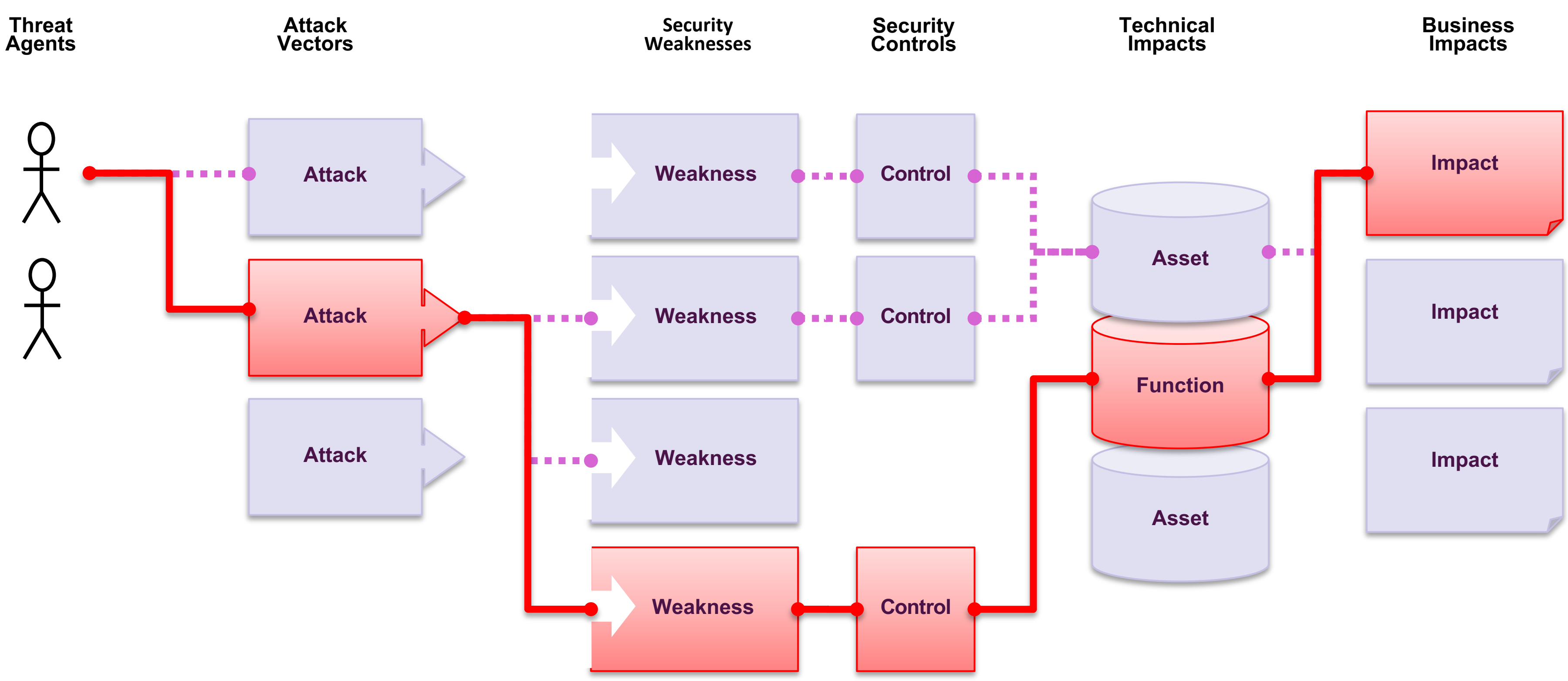
LOG 8371E
Software Quality Engineering

Lecture 09:
Software Security and Security Testing (2)

