6. Software Security

Jin Hong

jin.hong@uwa.edu.au

What you need today

- Kali VM
  - You will need to install OWASP DependencyCheck
  - https://github.com/jeremylong/DependencyCheck
    - `mvn -s settings.xml install -DskipTests=true`
    - `./cli/target/release/bin/dependency-check.sh --out . --scan ./src/test/resources`
  - The install takes some time, so it will be good to pre-download it.

- Things we do today
  - Software Dependency Analysis

Software Security

- Nowadays almost all applications utilize the internet for functionalities
  - Getting updates
  - Checking user requests
  - Saving states of the applications etc.
- No point drawing a line between application and web security
  - We now have web application security!
- We will look at both developing software applications issues and some examples of malicious software

Issues

- Many possible security vulnerabilities for a software system, depending on:
  - Requirements (missing, misunderstood, features not securely implemented)
  - Design
  - Implementation language/library issues
  - Subsystem interactions/issues
  - Network environment issues
  - (Lack of) testing/vulnerability analysis
  - Other factors

Software Development

- Common specifications of software development process
  - ISO 12207 – Systems and software engineering – software life cycle processes.
  - The current version is 2017*, but the list remains the same.
  - There are no explicit security related tasks.

- Business or mission analysis
- Stakeholder needs and requirements definition
- Systems/Software requirements definition
- Architecture definition
- Design definition
- System analysis
- Implementation
- Integration
- Verification
- Transition
- Validation
- Operation
- Maintenance
- Disposal

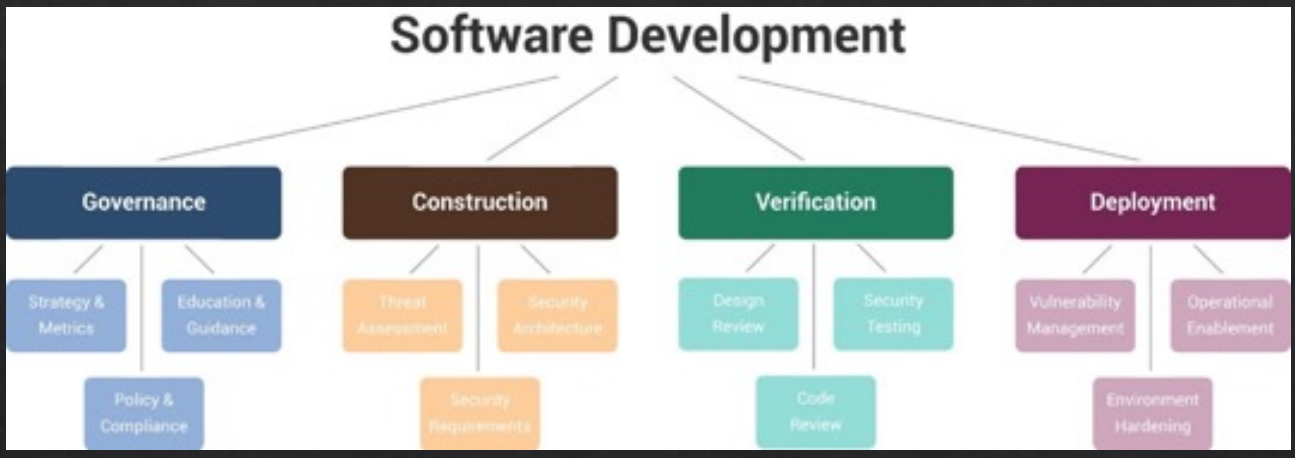
*Current as of Aug 2023

SLC - Organisation

- Many different SLC methodologies:
  - **Waterfall** – single series of steps
  - **Iterative** – multiple iterations, one feature at a time
  - **Spiral** – requirements or functions analysed for risks, developed as prototype, integrated into whole
  - **MS SDL** (MS Security Development Lifecycle) – Microsoft methodology
  - **Team Software Process** (TSP) – framework with set of processes, used with Personal Software Process (PSP) and CMMI
  - **Agile** methodologies – e.g. Test-driven development (TDD): gather user stories, generate reasonable and prioritized set of features, then cycle (per feature) of develop tests, implement, refactor design if needed
  - **Touchpoints** – from Gary McGraw
    - http://www.swsec.com/resources/touchpoints/

