Session 3: Security in a Software Project
Part 1: Parts of a software project

DISCUSS: What happens in a software project?
(Hint: How do we know what to do? How do we do it? How do we know it works? How do we use it?))

DISCUSS: What sort of models exist for managing work in a software project?
(Hint: What sort of models exist for organising the work above? Ever heard of agile, Continuous Delivery, DevOps?)

- Current buzzwords are “Continuous Delivery” and “DevOps”. This means that there are no distinct phases of definition, implementation, testing, and operations, but these are integrated into an iterative model. It is not yet very common to find this model in software development, but it is gaining ground rapidly.
- Until now, we have mainly discussed what goes wrong during low-level design and implementation. However, implementation in a real software project does not happen in isolation.
- How well a software project can actually perform software security work depends heavily on how well the actual software development works.
- It is very typical for a company to start from security testing - and within testing, usually from exploratory security assessments, and move towards development-time “secure coding”.
- Security testing, however, is by nature trying to detect what has already gone wrong.
- The contribution of the various software project activities to software security:
  - Product management & requirements engineering: Avoidance of architectural and design level security issues; giving enough time and people for secure software engineering; balancing functional and non-functional security requirements.
  - “Functional” requirements are those who actually result in a feature or functionality. A password dialog, use of a security protocol, firewall configuration, and such things are “functional”.
  - “Non-functional” requirements are about how something has been implemented. This is about avoidance of exploitable weaknesses in any part of the code (not just security-related code) and qualities such as
usability and performance of security features.

○ Design: Avoidance of architectural and design level security issues, mainly through creation and updating of an attack model (what is it that we are trying to protect against) and architectural risk analysis (also called threat modelling).

○ Development: Avoidance of design and implementation level security bugs, and using security features correctly so that they actually offer the intended security control effect.

○ Building: Specifically in an automated build setup, offers automation opportunities for various security activities. The most obvious contenders are static code analysis and security testing. The build process can also act as a “gate”, where security issues can stop the delivery of code into further testing or production.

○ Testing: Verifying whether the security functionality works as expected (through positive and negative testing), and trying to determine whether there are any non-functional security issues such as robustness problems. The levels and types of testing and their contribution to a security test vary.
  ■ Unit testing: Positive and negative tests for security features. (Does a wrong password return an error code? Does an invalid input get filtered?)
  ■ System testing: Positive and negative tests for security features, and robustness testing. (Does injection of bad data cause issues downstream?)
  ■ Acceptance testing: User story / use case level checks for security features and verifying that “attacker use cases” have an expected result. Exploratory security testing. (What if an attacker tries to brute force a password? What can we see in a HTTP request on the line? Is the client actually checking certificates properly?)
  ■ Security assessments, penetration testing (pentesting) or intrusion testing: These are usually exploratory test runs that are often done by an external party due to lack of in-house expertise, or because of a regulatory or contractual requirement.

○ Deployment: Deployment automation provides a faster round-trip time for security fixes, and the deployment configuration is where “hardening” happens for the underlying platform. For applications that run on devices that the software vendor doesn’t control, this includes secure software update systems.

○ Operations: This is where we cross over to the “IT security” realm and we don’t go into much detail here, but this part involves monitoring the system for signs of an attack, and responding to them, as well as ensuring that the software dependencies, the platform, and the network stay secure, unless these are provided as a part of the deployment.

○ Bug management and support operations: From security perspective, operating a security contact point for internal and external events, tracking known vulnerabilities in dependencies and the platform, and feeding all this back into the development.

○ Decommissioning: From security perspective, data retention and erasure is an area which is notable here.
To summarise all the above: Software security is a chain of events throughout the whole “lifecycle” of software creation. The activities, as described here, support and build on each other.

Part 2: The Holistic View on SW Security

- There are a number of software security activities in a software development organisation that are not directly connected to implementation activities.
- There are several documents trying to specify what exactly would need to be done. We will shortly go through each of the examples here.
- BSIMM (“Building Security In Maturity Model”) is a project by a consultancy called Cigital.
  - Based on actually interviewing companies and trying to detect what exactly they are doing. The model gets usually an annual update.
  - Summary data is available under a permissive license and this gives a good view as to what companies are doing in reality. However, it should be noted that if a company actually takes part in BSIMM, they are likely to be “in the know” already, so I suspect that this gives a rosier picture than the reality is.
  - 12 “practice areas” containing a total of 111 “activities”, in three “maturity levels”.
- OpenSAMM, or SAMM (“Software Assurance Maturity Model”) is an OWASP project that has common, although very distant, roots with BSIMM. You can compare and contrast OpenSAMM with BSIMM activity areas.
  - OpenSAMM has been on a hiatus since the first release in 2009, but is being reinvigorated. OpenSAMM is more prescriptive as it is not based on actual research on existing companies.
- ISO 27034 is one of the newest members in the ISO 27000 series of standards.
  - The first part has been published as ISO 27034-1, and it specifies the information model for a software security program.
  - Specifies vocabulary and concepts for software security process requirements for an organisation.
  - One peculiar thing about ISO 27034-1 is that it uses Microsoft SDL as an example in its appendix.
  - My personal guess is that ISO 27034 becomes more important in the future as the more specific parts will get published; it may also affect the vocabulary. Those companies who currently require ISO 27001 (the canonical “infosec” standard) from their vendors might require ISO 27034 too at some point.
- PA-DSS (Payment Application Data Security Standard), a part of PCI-DSS (Payment Card Industry…). Any application processing (or storing or transmitting) credit card numbers needs to conform to this. PCI-DSS is a big reason why websites use third party payment processors - they don’t want to touch the card data.
- Specifically for the Finnish audience, there is a VAHTI (“Valtionhallinnon tietoturvaohje”, “Government information security guideline”) Application Security guideline from 2013. This document mainly has only national significance, but may become soon a de facto requirement for any governmental or public sector procurement.
Part 3: Visitor from the security industry
- In order to give some more insight as to what is currently happening in the software industry, we have a great opportunity to hear from a person who works in a security consultancy company.
- The visitor will give insight to what a “real” company software security activity looks like (contrasting the models above).

