INF226 – Software Security

Håkon Robbestad Gylterud

2019–10–14

Security through the software development cycle

The software development cycle

1. Requirements
2. Design
3. Implementation
4. Testing
5. Deployment

The software development cycle

1. Requirements
2. Design
3. Implementation
4. Testing
5. Deployment

Question: What security related activities can you think of in each phase?

Security activities

The book ("Secure and resilient. . . ") suggests:

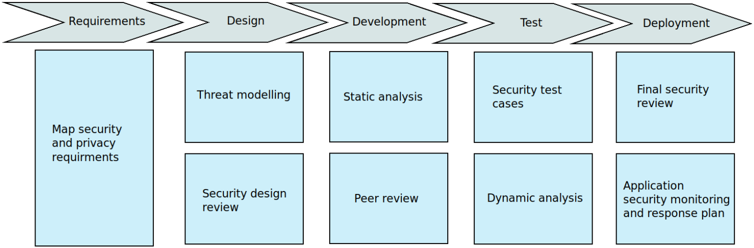

Figure 1: Security activities in the software development cyle

Software security design

Definition

Software security is the ability of software to function according to intentions in an **adverserial environment**.

Designing secure software

Figure 2: Requirements, assumptions and mechanisms

Designing secure software

Figure 2: Requirements, assumptions and mechanisms

1. **Identify security requirements** which capture the intentions for the software.
2. **Make explicit the assumptions** about the environment the software will run.
3. **Design mechanisms** which satisfy the requirements given the assumptions.

Non-functional requirements

Non-functional requirements

- Security and privacy
- Availability, capacity, performance and efficiency
- Extensiblity, maintainability, portability and scalability
- Recoverability
- Manageability and serviceablility
- Cohesion

Availability

Definition

Availability is the proportion of time a system spends in a functional state.

Question: What causes downtime for software?

Causes for downtime

- Malicious attacks
- Software bugs
- Hardware failure
- Failure of services
- Exessive usage (exhaution of scarse resources: CPU/GPU,memory,bandwidth,threads,filehandles,$\cdots$)

Question: How can we increase availabilty?

Increasing availability

- Write **secure software**.
- Not having bugs (How?)
- Redundance
- Less reliance on services
- Testing (Example: Chaos Monkey)
- Scalability

Capacity

Capacity refers to the maximum number simultaneous of users/transactions.

- What is the target capacity of the system?
- How do we determine the capacity?
- What happens if we reach the limit of capacity?

Scalability

Scalability is the ability to *increase capacity*.

Scalability

Scalability is the ability to *increase capacity*.

Need to identify: What are the bottle-necks?

Scalability

Scalability is the ability to *increase capacity*.

Need to identify: What are the bottle-necks?

Running multiple instances:

- Load balancing (example: DNS round-robin)
- Location
- Secure communication between instances
- Eventual consistency

Peformance

Performance is:

- *Responsiveness* of the software to users
- Rate of *transaction processing*

Peformance

Performance is:

- *Responsiveness* of the software to users
- Rate of *transaction processing*

This covers both **latency and throughput**.

- What is acceptable performance?
- How does performance degrade when approaching the limit of capacity?

Efficiency

Efficiency is the ability to make use of scarce resources such as:

- Memory / cache
- Processing power
- Storage
- Network bandwidth
- Latency

Increasing software efficiency gives a better performance/hardware requirement.

Maintainability & extensibility

How easy is it to **develop and deploy fixes and new features**?

Developing fixes / new features

How easy is it to maintain the code?

Developing fixes / new features

How easy is it to maintain the code?

Depends on code qualities:

- Readability
- Structural properties:
  - Isolation of concerns
  - Brittleness
- Documentation

Developing fixes / new features

How easy is it to maintain the code?

Depends on code qualities:

- Readability
- Structural properties:
  - Isolation of concerns
  - Brittleness
- Documentation

When multiple fixes/features are developed at the same time:

- Merging:
  - How often?
  - How to ensure quality?