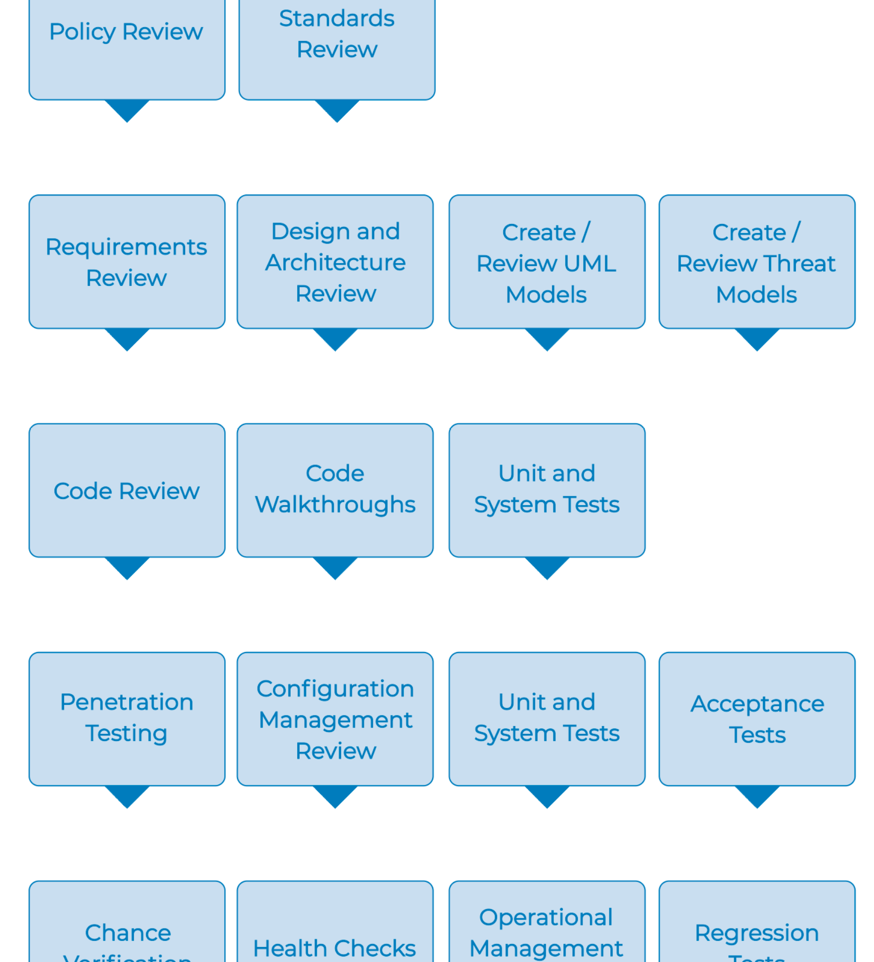
Armstrong Foundjem Ph.D. — Winter 2024

Attackers exploit the weaknesses of the application to do harm to business or organization



OWASP Testing Framework

A reference framework that comprises techniques and tasks that are appropriate at various phases of the SDLC



Penetration test vs Static analysis

Penetration test	Static analysis