Other Approaches

- OWASP Software Assurance Maturity Model (SAMM)
- Building Security In Maturity Model (BSIMM)



SAMM

https://www.ptsecurity.com/ww-en/analytics/knowledge-base/how-to-approach-secure-software-development/

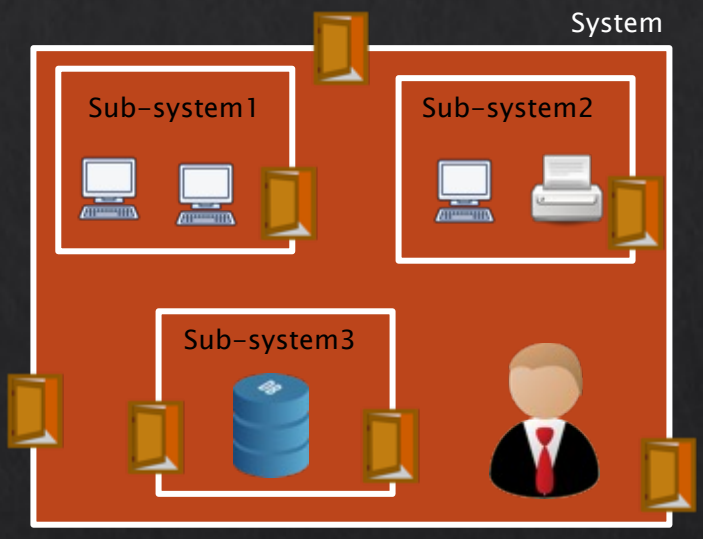
BSIMM

Security in Design – Architectural Level (1)

- Security Architectural Patterns
  - Single-Access Point Pattern
    - single point of entry into system
  - Check Point Pattern
    - centralised enforcement of authentication and authorisation
  - Role Pattern
    - disassociation of users and privileges
  - Distrustful Composition
    - decompose multi-function systems to reduce attack surface
  - Privilege Separation
    - Reduce the amount of code that works with increased privilege level
  - Defer to Kernel
    - Use existing user verification functionality within kernel to avoid reinventing decision control

```
/* Our product has
multiple layers of Security */
if (isAuthorised(user)) {
    if (isAuthorised(user)) {
        if (isAuthorised(user)) {
            access_data();
        }
    }
}
```

