



Security in web development

Relatore: Roberto Santini



A brief introduction to OWASP



What is OWASP?

OWASP stands for **O**pen **W**eb **A**pplication **S**ecurity **P**roject.

It's not a development standard, it's a nonprofit foundation that works to improve the security of software.

How it try to reach its goal?

Creating projects for guidelines, tools and methodologies.



A lot of projects!

There are three categories of projects



Flagship Projects: Flagship designation is given to projects that have demonstrated strategic value to OWASP and application security as a whole



Lab Projects: Represent projects that have produced an OWASP reviewed deliverable of value



Incubator Projects: Represent the experimental playground where projects are still being fleshed out, ideas are still being proven, and development is still underway

The Top Ten Project

From the project's web page: ***“Globally recognized by developers as the first step towards more secure coding.”***

It's a document which describes the ten major application security risks.

Every web application SHOULD follow the “Top Ten” guidelines.



The Top Ten Project - the list

```
SELECT application_security_risk FROM top_10_list ORDER BY risk DESC;
```

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities (XXE)
5. Broken Access Control
6. Security Misconfiguration
7. Cross-Site Scripting XSS
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging & Monitoring

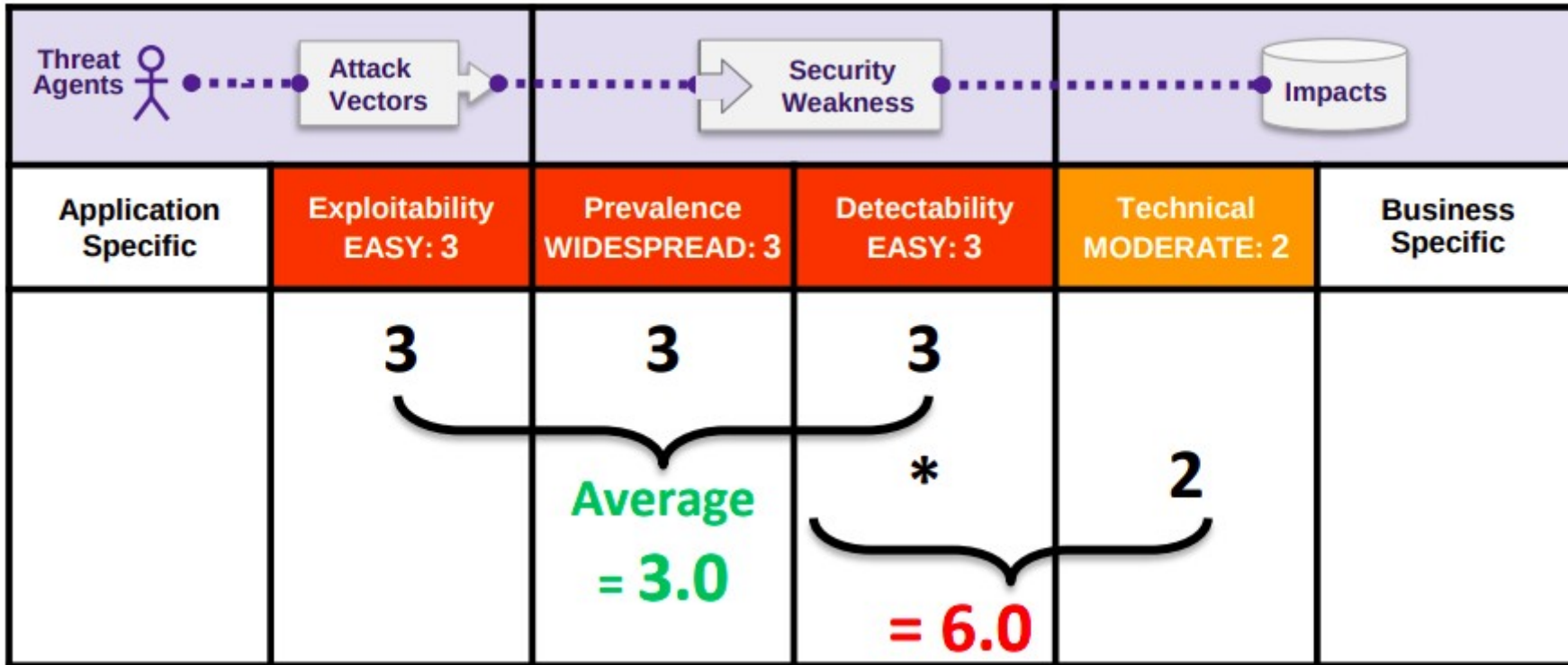


The Top Ten Project – risk classification



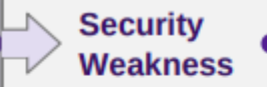
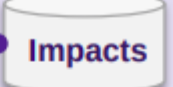
Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Application Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	