▪ After the deployment. ▪ Must combine with manual efforts. ▪ Increase the certainty of the risks.	▪ During the development. ▪ Mostly automatic, but with the risk of false positives. ▪ Capable of early identification and minimization of correction costs.

Agenda

- **OWASP Top 10 security risks**
- **Fuzz testing**
- **Self-protective software**

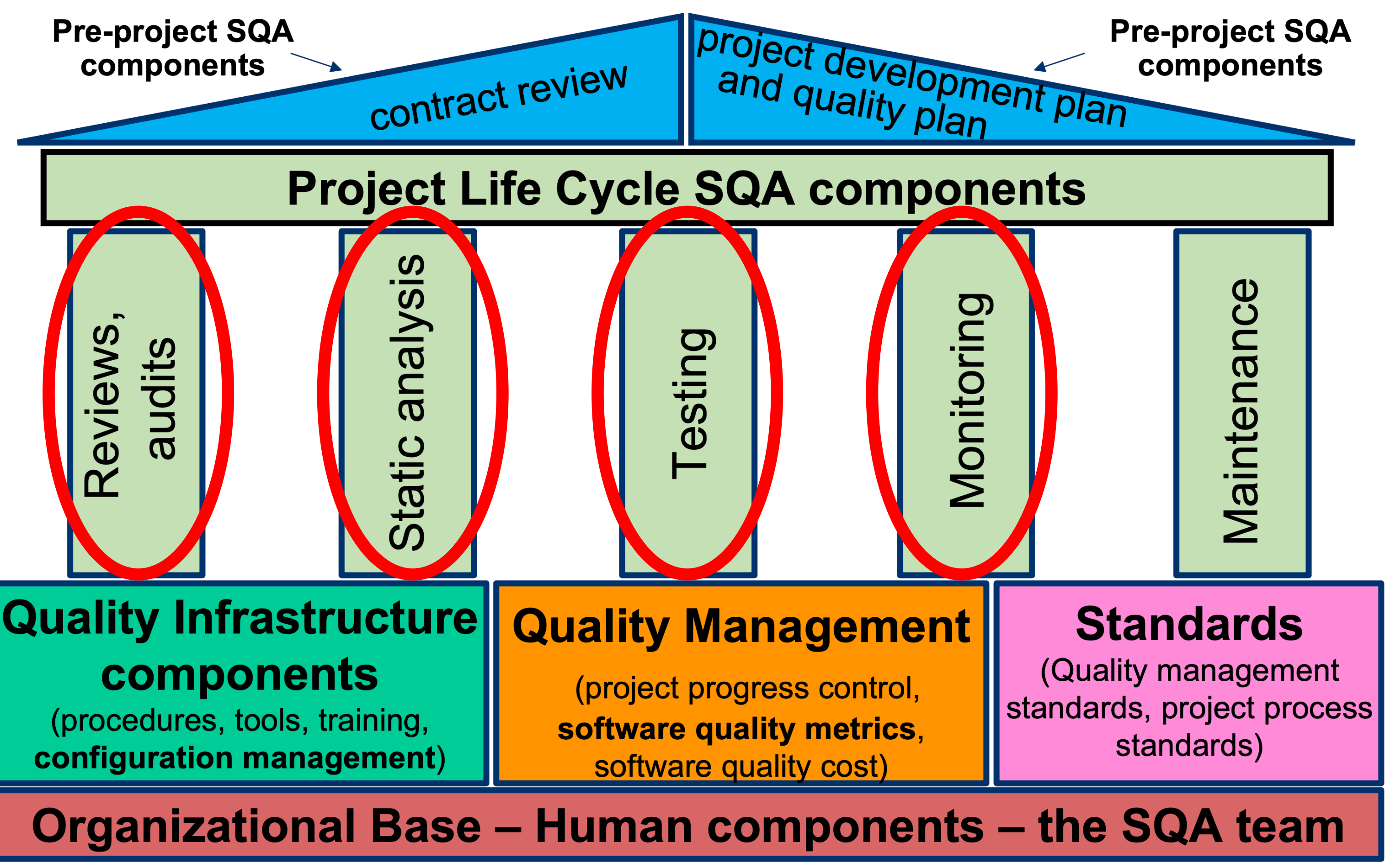
References:

