The OWASP Top Ten for the Uninitiated. Putting it into practice.

CONTENTS

INTRODUCTION

Understanding the implications of the most commonly known web application attacks is crucial in securing your software. Here, we explain the vulnerabilities and some methods, with examples, by which you can avoid them.

1. INJECTION

An injection attack is the most commonly used method of attack by hackers when trying to access data. The first thing you should do is assume that all data that will be entered is malicious in nature. By starting here, you can build all the necessary checks and filters to prevent these types of attacks. For example, with an SQL injection, you can use input validation functions to prevent incorrect characters from being passed and only allow characters associated with the input type e.g. email, to be passed. Other things to do would include not allowing for the use of dynamic SQL as you are handing power over to the input variables. Many databases and OS constantly require security patches and so having a patch management process in place can often be crucial. Don't forget, always keep your database credentials separate and encrypted.

EXAMPLE OF AN ATTACK.

The first example below uses SQL query preparation steps to demonstrate a very common vulnerability namely SQL injection. The example indicates that the short program has no defensive action taken before sending the user input to the database when a user could insert some malicious SQL strings, which could then cause Remote Code Execution when the input is submitted to the database.

```
Java-SQL:

username = request.getArguments().getField("userName");

sql = "SELECT * FROM user WHERE username = '" + username + "'";


Java:

String userInput = request.getArguments().getField("userValue");

String basicResponse = "<div data-value='" + userInput +"'> Click to upload </div>";
```

The second Java example is merely combining the user input with an HTML template into the response. In this case, the dangerous operation is in the second line where user input is directly placed into the result without any form of sanitisation. To prevent this from happening, one could use input validation libraries to perform a check before combining the input with the template.

2. BROKEN AUTHENTICATION

You can help prevent broken authentication attacks by carefully deciding your default login credentials mode and make sure that after the initial setup, the default password is not used for further authentication.

EXAMPLE OF AN ATTACK.

The example below directly uses MD5 to perform the checking of a password instead of salting it properly. This could mean that if any user of the system happens to use a password that is used by someone else, there would be a high chance that those who have administrative control of the system could inspect that result and could then extract the user's password.

Java:

```
SysUser user = system.getUserByName(name);

if (MD5hash(user.getPassword()).equals(userInputPasswordHash)) {

  return true;

} else {

  return false;

}
```

3. SENSITIVE DATA EXPOSURE

Sensitive data exposure occurs when customer private data or company confidential information is inadvertently exposed due to inadequate protection. To prevent this, as a developer you must ensure proper usage of secure cryptographic algorithms, safe storage of secret keys and transport security. It is crucial to identify any occurrences of missing cryptographic protection.

EXAMPLE OF AN ATTACK.

The example below directly uses MD5 to perform the checking of a password instead of salting it properly. This could mean that if any user of the system happens to use a password that is used by someone else, there would be a high chance that those who have administrative control of the system could inspect that result and could then extract the user's password.

```
Java:

System.out.println("User password : " + user.getPassword());

Logger.info("User ID { } Name = {}, Password = {} ", user.getId(),

user.getName(), user.getPass());
```

Other dangers can occur when a developer tries to save the server's private key or access keys to the log file or doing so obscurely. For example, when logging the environment variables, which may cause the server's identity to be faked, this breaks the confidentiality of the communication between users and servers.

4. XML EXTERNAL ENTITIES (XXE)

The safest way to prevent XXE is to always disable External Entities completely. Disabling these also makes the parser secure against denial of services (DOS) attacks such as Billion Laughs CVE-2019-5442 (https://nvd.nist.gov/vuln/detail/CVE-2019-5442) which is specifically aimed at parsers of XML documents. In October 2019, the Kubernetes API server GitHub repository by StackRox was discovered to have a security issue through its deployment of YAML that makes it vulnerable to the billion laughs attack for DoS. StackRox themselves stated, "The issue once again serves as a reminder that, like all software, Kubernetes is vulnerable to zero-day exploits".

EXAMPLE OF AN ATTACK.

This Java example uses the XMLReader library which directly parses this XML file with underlying XML libraries. During this operation, the external entity will be resolved and retrieved i.e. the common.xml, which could be a shared XML file for multiple users of the system and may be controlled by some external user of the system. If a malicious user meticulously crafts some content in that file, it would cause the downstream system to have its configurations endangered.

```
XML:

Base.xml:

    <DOCTYPE ...>

    <ENTITY1>

    <...> <EXTERNAL foo SYSTEM "file:///usr/local/shared/common.xml"></EXTERNAL> </...>

    </ENTIY1>


Java:

xmlObject = XMLReader.parseEntityFromFile("base.xml");

xmlObject.getChild("ENTITY1");
```

5. BROKEN ACCESS CONTROL

To avoid incidents of broken access control, it is essential to choose and stick to one access control model for your application throughout development and to continuously test it to ensure few points of failure. The four standard access control models include Mandatory Access Control (MAC), Role-Based Access Control (RBAC), Discretionary Access Control (DAC), and Rule-Based Access Control (RBAC or RB-RBAC). Each model has positives and negatives and must be selected carefully for your system design and purpose.

EXAMPLE OF AN ATTACK.