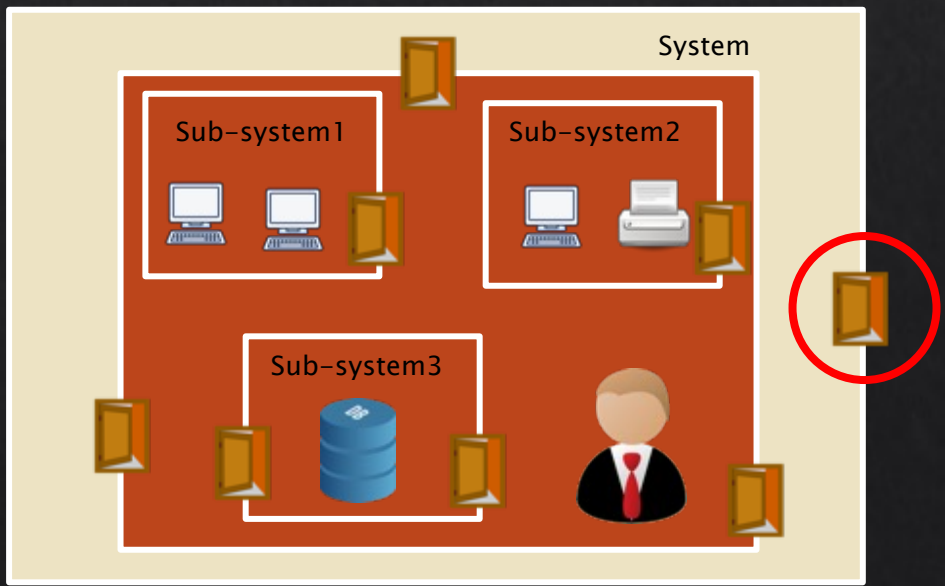
Our System



Single-Access Point Pattern

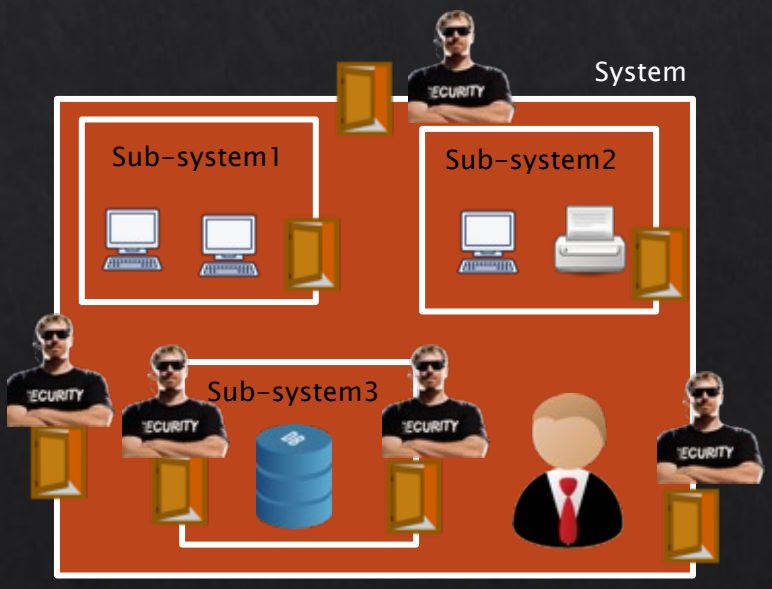
- System should have single point of entry
  - Common to all incoming requests
  - All users must pass through it
  - Any unauthorised requests of any other component of the system are directed to this point

- Example: Java Enterprise Edition (EE) – supports with declarative security implemented through web container



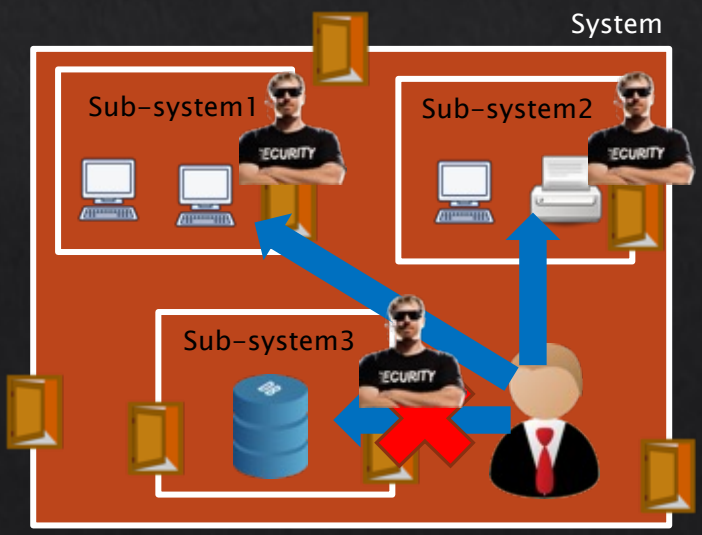
Check Point Pattern

- Centralises and enforces authentication and authorisation
- Single mechanism determines if access (or ability to grant access) to particular resource is proper
- Matches user with **level of security** allowed
- May be multiple check points
- Java EE supports with Java Authentication and Authorisation Service (JAAS)
- Pluggable Authentication Module (PAM) also supports this pattern



Role Pattern

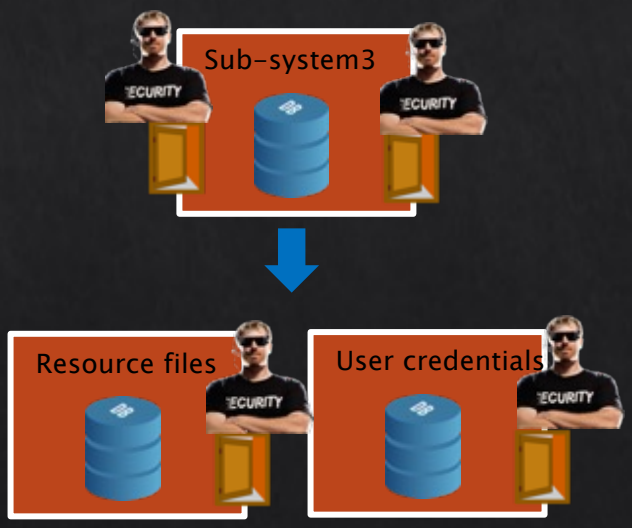
- Disassociation of users and their privileges
- Avoid an un-manageable rats' nest of combinations
- User assigned one or more roles
- Role assigned one or more privileges
- Supports easier management
- Java EE supports through declarative security model
  - Can specify in XML deployment descriptors
- Found in other large software systems (e.g. Oracle)