OWASP Top Ten: https://owasp.org/www-project-top-ten/

Common Weakness Enumeration: https://cwe.mitre.org

Yuan, Eric, et al. *Architecture-based self-protecting software systems*. *QSA*. 2013.

Topic's topic in the SQA system



OWASP Top 10 Security Risks

Version 2021

What is OWASP Top 10

A categorization of security risks (CWEs)

A01:2021-Broken Access Control

A02:2021-Cryptographic Failures

A03:2021-Injection

A04:2021-Insecure Design

A05:2021-Security Misconfiguration

A06:2021-Vulnerable and Outdated Components

A07:2021-Identification and Authentication Failures

A08:2021-Software and Data Integrity Failures

A09:2021-Security Logging and Monitoring Failures

A10:2021-Server Side Request Forgery

Why is OWASP Top 10 important?

- To be used as a **guideline** for security requirements, secure architecture/design, security-aware implementation, testing, and deployment.

- To be used as a **standard** for assessing the security maturity of an application or benchmarking different versions.

How were the Top 10 risks made?

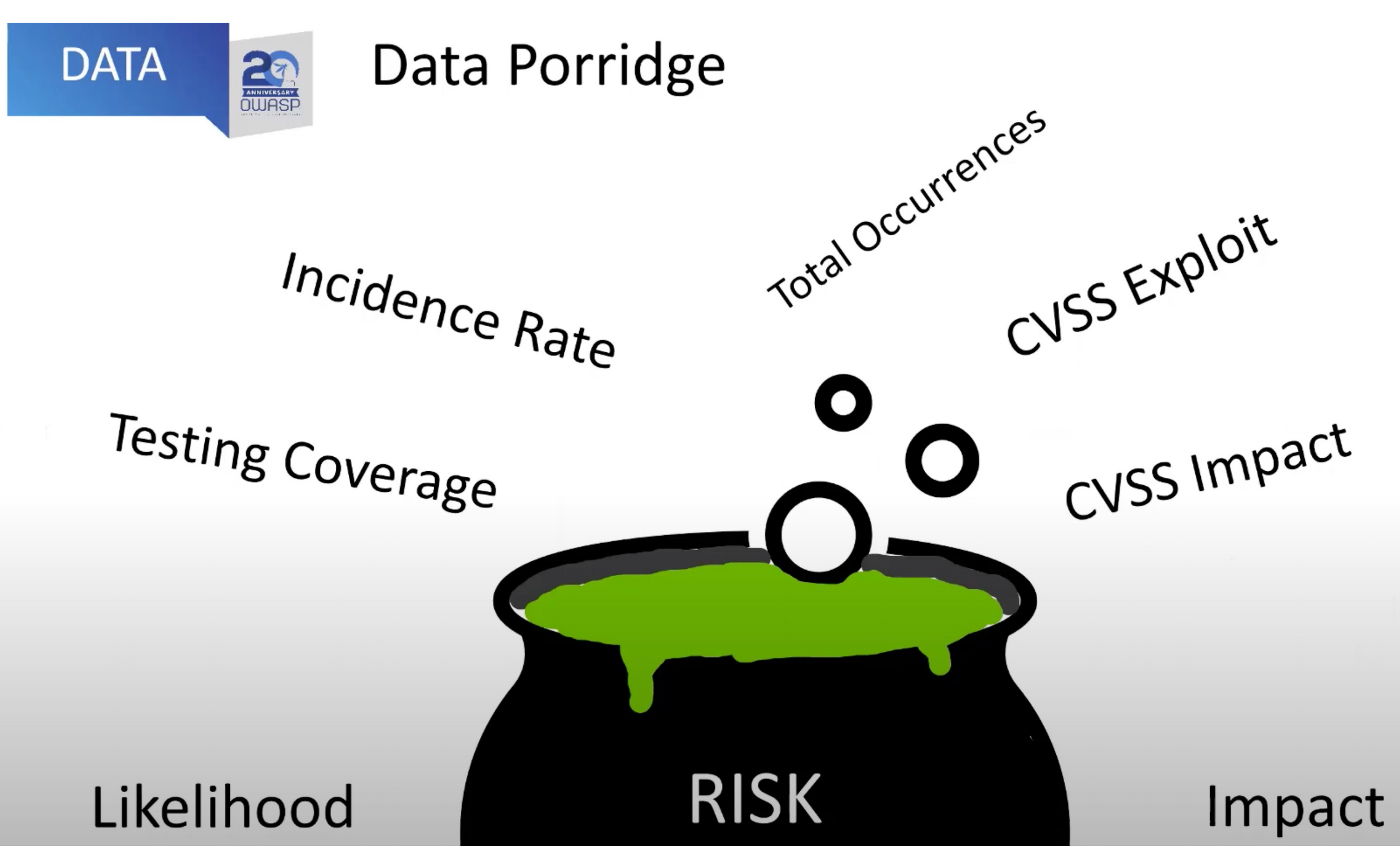
- Security data from 500,000 applications [8 risks]
  - Look into the past
  - Each contributing organization contributes a list of CWEs w/ count of applications found to contain that CWE
- Community survey [risks]
  - Essential risks that past data may not show yet
- Categorization of about 400 CWEs

How were the Top 10 risks made (Core Principles)

- OWASP Top 10 is a baseline, not a ceiling
- Data is good, data isn't everything
- Data is looking in the past, hence the community survey
- Stability is good
- Need to raise the minimum bar
- Drive the right behavior to improve software security
- Focus on root cause over symptom

Brian Glas: The making of the OWASP Top 10 and beyond, 2021.

How is the security risk level calculated?



Brian Glas: The making of the OWASP Top 10 and beyond, 2021.

How is the security risk level calculated?

RISK = Likelihood * Impact

Likelihood:

- Incident Rate -> num apps CWE found / num app CWE tested
- Coverage -> the percentage of the apps tested for the CWE

Impact:

CVSS sub-scores for Exploit and Impact

OWASP Top 10

A categorization of security risks (CWEs)

A01:2021-Broken Access Control

A02:2021-Cryptographic Failures

A03:2021-Injection

A04:2021-Insecure Design

A05:2021-Security Misconfiguration

A06:2021-Vulnerable and Outdated Components

A07:2021-Identification and Authentication Failures

A08:2021-Software and Data Integrity Failures

A09:2021-Security Logging and Monitoring Failures

A10:2021-Server Side Request Forgery

A01: Broken Access Control

- Access control failures typically lead to:
  - Bypass access control checks
  - Unauthorized access to accounts
  - Unauthorized creation, reading, updating and deletion of data
  - Elevation of privilege

- Privacy and regulatory impacts
- The biggest breaches and largest costs

34 CWEs
19k CVEs

Found in 3.8% apps
Occurred 318k times

Weighted Exploit: 6.9
Weighted Impact: 5.9

A01: Broken Access Control

Example CWEs

- ***CWE-200***: Exposure of Sensitive Information to an Unauthorized Actor,
- ***CWE-201***: Insertion of Sensitive Information Into Sent Data, and
- ***CWE-352***: Cross-Site Request Forgery