Deploying fixes / new features

How to securely deploy a new version?

- Possible attack vector: **Malicous updates.**

Most modern distribution systems include some signature mechanisms.

Deploying fixes / new features

Does upgrading cause disruption?

- Downtime?
- Can different version coexist?
- Portability of persistent data
  - Serialization is brittle
  - Use data formats with clear specifications

Portability

Portability is the ability of the software to run on different systems with little adaptation.

- Language dependent (Assembly vs C vs Java)
- Portability favours abstractions
- Documentation

Recoverability

Recoverability is the *time to recovery from distruptive events*.

- Backups
- Failover systems (Hardware or virtual)
- Update deployment

Cohesion

Cohesion is the degree to which parts of a system/module *belong together*.

Strong cohesion: each module is **robust** and **reusable**.

Contrast with **coupling**, the interdependency between modules.

Threat model

The threat model

Figure 3: Requirements, assumptions and mechanisms

The threat model

Figure 3: Requirements, assumptions and mechanisms

1. Identify security requirements which capture the intentions for the software.
2. **Make explicit the assumptions about the environment the software will run.**
3. Design mechanisms which satisfy the requirements given the assumptions.

Threat model

What **threats** (to the defined requirements) can an attack pose? (STRIDE can be an inspiration)

Threat model

What **threats** (to the defined requirements) can an attack pose? (STRIDE can be an inspiration)

Which part of the system is likely to be **controlled by an attacker**?

Threat model

What **threats** (to the defined requirements) can an attack pose? (STRIDE can be an inspiration)

Which part of the system is likely to be **controlled by an attacker**?

What **motivates** an attacker?

Threat model

What **threats** (to the defined requirements) can an attack pose? (STRIDE can be an inspiration)

Which part of the system is likely to be **controlled by an attacker**?

What **motivates** an attacker?

What **attack vectors** can an attacker use?

Threat model

What **threats** (to the defined requirements) can an attack pose? (STRIDE can be an inspiration)

Which part of the system is likely to be **controlled by an attacker**?

What **motivates** an attacker?

What **attack vectors** can an attacker use?

In order to perform this analysis we need:

- Functional decomposistion (A diagram of software components)
- An overview of trust-relationships between components
- Good knowledge of specific security pitfalls (injection, XSS, CSRF, authentication, access control, $\cdots$)

Security review

Security review

Manual review happens in different phases:

- Security design review
- Peer review of implementation:
    - Reviewing commits
    - Pair programming
- Final security review (before deployment)

Logging

Developer error messages

Debugging/developer error messages should

- be logged to a separate, safe storage.
- be append only (enforced by storage mechanism and API)

stdout/stderr are often not good for services, because they are often redirected to surprising places.

What to log

- Authentication events
- Attempted intrusions
- Violations of invariants
- Unusual behaviour
- Performance statistics

What not to log

Not everything should go to the log:

- Sensitive information
- Keys
- Passwords
- User data

Monitoring

Monitoring

In order to respond to an ongoing threat four things must happen:

1. Detection
2. Logging
3. Monitoring
4. Response

Example

On a server, a user has an insecure password.

1. An attacker logs in and tries to run `sudo`, which the user was not permitted to run.
2. `sudo` logs the event
3. E-mail automatically sent to administator
4. Admin decides to lock the user account and resett their password

Why did this succeed?

Question: What (if any) mitigations should be taken after an event?

Securing development and deployment

Security is important during development:

- An attacker who can modify the source code can make his own back-doors.
- How can we trust third party libraries and APIs?

Muddest point

Answer at `mitt.uib.no`

Language based security

Language based security

How can the programming language help us avoid bugs?

Language based security

How can the programming language help us avoid bugs?

How should we write code to fully utilise the compiler's ability to verify our code?

Language based security

How can the programming language help us avoid bugs?

How should we write code to fully utilise the compiler's ability to verify our code?

How to make our intensions visible in the code? (How to write *what to do* rather than *how to do it*)