Distrustful Composition

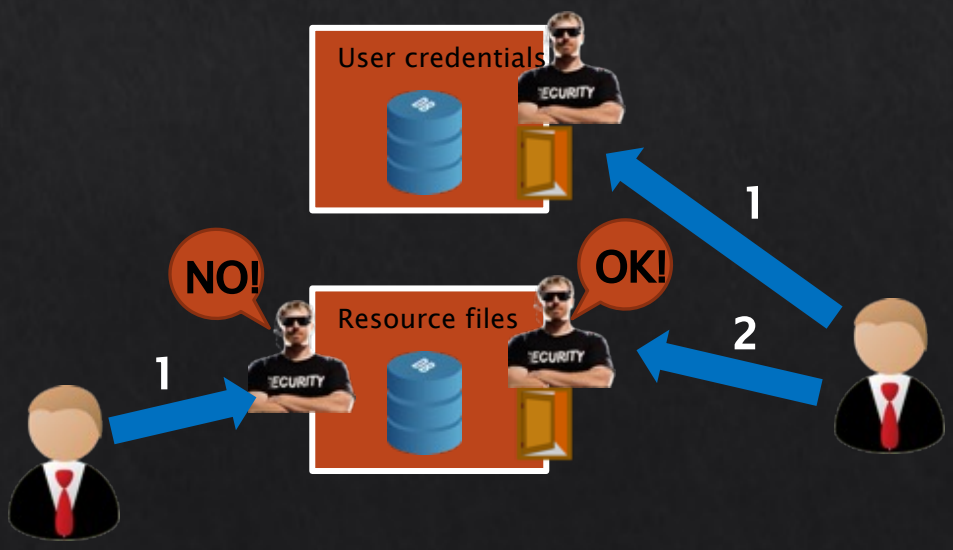
- Normally, functionality accumulates within modules, leaving large modules with multiple functionality and possibly multiple interfaces and/or attack surfaces

- Pattern: decompose multi-function systems to **reduce** the attack surface
  - Small cohesive units have less vulnerabilities by providing a smaller attack surface
  - Small, modular units can be better isolated and tested



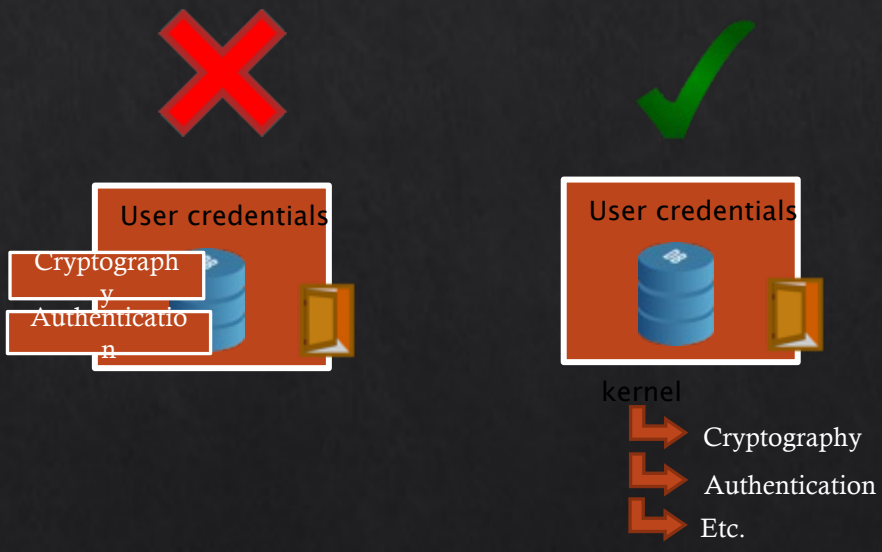
Privilege Separation

- Reduce the amount of code that works with increased privilege level
- Specific instance of Distrustful Composition
- Uses pre-authentication and post-authentication states
  - Pre-authentication: before undergoing authentication, client or responding server process are given no privilege
  - Post-authentication, privilege is introduced, but not directly given to the client
- Uses privileged parent process as monitor and less-privileged child process as worker