A01: Broken Access Control

Example Language: **Perl**

```
my $username=param('username');
my $password=param('password');

if (IsValidUsername($username) == 1)
{
   if (IsValidPassword($username, $password) == 1)
   {
      print "Login Successful";
   }
   else
   {
      print "Login Failed - incorrect password";
   }
}
else
{
   print "Login Failed - unknown username";
}
```

A02: Cryptographic Failure

- Determine the protection needs of data in transit and at rest:
  - Passwords, credit card numbers, health records, personal information, and business secrets;
  - Privacy laws, e.g., EU's General Data Protection Regulation (GDPR);
  - Regulations, e.g., financial data protection such as PCI Data Security Standard (PCI DSS).

- Mostly found during code reviews or static code analysis

29 CWEs	Found in 4.5% apps	Weighted Exploit: 7.3
3075 CVEs	Occurred 234k times	Weighted Impact: 6.8

A02: Cryptographic Failure

Example CWEs

- ***CWE-259:*** Use of Hard-coded Password,
- ***CWE-327:*** Broken or Risky Crypto Algorithm, and
- ***CWE-331:*** Insufficient Entropy.

A02: Cryptographic Failure

Example Language: **PHP**

```
function generateSessionID($userID){
    srand($userID);
    return rand();
}
```

A03: Injection

- Covers Cross Site Scripting (XSS) and JavaScript injection due to safer view frameworks

- Easily - but now rarely - found using tools
- Still quite exploitable

- Adopt better frameworks and more secure paved roads
- Provide observability to development teams if they use less secure alternatives
- Help by providing paved roads and gold standard support for safer frameworks

33 CWEs	Found in 3.4% apps	Weighted Exploit: 7.3
32k CVEs	Occurred 274k times	Weighted Impact: 7.2

A03: Injection

Example CWEs:

- ***CWE-79***: Cross-site Scripting,
- ***CWE-89***: SQL Injection, and
- ***CWE-73***: External Control of File Name or Path

A03: Injection

Example Language: **PHP**

```
$username = $_GET['username'];
echo '<div class="header"> Welcome, ' . $username . '</div>';
```

Attacking:

http://trustedSite.example.com/welcome.php?username=<Script Language="Javascript">alert("You've been attacked!");</Script>

A04: Insecure Design

- Broad category, but it's NOT a catch all bucket!

- Insecure design directly impacts application security
- Insecure design is easily the costliest to fix later (up to 100x)

- **Really shift left**! Earlier integration with the development and teams
- **Threat model** Where are controls needed? Are they there? Do they work?
- **Adopt better frameworks!** Create secure paved roads **with** dev teams

40 CWEs	Found in 3.0% apps	Weighted Exploit: 6.5
2691 CVEs	Occurred 262k times	Weighted Impact: 6.8

A04: Insecure Design

Example CWEs:

- ***CWE-209***: Generation of Error Message Containing Sensitive Information,
- ***CWE-256***: Unprotected Storage of Credentials,
- ***CWE-501***: Trust Boundary Violation, and
- ***CWE-522***: Insufficiently Protected Credentials.

A04: Insecure Design

Example Language: **Java**

```
usrname = request.getParameter("usrname");
if (session.getAttribute(ATTR_USR) == null) {
   session.setAttribute(ATTR_USR, usrname);
}
```

A05: Security Misconfiguration

- Cloud infrastructure as code == slight jump to A5
- Covers unhardened, misconfigured, and default configurations

- Eliminate the risk: Build "paved road" pre-hardened development and production frameworks, components, and build configurations
- Surface the risk: Build tools to identify weakly or insecurely configured components and applications

20 CWEs
789 CVEs

Found in 4.5% apps
Occurred 208k times

Weighted Exploit: 8.1
Weighted Impact: 6.6

A05: Security Misconfiguration

CWE examples

- ***CWE-16:*** Configuration, and
- ***CWE-260***: Password in Configuration File
- ***CWE-611:*** Improper Restriction of XML External Entity Reference

A05: Security Misconfiguration

Example Language: **ASP.NET**

```
...
<connectionStrings>
  <add name="ud_DEV" connectionString="connectDB=uDB; uid=db2admin; pwd=password;
  dbalias=uDB;" providerName="System.Data.Odbc" />
</connectionStrings>
...
```

A06: Vulnerable and Outdated Components

- You are likely vulnerable if:
  - you do not know the versions of all components;
  - if a component is vulnerable, unsupported, or out of date.
- Root cause of the LARGEST and MOST COSTLY breach of all time
- Recommend using CI/CD tools to warn for outdated components
- Strongly recommend breaking the build for vulnerable components

