threats (e.g. vulnerabilities) can be identified, enumerated, and prioritized
- The process of then understanding and communicating those threats (and their mitigations)

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 technically just a taxonomy (plus mnemonic) for threats, developed by Praerit Garg and Loren Kohnfelder (textbook author) at Microsoft.

But used as part of threat modelling, 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

Each of these categories of threats violates some security property we want systems to have

- Spoofing: Violates *authenticity*
- Tampering: Violates *integrity*
- Repudiation: Violates *non-repudiation*
- Information disclosure: Violates *confidentiality*
- Denial of service: Violates *availability*
- Elevation of privilege: Violates *authorization*

Threat modeling with STRIDE

STRIDE approach uses a *model* of the system to identify

- *assets* (valuable data and resources) that need protection
- *flows* of data through the system
- *attack surfaces* (places an attack could originate)
- *trust boundaries* (the borders between more-trusted and less-trusted parts of the system)

Threat modeling with STRIDE

- 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

Threat modeling with STRIDE

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)

More on a suggested process incorporating STRIDE later.

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:** the default configuration should be secure
- **Complete mediation:** check everything, every time
- **Open design:** assume attackers get the source and spec
- **Separation of privilege:** split up responsibilities
- **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

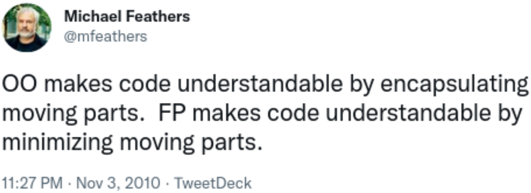
> Keep things as simple as they possibly can be (but no simpler).[1]
> – *Einstein? William of Ockham? Anonymous?*

- 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

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.

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
- Penetration testing
  - simulate an attack on the system
    (is a whole subject of its own – not covered here)

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).

Threat modelling with STRIDE

Four questions

Approach from Adam Shostack at Microsoft:

1. What are we building?
2. What can go wrong?
3. What are we going to do about it?
4. Did we do a good job?

Four questions – activites

1. What are we building?
   - Outcome: a *model* or diagram of the system (and identified assets)
2. What can go wrong?
   - Outcome: prioritized list of threats
3. What are we going to do about it?
   - Outcome: prioritized list of mitigations or countermeasures
4. Did we do a good job?
   - Outcome: tests; gaps identified; improvements to process

Scope

“Small and often” is better than “comprehensive, but never finished”.

- *Full* inventory of all potential threats for a large, complex system could be huge
- But it’s better to do *something than nothing*, and it’s better to identify the *most critical* threats than to aim for completeness

First pass = focus on biggest, most likely threats, to high-value assets

- Other assets and threats can be dealt with later; scope can be increased

What are we building?

The aim is to produce a *model* or high-level description of the system, including assets (valuable data and resources) that need protection.

- Traditionally:
  - data flow diagram (DFD), or
  - Unified Modelling Language (UML)

But *any* sort of model will do.

Could be a design document or a box-and-arrows whiteboard sketch.

Level of detail

No model is perfect – it is a useful *simplification* of reality.

- Needs enough granularity that we can analyse it, identify assets and threats
- Always possible to iterate the process later with more detailed models if necessary
- Too little detail $\Rightarrow$ details will be missed
  Too much detail $\Rightarrow$ the work will take too long

Iterating a model

We can always note down spots where a model could be improved later.

Phrases to watch out for: "sometimes", "also".

- "*sometimes* this data store is used for X", "this component is *also* used for Y" $\Rightarrow$ more detail could be useful

Identify assets

These are things we want to protect.

Usually data.

But could also be:

- hardware
- information technology resources (like bandwidth, computational capacity)
- physical resources (electricity)

Can you think of threats targeting these?

Identify assets

Assets should be *prioritized* – which are most important?

- We could try to hide *everything* about our server, for example
  - But is the best use of our time?
- Compare server details with (e.g.) financial data, password hashes

Identify assets

Don't try and put a dollar value on assets. Avoid superfluous and unrealistic granularity.