Defer to Kernel

- Use **existing** user verification functionalities within **kernel** to avoid reinventing decision control
  - Avoid re-implementing work that the kernel already can do
  - The kernel has proven, existing security features and has been analysed more thoroughly than new code

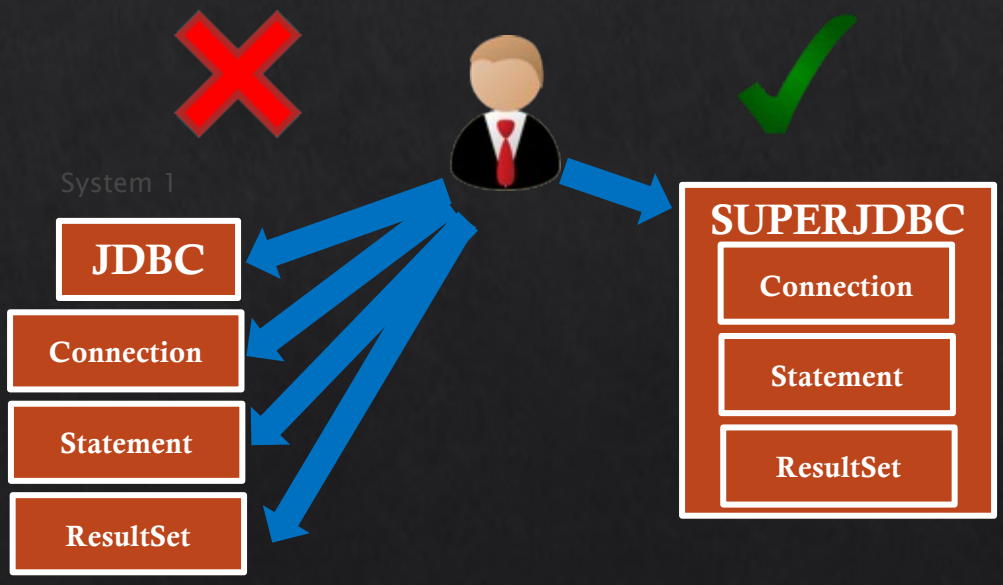


General Design Patterns Relevant to Software Security

- Two Examples:
  - Façade (fa-saad)
  - Proxy

Facade Design Pattern (1)

- Facade pattern provides an **unified** interface to a complex subsystem
  - Example: JDBC interface requires client to use three major Java classes: `Connection`, `Statement` and `ResultSet`
  - Could generate a **single** facade class, so that client can access that class, and that class deals with the actual detail work through the three classes, as well as providing an additional point of security



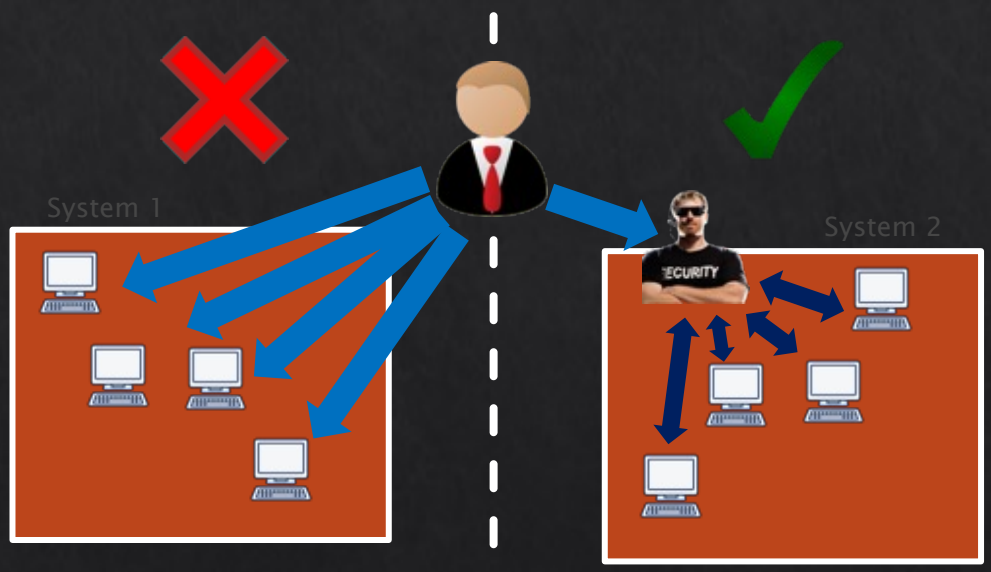
Facade Design Pattern (2)