3 CWEs
0 CVEs

Found in 8.8% apps
Occurred 30k times

Weighted Exploit: 5.0
Weighted Impact: 5.0

A06: Vulnerable and Outdated Components

Example CWEs

- ***CWE-937/1035***: Using Components with Known Vulnerabilities
- ***CWE-1104***: Use of Unmaintained Third Party Components

A07: Identification and authentication failures

- Includes authentication and session management issues

- Protect against re-used, breached, and weak passwords
- Add MFA to all the things
- Use the ASVS to improve authentication of your apps
- Consider a "paved road" secured and shared authentication service

22 CWEs
3897 CVEs

Found in 2.6% apps
Occurred 132k times

Weighted Exploit: 7.4
Weighted Impact: 6.5

A07: Identification and authentication failures

Example CWEs:

- ***CWE-297***: Improper Validation of Certificate with Host Mismatch,
- ***CWE-287***: Improper Authentication, and
- ***CWE-384***: Session Fixation.

A07: Identification and authentication failures

Example Language: **C**

```
cert = SSL_get_peer_certificate(ssl);
if (cert && (SSL_get_verify_result(ssl)==X509_V_OK)) {

   // do secret things
}
```

A08: Software and Data Integrity Failures

- Integrity of business or privacy critical data
- Lack of integrity of includes from content data networks
- Software updates without integrity
- CI/CD pipelines without check in or build checks, unsigned output

- Improve the integrity of the build process
- Use SBOM to identify authentic builds and updates
- Use sub-resource integrity if using CDN for web page includes
- Consider how you vet and ensure npm, maven, repos are legit

10 CWEs	Found in 2.0% apps	Weighted Exploit: 6.9
1152 CVEs	Occurred 47.9k times	Weighted Impact: 7.9

A08: Software and Data Integrity Failures

Example CWEs

- ***CWE-829***: Inclusion of Functionality from Untrusted Control Sphere,
- ***CWE-494***: Download of Code Without Integrity Check, and
- ***CWE-502***: Deserialization of Untrusted Data.

A08: Software and Data Integrity Failures

Example Language: **HTML** *(bad code)*

```html
<div class="header"> Welcome!
  <div id="loginBox">Please Login:
    <form id ="loginForm" name="loginForm" action="login.php" method="post">
    Username: <input type="text" name="username" />
    <br/>
    Password: <input type="password" name="password" />
    <input type="submit" value="Login" />
    </form>
  </div>
  <div id="WeatherWidget">
    <script type="text/javascript"
    src="externalDomain.example.com/weatherwidget.js"></script>
  </div>
</div>
```

A09: Security Logging and Monitoring Failures

- Without sufficient logging and monitoring, breaches cannot be detected effectively
- Critical to reduce the breach window, response time, and cleanup
- Necessary if you have breach disclosure laws
- Critical if you intend to prosecute

- Interview or code review the best review technique
- Static code analysis can't find the absence
- Still difficult to dynamically test

4 CWEs	Found in 6.5% apps	Weighted Exploit: 6.9
242 CVEs	Occurred 53.6k times	Weighted Impact: 5.0

A09: Security Logging and Monitoring Failures

Example CWEs:

- ***CWE-778***: Insufficient Logging,
- ***CWE-117***: Improper Output Neutralization for Logs,
- ***CWE-223***: Omission of Security-relevant Information, and
- ***CWE-532***: Insertion of Sensitive Information into Log File.

A09: Security Logging and Monitoring Failures

Example Language: **Java**

```
if LoginUser(){
   // Login successful
   RunProgram();

} else {
   // Login unsuccessful
   LoginRetry();

}
```

A10: Server-Side Request Forgery (SSRF)

- SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL

- Frameworks need to protect against SSRF by default
- IDEs (and frameworks though *doc) need to highlight potential SSRF

1 CWEs 385 CVEs	Found in 2.7% apps Occurred 9.5k times	Weighted Exploit: 8.2 Weighted Impact: 6.7

A10: Server-Side Request Forgery (SSRF)

CWEs:

- ***CWE-918***: Server-side request forgery (SSRF)

Fuzz Testing

Fuzzing

- ***Fuzz testing*** or ***Fuzzing***:
  - a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion.
- A ***fuzzer*** is a program which injects automatically semi-random data into a program/stack and detect bugs.

Fuzzing example

Consider an integer in a program, which stores the result of a user's choice between 3 questions. When the user picks one, the choice will be 0, 1 or 2,which makes three normal cases.