Partially checking access controls (black-list mode) is risky. In case the developer adds some other pages but forgets to update the access control logic, there could be a viable breach where users without proper access rights could still visit those pages and perform malicious actions.

```
Java:


if (pageName.equals("/home/admin/landing")) {

 checkUserAccessControls ();

}


// The access control are left unchecked from some other admin pages.
```

6. SECURITY MISCONFIGURATION

Something that you should recognise is that misconfigurations are usually to do with human error rather than a weakness or an attack. To avoid this, you can design and use a micro-segmentation policy to ensure you are protected in case of a breach, limiting the attack surface if misconfigurations go unresolved, or if patch management is delayed.

EXAMPLE OF AN ATTACK.

This example shows how a backdoor, intended for internal use, is switched on whenever some specific parameter is given in a login request. This is quite dangerous given that user could mount a service path traversal attack on the program logic.

```
Java:

userInput = request.getArguments().getField("extraParams");

if (userInput.equals("debug=on")) {

  return debugInfo;

}
```

7. CROSS SITE SCRIPTING (XSS)

Cross-site scripting can be prevented by carefully displaying and storing user input which could be displayed to other users. User input can also be run through an HTML sanitisation engine that can check for XSS code. Cross-site Scripting is a form of code injection.

EXAMPLE OF AN ATTACK.

This example demonstrates how a cross-site scripting bug deals with user input, especially, with a short piece of text that is intended to be displayed to others. Since the text (in the Java code) will be directly appended to the HTML, there is room left for an attacker to abuse this and inject arbitrary code that would run on other users' browser. The <script> here is a simplified demonstration.

```
Java:

...

return "<html> ... " + userComment.getBody() + "</html>"

...


Where user input included:


<script> ... ... </script>
```

8. INSECURE DESERIALISATION

Preventing insecure deserialisation can be achieved by using encryption, monitoring the deserialization process and not accepting serialised objects from unknown sources. Use of RASP technology is also commonly used to cover insecure deserialisation problems for web applications.

EXAMPLE OF AN ATTACK.

This example shows a popular way of sending IPC messages in Android which could be dangerous in terms of deserialising an object (Intent) that could be sent from a malicious App in a users' environment. This could cause Remote code execution if that intent has some hidden code behind the normal functions e.g. Intent.getStringExtra below.

```
Java:


response.write(user.serialize()); // Sending the serialized object to the user

...

user = MyUserImplementation.deserialize(request.getArguments().getField("userInfo"));

  // Retrieve the serialized object from the user.

...


Android Java:


Intent intent = new Intent(Intent.ACTION_SEND);

intent.putExtra("EXPECTED", "someString"); //An expected parameter

intent.putExtra("UNEXPECTED", someObject); //Unexpected object


....


Intent intent = getIntent();

String expected = intent.getStringExtra("EXPECTED");
```

9. USING COMPONENTS WITH KNOWN VULNERABILITIES

The best way to prevent attacks resulting from this vulnerability is to ensure rigorous processes and policies are applied that constantly monitor libraries and components to ensure that the latest versions are used and to regularly apply up to date patches.

EXAMPLE OF AN ATTACK.

In the field of the Internet of Things (IoT), it is difficult to maintain the use of current libraries and the application of patches due to the speed of product development and insecure policies with multiple devices interacting with each other. Attackers can create programs to detect unpatched or misconfigured applications. Even today, the Heartbleed bug (https://nvd.nist.gov/vuln/detail/CVE-2014-0160) which was discovered in 2014, is still active in devices that are not protected from it. The Heartbleed bug, in short, the original risk in OpenSSL, all boils down to this line of code:

```
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
```

In this instance, memory is allocated from the payload + padding with a value entered by the user. Because there is no length check, an attacker could feasibly force the OpenSSL server to read arbitrary memory locations.

10. INSECURE DESERIALISATION

Proper logging and error condition checking is crucial for defence against brute force attacks such as guessing passwords or detecting illicit network traffic. Logging and Monitoring are important as they give security teams precious time for forensic analysis and evolving their defence mechanism. Insufficient logging and monitoring mechanism can often lead to severe consequences.

Login failure without proper anti-brute force detection has caused hundreds of millions of user accounts to be hacked. During network infiltration tests, an important factor of the attack team is to find ways of entering the target environment without being monitored or logged. Today, developers that think like attackers are crucial in software development to try and identify defects and vulnerabilities whilst coding.

EXAMPLE OF AN ATTACK.

This example shows that if there is no action taken after user login failure, there is a high chance that someone else could mount a brute force attack on the system to guess the users' passwords. The right way of doing this would be to properly log the failed login attempts and once there are too many failed login attempts, the system should block this user from logging in during a 'freeze' period.

```
...
If (isPasswordMatch(user.getPassword(), recordPassword)) {
  return "login failed";
}
...
```

CONCLUSION

It is important to note that knowledge of OWASP needs to be included as part of a package of activities to enhance security. Organisations need to have an established security plan, leadership creates awareness from all the team, ensure continuous reviews and testing, use secure coding practice with proper security in the development environment, proper governance and oversight etc.

One of our philosophies at Xcalibyte is that real security comes from training people about the risks and then providing them with the methodologies and tools to stay secure. QA and testing that happens in the SDLC is critical but what we believe is that security needs to be considered at the beginning and development must kick off with security in mind. Creating awareness and making sure you learn this top ten list is a good start!

ABOUT XCALIBYTE

Xcalibyte's mission is to improve the quality of software by creating easy-to-use tools that help developers build and deploy reliable and secure code. Using advanced static code analysis techniques, Xcalibyte's solutions help developers identify defects, adhere to compliance standards and detect vulnerabilities at an early stage in the software development process.
Find out more at www.xcalibyte.com