- Not only simplifies class structure (from client view) and guides user, but limits what client can do with the complex subsystem

- Facade can help implement *Single Access Point* and *Check Point* security patterns

Proxy Design Pattern (1)

- Used to provide varying degrees of access and instantiation on demand to varying clients
- Proxy acts as a *reference* to the real object
- Many networking systems use proxy classes to protect actual server classes from direct contact and security issues
- Proxy controls access to object, may be responsible for creating/deleting it



Proxy Design Pattern (2)

- Proxy can help implement a security layer:
  - Authentication
  - Authorisation
  - Filtering requests
  - Auditing/Logging

Software Testing

- Involves good software engineering principles regarding testing
- Involves security mindset, principles, and consideration

- Two possible results of a test
  - Positive test – no problem found
    - Only validates that software works for specified test cases
    - Doesn't mean there aren't defects lurking outside of the tested areas/aspects
  - Negative test – problem found
    - One negative test result is sufficient to show that defects exist

Software Testing

- Levels of testing
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing
- Good testing done by multiple testers that are not developers for that code
  - Often independent quality assurance (QA) group
  - Exception: unit testing is done by developers

Software Testing

- **Unit Testing (Module testing)**
  - debuggers, tracers
  - done by programmers as they develop (hopefully continuously)
- Advantages
  -
  -
- Disadvantages
  -

Software Testing

- **Integration Testing**
  - communication between modules
  - start with one module, then add incrementally
- Advantages
  -
  -
- Disadvantages
  -
  -
  -

Software Testing

- **System Testing**
  - manual procedures, restart and recovery, user interface
  - real data is used
  - users involved
- Advantages
  -
  -
- Disadvantages
  -
  -

Software Testing

- **Acceptance Testing**
  - user-prepared test data
  - Verification Testing, Validation testing, Audit Testing
- Advantages
  -
  -
  -
  -
- Disadvantages
  -
  -

Software Testing

- Two testing types
  - White Box Testing
    - Assume access to source code
    - Performed by examining logic flow through code
  - Black Box Testing
    - Assume no access to source code
    - Performed by testing with expected and unexpected input, examining output and side effects
- Two aspects of security testing:
  - Testing ***security mechanisms*** in system to ensure that their functionality is properly implemented
    - Part of testing general system functionality
    - Test authentication, authorization, access control, etc.
  - Performing ***risk-based security testing***, motivated by understanding and simulating the approach of a likely attacker
    - Not considered part of general functionality testing
    - Testing misuse cases to see if they have been prevented

Software Testing

- In general:
  1. Decide how much security is needed
  2. Carry out automated testing
  3. Test at every stage
  4. Make a test plan
  5. All system components are tested