But what if we transmit 3, or 255 ? We can, because integers are stored a static size variable. If the default switch case hasn't been implemented securely, the program may crash and lead to "classical" security issues: (un)exploitable buffer overflows, DoS, …

Fuzz vectors (known-to-be-dangerous values)

A common approach to fuzzing is to define lists of "known-to-be-dangerous values" (fuzz vectors) for each data type, and to inject them or recombinations.

- for integers: zero, possibly negative or very big numbers
- for chars: escaped, interpretable characters / instructions (ex: For SQL Requests, quotes / commands…)
- for binary: random ones

Fuzzer categorization

- Generation-based: inputs are generated from scratch
- Mutation-based: by modifying existing inputs
- Dumb or smart depending on whether it is aware of input structure
- White-, grey-, or black-box, depending on whether it is aware of program structure.

Goal

1. Find real bugs
2. Reduce the number of false positives
    a. Generate reasonable input
    b. If we're expecting a string, passing a file will be rejected before it even makes it to our code

Generated Input

1. Generate completely random input
   a. Don't necessarily control the input type
   b. "Milk, 3.99" -> 9620
2. Understand the input type
   a. "Milk, 3.99" -> (is a string) -> %&$#"
3. Understand the input structure
   a. "Milk, 3.99" -> '\w+, \d\.\d\d' -> "HFSDMEX, 8.43"
4. Formal approaches
   a. Model-, Grammar-, Protocol-based fuzz
   b. Useful when problem is well structured
   c. Often impractical for large realworld programs

Mutation Fuzzing

1. Take existing input
2. Randomly modify it
3. Pass it to the program

Examples

1. A set of image files that will be randomly mutated
2. A set of logged input that will be randomly modified
   a. "Milk, 3.99" -> "Gilk, 2.99"

Problems

- An overwhelming number of false positives
  - False positives are very expensive as they require manual effort
  - You put garbage in, what did you expect?
- Focus on code coverage
  - Especially the formal method approaches
  - Coverage is less important than reasonable inputs
- Cleaning
  - Sanitizer: make the random input more reasonable
  - Minimization: eliminate redundant test failures through diffing
  - Triage: finding similar outputs/stackdumps and grouping them in the same bug report

Fuzz Summary

- Test with reasonable random input
  - Goal: find real bugs
  - Problem: most failures are false positives that are expensive
- Used in practice
  - Google, Microsoft, Apple, etc use it especially in well specified/controlled environments
- Should you use it?
  - At a basic level it's simple to add
    - Example: instead of testing the same int, test a random int in a range
- Why generate random input if you have real logged input?
  - Use logged input that caused field failures
  - Turn this into a test case

Fuzzing tools

Open Source Mutational Fuzzers

- american fuzzy lop
- Radamsa - a flock of fuzzers
- APIFuzzer - fuzz test without coding
- Jazzer - fuzzing for the JVM
- ForAllSecure Mayhem for API