The Top Ten Project – risk calculation



The Top Ten Project – injection

 Threat Agents		 Attack Vectors		 Security Weakness		 Impacts	
App. Specific	Exploitability: 3	Prevalence: 2	Detectability: 3	Technical: 3	Business ?		
<p>Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.</p>		<p>Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries.</p> <p>Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.</p>		<p>Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover.</p> <p>The business impact depends on the needs of the application and data.</p>			

The Top Ten Project – injection

Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source ([SAST](#)) and dynamic application test ([DAST](#)) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).
Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.



The Top Ten Project – injection

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following **vulnerable** SQL call:

```
String query = "SELECT * FROM accounts WHERE  
custID=" + request.getParameter("id") + "";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts  
WHERE custID=" + request.getParameter("id") + "");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: ' or '1'='1. For example:

```
http://example.com/app/accountView?id=' or '1'='1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data, or even invoke stored procedures.

References

OWASP

- [OWASP Proactive Controls: Parameterize Queries](#)
- [OWASP ASVS: V5 Input Validation and Encoding](#)
- [OWASP Testing Guide: SQL Injection, Command Injection, ORM injection](#)
- [OWASP Cheat Sheet: Injection Prevention](#)
- [OWASP Cheat Sheet: SQL Injection Prevention](#)
- [OWASP Cheat Sheet: Injection Prevention in Java](#)
- [OWASP Cheat Sheet: Query Parameterization](#)
- [OWASP Automated Threats to Web Applications – OAT-014](#)

External

- [CWE-77: Command Injection](#)
- [CWE-89: SQL Injection](#)
- [CWE-564: Hibernate Injection](#)
- [CWE-917: Expression Language Injection](#)
- [PortSwigger: Server-side template injection](#)



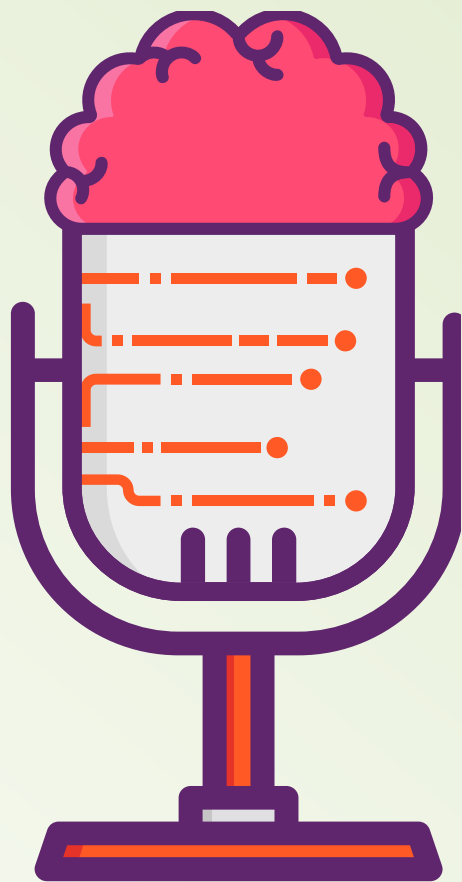
Is it enough to follow the Top Ten Project guidelines?

The answer is... NO!



It is the minimum wage, to achieve the AgID qualification the road is still long!





SonarQube



SonarQube – code analysis overview

2  Bugs

Reliability 

11  Vulnerabilities

Security 

49  Security Hotspots 

 0.0% Reviewed

Security Review 


111d Debt

2.3k  Code Smells

Maintainability 

 **0.0%**
Coverage on **69k** Lines to cover

-
Unit Tests

 **9.4%**
Duplications on **172k** Lines

1.1k
Duplicated Blocks

SonarQube – OWASP Top Ten security report

Categories ?

🔒 Security Vulnerabilities

🛡️ Security Hotspots

A1 - Injection ?	2 E	0 A
A2 - Broken Authentication ?	0 A	2 E
A3 - Sensitive Data Exposure ?	8 D	47 E
A4 - XML External Entities (XXE) ?	-	-
A5 - Broken Access Control ?	1 E	0 A
A6 - Security Misconfiguration ?	8 D	13 E
A7 - Cross-Site Scripting (XSS) ?	0 A	0 A
A8 - Insecure Deserialization ?	0 A	0 A
A9 - Using Components with Known Vulnerabilities ?	-	-
A10 - Insufficient Logging & Monitoring ?	0 A	0 A