- One idea: categorize with "T-shirt sizes"
- "Large" (major assets), "Medium" (valuable assets, but less critical), "Small" (minor consequence).

Remember other parties/stakeholders' viewpoints – something *you* think is of "minor consequence" could be much more important to (e.g.) a customer, CEO, HR, etc.

What can go wrong? – Identify threats

Methodically go through the model, component by component, flow by flow, looking for possible threats.

Identify

- *attack surfaces* (places an attack could originate)
  - Points where an attacker could interpose themselves
- *trust boundaries* (the borders between more-trusted and less-trusted parts of the system)
  - These will intersect data flows
- threats in each of the possible STRIDE categories.

Tip: threats often lurk at trust boundaries.

Identify attack surfaces

These are an attacker's "points of entry", or opportunities for attack. (For example: communication over a network.)

- When we look at mitigations – try to completely remove, or at least reduce, opportunities for attack

Identify attack surfaces

Physical example: we have a building we want to secure.

- What's better – many exits and entries?
- Or: just a single exit and entry, which we can monitor carefully, and have (e.g.) security screening, metal detectors at.

Identify threats

For each of the STRIDE categories – e.g. tampering – we ask, What advantages could an attacker gain if they did/subverted X?

A suggested approach: brainstorm first – come up with ideas quickly, without critiquing or judging them yet

Analyse, understand and prioritize threats

For each identified threat:

- flesh out the details
- try to assess the chance of them happening
- assess what the impacts would be

Analyse, understand and prioritize threats

- For probability and impact – no need for exact numbers – just use a point/level system (e.g. 1 to 3, 1 to 5)
  - Give your levels labels – "likely", "unlikely"; "minor impact", "showstopping / enterprise-destroying"
- Be cautious of unrealistic levels of granularity – can you *really* distinguish "5%" versus "7.5%" probability, or "3/10" from "4/10"?

Ranking threats

Microsoft "DREAD" model:

- Damage: How big would the damage be if the attack succeeded?
- Reproducibility: How easy is it to reproduce an attack?
- Exploitability: How much time, effort, and expertise is needed to exploit the threat?
- Affected Users: If a threat were exploited, what percentage of users would be affected?
- Discoverability: How easy is it for an attacker to discover this threat?can increase scope

What are we going to do about it? – mitigations

- Propose ways of dealing with each threat – usually called "mitigation" or "countermeasures".
- But in full: either mitigate, remove, transfer, or accept.

Mitigations and other approaches

- *Mitigate* risk by redesigning or adding defenses.
  - The aim is either to reduce the chance of the risk occurring, or lower degree of harm to an acceptable level
- *Remove* a threatened asset if it is not actually needed
  - Or if removal is not possible – seek to reduce the exposure of the asset, or limit optional features of your system that increase the threat.

Mitigations and other approaches

- *Transfer* the risk – offload responsibility to a third party.
  - Example: Insurance is a common type of risk transfer
  - Example: Outsource responsibility for e.g. processing payments, or processing sensitive data, to an enterprise with expertise in the area.
- *Accept* the risk (once it's well understood) as being reasonable to incur.

Mitigations – questions to ask

- Can we make the attack less likely to work?
- Can we make the harm less severe – perhaps only some of the data is accessible?
- Can we make it possible to undo the harm – e.g. backups?
- Can we make it more obvious when harm has occurred – e.g. by ensuring we have comprehensive logging and monitoring?

Mitigations – technologies to apply

Mitigations may incorporate

- cryptography
- access control

Did we do a good job? – validation, review and testing

- Validate previous steps, act upon them, look for gaps missed
- Test the efficacy of mitigations, from most to least critical

Validation

- For a model – does it match what has actually been implemented?
- For threats – have we describe them properly? missed any?
  - do they: describe the attack, the context, the impact?
- Other stakeholders – have testing/quality assurance staff reviewed the model?
- Mitigations – is each threat mitigated (or otherwise dealt with)
- Are the mitigations done correctly? Have they been tested?