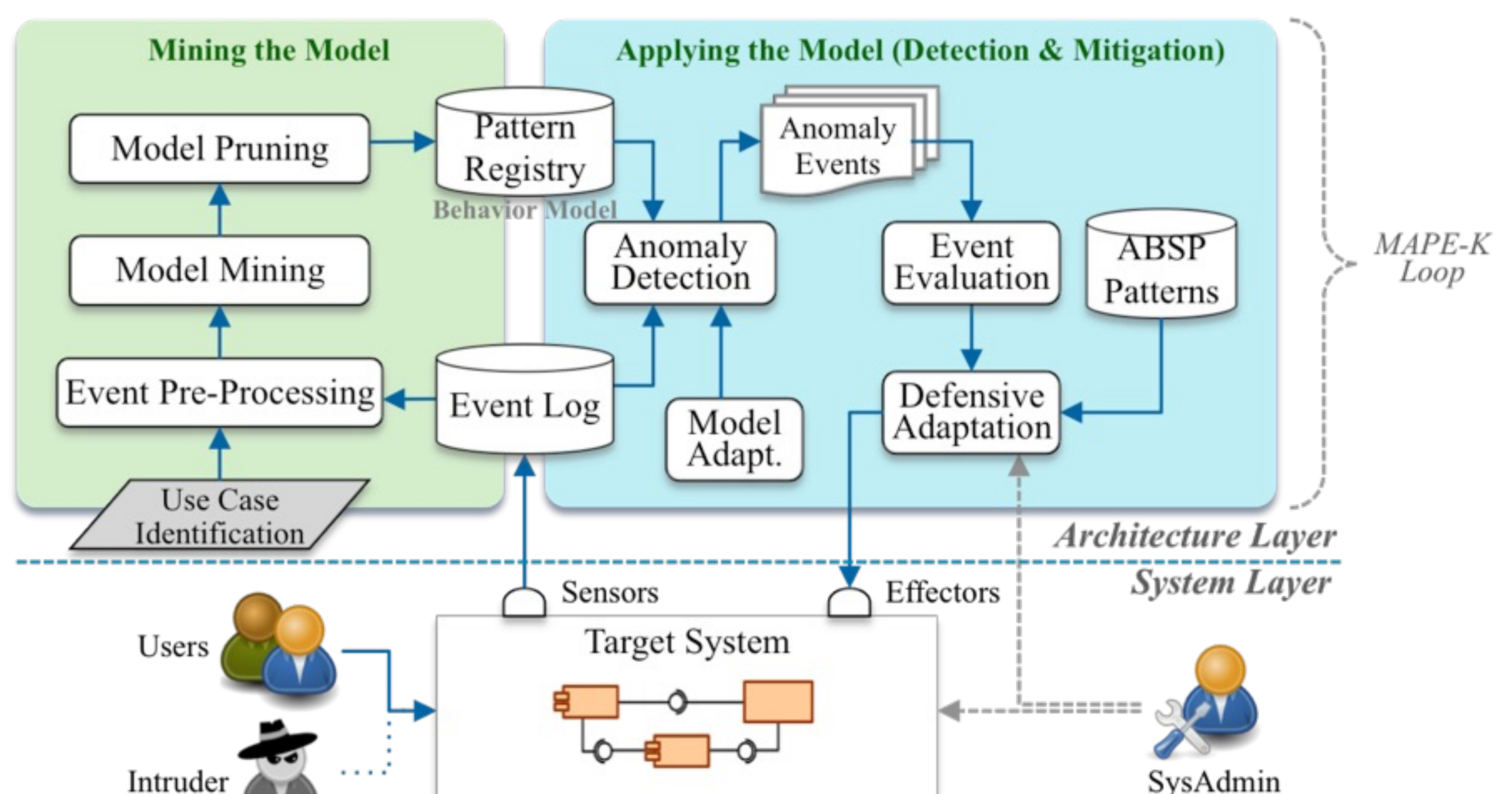
Self-protective software

Self-protective software

- Software security faces some challenges:
  - ✓ Almost every effort to ensure security is concentrated during development and just before deployment, and the software remains unprotected while running (except its periphery).
  - ✓ Static and penetration testing is controlled, but the use (or missuse) of the software is not always expected.
- To combat these challenges, researchers have suggested that software defends itself.
- As in the case of performance, where we have self-adaptive systems, which can adapt their infrastructure to better manage their performance, in case of security, we have self-protective systems.
- Generally, we talk about the self-protection of the application during its execution (Runtime Application Self-Protection (RASP)).
- Methods of self-protection at the architectural level have also been proposed (Architecture Based Self-Protection (ABSP)).

Self-protection

- RASP is implemented by inserting instructions into the application code to monitor and protect it.
  - ✓ As in the case of instrumented profiling.
  - ✓ Alternatively, one can develop an external system that is attached to the application.
- The system monitor the application to recognize malicious activities and finally protect the application by blocking the attack.



Self-protection : Advantages and disadvantages

Advantages

- RASP has increased accuracy.
  - ✓ The analyzed information is alive and real.
  - ✓ Sometimes attacks are repeated, so once detected, one can detect all future attacks of the same nature.
- RASP not only detects vulnerabilities or attacks, but can also block them.
  - ✓ E.g., It can block an IP address that has tried a lot of requests (DoS attack).
- RASP can defend the application against both external and internal threats.

Disadvantages

- RASP can affect the performance of the application that it protects.
  - ✓ Additional analysis may delay response.
- Protective actions can warn the attacker.
  - ✓ The attacker knows that it is detected and it can modify his attack strategy.
- RASP is as good as the expertise and experience of security professionals.

How can we prevent or detect the OWASP Top 10 security issues?

A01:2021-Broken Access Control

A02:2021-Cryptographic Failures

A03:2021-Injection

A04:2021-Insecure Design

A05:2021-Security Misconfiguration

A06:2021-Vulnerable and Outdated Components

A07:2021-Identification and Authentication Failures

A08:2021-Software and Data Integrity Failures

A09:2021-Security Logging and Monitoring Failures

A10:2021-Server Side Request Forgery