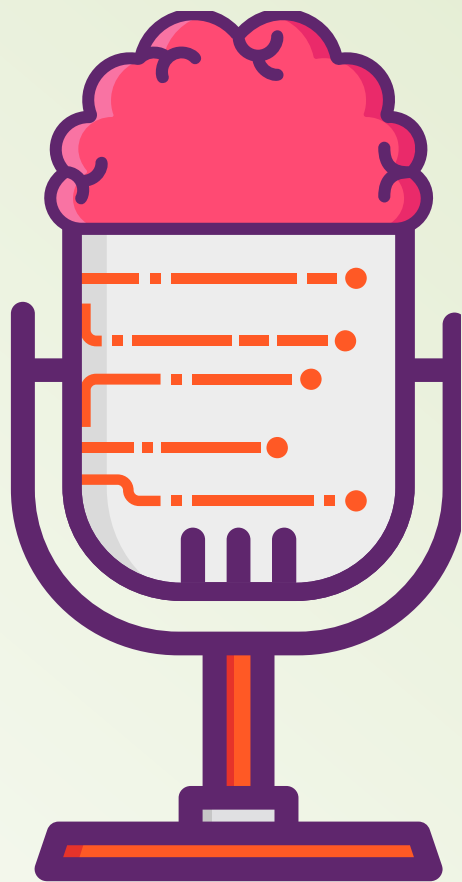
SonarQube – SQL injection vulnerability(?)

```
157 sant... public function listAlarmClockInstanceAction(Request $request): Response
158 {
159     $this->denyAccessUnlessGranted(Permission::ACCESS_LEVEL_LIST, Action::ALARM_CLOCK_OPERATION);
160
161     $filterExten = $request->query->get('exten');
162     2 $filterStatus = 1 $request->query->get('status');
163
164     /** @var EntityManager $campaignInstanceStatusEm */
165     $campaignInstanceStatusEm = $this->getDoctrine()->getManager('call_campaign_status');
166     $qb = $campaignInstanceStatusEm->getRepository(CampaignInstance::class)->createQueryBuilder('ci');
167     $qb->where('ci.type = :alarmClockType')
168         ->setParameter('alarmClockType', Consts::CALL_CAMPAIGN_TYPE_ALARM_CLOCK);
169     if (null !== $filterStatus) {
170         4 $statusList = 3 explode(',', $filterStatus);
171         if (1 === count($statusList)) {
172             $qb->andWhere('ci.status = :status')
173                 ->setParameter('status', intval(array_shift($statusList)), Type::INTEGER);
174         } else {
175             // A list of status has been required. Cast to integer the status values.
176             array_walk(
177                 $statusList,
178                 static function (&$item, $key) {
179                     $item = intval($item);
180                 }
181             );
182             $qb->andWhere( 5 $qb->expr()->in('ci.status', $statusList));
```

Change this code to not construct SQL queries directly from user-controlled data. Why is this an issue? last year L182

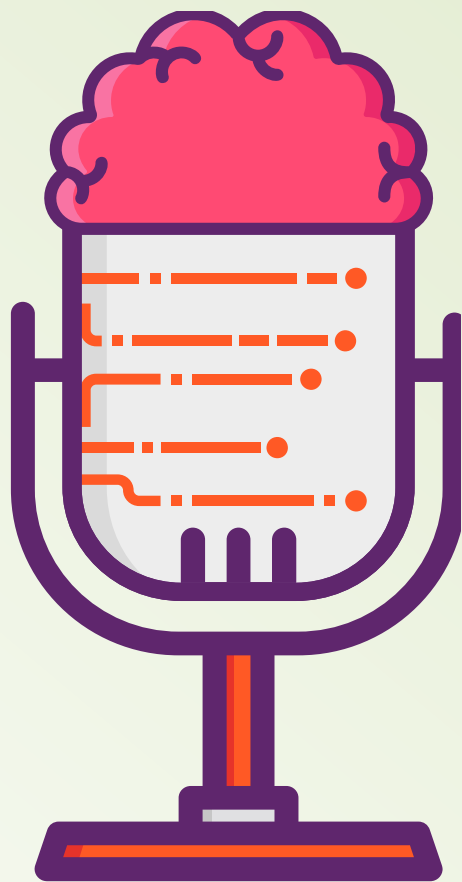
Vulnerability Blocker Open Not assigned 30min effort Comment cwe, owasp-a1, sans-top25-insecure, sql





References

- <https://owasp.org/>
- <https://owasp.org/www-project-top-ten/>
- <https://www.sonarqube.org/features/security/owasp/>



That's all guys!

Any questions?...

go to **<https://owasp.org/>**