Software Dependency

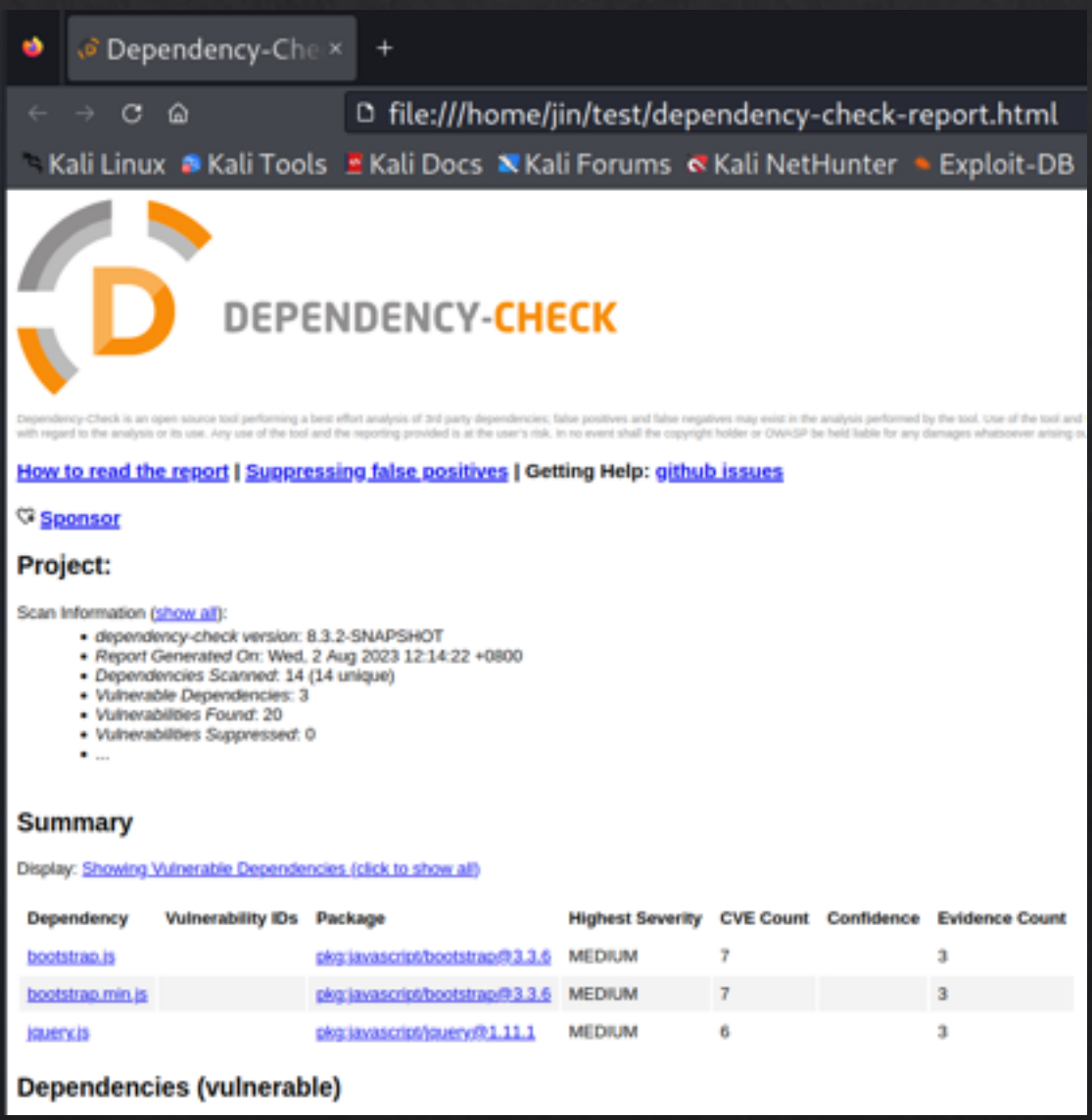
- We reuse code and APIs made available to use.
- Not many projects are built from scratch.
- Managing application dependencies is a challenging task.
- Attackers can use this information to discover vulnerabilities.

DependencyCheck

- demo

DependencyCheck

```
(base) ┌──(jin㉿kali)-[~/test]
└─$ DependencyCheck/cli/target/release/bin/dependency-check.sh --out . --scan vulnerable-node
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
[INFO] Checking for updates
[INFO] Skipping NVD check since last check was within 4 hours.
[INFO] Skipping RetireJS update since last update was within 24 hours.
[INFO] Skipping Hosted Suppressions file update since last update was within 2 hours.
[INFO] Skipping Known Exploited Vulnerabilities update check since last check was within 24 hours.
[INFO] Check for updates complete (9 ms)
[INFO]
```



Additional Items

- Agile
  - MS SDL: https://www.blackhat.com/presentations/bh-dc-10/Sullivan_Bryan/BlackHat-DC-2010-Sullivan-SDL-Agile-wp.pdf
  - SAFECODE: http://www.safecode.org/publication/SAFECode_Agile_Dev_Security0712.pdf
- US CERT: security requirements engineering
  - https://www.us-cert.gov/bsi/articles/best-practices/requirements-engineering/security-requirements-engineering
- OWASP SAMM: Software Assurance Maturity Model
  - https://www.owasp.org/index.php/OWASP_SAMM_Project
- OWASP Dependency Check
  - https://owasp.org/www-project-dependency-check/