Part 4: In preparation for the weekly exercise - Typical tooling
- There are a lot of vendors providing automated or semiautomated tools that they hope software development projects would integrate into their lifecycle. Three major tool types with a significant market are discussed here.
- This is not intended to explain the deep details and usage modes of each tool, but only as a quick explanation of the categories and aims in order to be more productive in the weekly exercise.
- Code analysis tools
  - “Static” analysis is “static” because the code does not execute during analysis. To contrast with “dynamic” tools where the code is being executed.
  - Linters (static)
    - Linters are mainly for purely syntactic analysis. These tools are often used in editors or IDEs to highlight lines that have syntax errors, use undefined variables, or something like that. Most languages have one or more linters available.
    - Using a linter should be done by anyone who programs. They are really great productivity boosters when used in conjunction with syntax highlighting in an editor. However, linters often cannot detect security bugs.
    - Some linter-type tools may provide metrics that may have be indirectly indicative of security issues. For example, cyclomatic complexity provides a count of the linearly independent paths through a certain block of code. The argument is that if the cyclomatic complexity is too high, it becomes hard for the programmer to keep track of what is happening, giving rise to bugs - including security bugs. Simple metrics such as this should not be played down as a useful security risk measurement. They could also be used to identify parts of code that needs more manual code review or refactoring.
    - Open source examples include pylint and pyflakes for Python and JSLint for JavaScript.
  - Data flow analysis (static)
    - Static analysis tools that actually do data flow analysis can be useful in finding actual security bugs.
    - Static analysis tools can do “tainting” (see session 2) statically. They are able to track the variables and branching through the code, and they can do a series of “what if” tests. They can detect blocks of “dead” code,
meaning code that cannot be reached by any inputs. They can also detect data flows between untrusted inputs and dangerous outputs.

- Static analysis tools are usually either language-specific, or they could compile the original source into an intermediate language. Some tools work directly with the compiled binary or bytecode.
- The effectiveness of static analysis tools is highly dependent on its capacity to support the specific language, platform, and the set of frameworks and libraries that you are using.
- Open-source examples of static analysis tools include FindBugs for Java.

- Dynamic analysis
  - Dynamic analysis tools can be used to monitor the program when it is running (hence “dynamic”).
  - Dynamic tools can, for example, detect memory leaks and detect if a freed memory block is used. These are especially useful analyses if you are programming in a language that is low-level and you have to do your memory management by yourself.
  - Dynamic tools can also be used for a number of non-security-specific profiling tasks, so they are useful in performance engineering. They might be already in use in a project because of one of these reasons.
  - Open-source examples of dynamic analysis tools include Valgrind.

- Security testing tools
  - These groups are not really separate; an attack proxy can be used to “scan” a web application and to test its robustness. I am just artificially describing the three groups here - this is not any kind of officially recognised division, but useful to bear in mind.
  - In selecting what tools are going to be used, it is useful to bear in mind the totality of software security activities performed in a project. If there is a significant amount of code review being done, perhaps it would be better to start tooling from fuzzing. If you already do a lot of testing after the implementation, perhaps your best bet would be to invest in something that precedes implementation.
  - In a typical real-life case, a manager (who really has no security experience) orders a “scan” for a web application. The word really elicits thoughts about an automated activity that somehow, actively or passively, could do a comprehensive check of the target application. In most cases this is not true. Either the scan will not find any complex issues, or it is not really a scan but actually exploratory testing that is just augmented by automated and semiautomated tools.
  - Robustness testing tools
    - This group includes fuzzers and other tools that inject incorrect inputs. Fuzzers were already discussed in the first session, so please refer there for details.
    - Open-source examples include Radamsa (which you used in the first exercise), Peach, Bunny the Fuzzer, and many more.
  - Attack proxies and sniffers
This group of tools acts as a middleman for traffic, or looks at the traffic. Attack proxies were described in the second session, so you are already very familiar with the Burp Suite.

Sniffers and protocol analysers are used to look at the traffic. One really useful mode of using a protocol analyser is just to grab some traffic and have an expert look at it. It is surprising how often there are some oddities that may point to a bug. This, however, requires quite a bit of experience to see what is normal and what isn’t.

Open-source examples include Zaproxy (an attack proxy) and Wireshark (a sniffer/protocol analyser).

Security scanners

Configuration security sits on the borderline between operations security and software security. As a part of move towards DevOps (combined development and operations), operational scanners become more widespread in the development teams, too.

DEMO: If we have time, I will show a system where Burp Suite is being run non-interactively in a test automation framework. This type of optimisation is important in a change towards Continuous Delivery, where human GUI-driven manual work should be minimised.

Operational tools

Web application firewalls

WAFs try to detect anomalous / attacker traffic before it hits the target system.

Instead of patching a web application, some companies just add a “detection” to a WAF.

Virtualisation

Virtualisation allows running untrusted code in sandboxed environment.

Various instrumentation tools such as Microsoft’s EMET

EMET tries to detect certain exploitation techniques from the behaviour of the process.