BILINGUAL PUBLISHING CO.
Pioneer of Global Academics Since 1984

Journal of Computer Science Research
https://ojs.bilpublishing.com/index.php/jcsr

ARTICLE

Optimization of Secure Coding Practices in SDLC as Part of Cybersecurity Framework

Kire Jakimoski[1] Zorica Stefanovska[1*] Vekoslav Stefanovski[2]

1. Faculty of Informatics, AUE-FON University, Skopje, Republic of North Macedonia
2. Sourcico, Tel Aviv, Israel

ARTICLE INFO

Article history
Received: 3 November 2021
Accepted: 13 June 2022
Published Online: 21 June 2022

Keywords:
Cybersecurity
Security risks
Secure SDLC
SQL injection
Broken authentication
Broken access control
Mitigation practices

ABSTRACT

Cybersecurity is a global goal that is central to national security planning in many countries. One of the most active research fields is design of practices for the development of so-called highly secure software as a kind of protection and reduction of the risks from cyber threats. The use of a secure software product in a real environment enables the reduction of the vulnerability of the system as a whole. It would be logical to find the most optimal solution for the integration of secure coding in the classic SDLC (software development life cycle). This paper aims to suggest practices and tips that should be followed for secure coding, in order to avoid cost and time overruns because of untimely identification of security issues. It presents the implementation of secure coding practices in software development, and showcases several real-world scenarios from different phases of the SDLC, as well as mitigation strategies. The paper covers techniques for SQL injection mitigation, authentication management for staging environments, and access control verification using JSON Web Tokens.

1. Introduction

Software is the transformation of an idea that becomes a reality in the form of a software solution to a specific real-world problem [1]. The international standard ISO/IEC/IEEE12207-2008 [1] which defines the working framework for all activities that are part of a software life cycle indicates that the software starts with an idea, i.e. with a precisely defined need for a certain type of software product. The software product is a set of computer programs accompanied by appropriate documentation, which was designed and developed for commercial purposes, i.e. sales. Everyday life imposes a great need for new software products that should, above all, be quality, but also safe. If secure coding is not applied during the development of new software, the possibility of a weakness of the software solution remains, i.e. the solution itself becomes a vulnerability of the system in general. Practice shows that such vulnerabilities are often the result of insufficient testing of the security aspect of the code, insufficient education of

*Corresponding Author:
Zorica Stefanovska,
Faculty of Informatics, AUE-FON University, Skopje, Republic of North Macedonia;
Email: zstefanovska@yahoo.com

DOI: https://doi.org/10.30564/jcsr.v4i2.4048

new IT specialists on the term secure coding, differences in security rules in different programming languages, use of free software (open source) and the like. On the other hand, due to insufficient information of people about cyber threats and in the absence of basic cyber-awareness in the common man, we are witnessing the massive use of unverified and non-validated software created with insecure coding which is one of the many vulnerabilities of systems.

There are many concepts for developing high quality and functional software [2]. The challenge of any good team of developers is to create a secure and high quality software solution that will meet security practices and measures while not being a bottleneck for the of software's functionality. Finding the optimal solution that will meet both conditions: Security and Functionality, is considered one of the most challenging tasks in the life cycle of software solution development. Achieving "ideally secure" software requires new mechanisms in coding, raising the expertise of developers to write secure code, investing in their additional education in the field of IT security, implementing additional security specifics in writing code and the like [3].

The purpose of this paper is to point out good practices and tips to be used in software development to integrate secure coding at all stages of the development cycle. This paper can help software product engineers anticipate and recognize the challenges in cyberspace that would be a vulnerability to the product they create. At the same time, this paper will contribute to raising awareness of cyber attacks among young developers and the need to write secure code, which will be subject to various types of testing in the first phase of SDLC. In other words, this paper presents the optimization of secure coding in the development of software applications using practices to improve the quality of software solutions from a security perspective, while offering the user an optimal security solution, and thus approaching the ideal security functional software.

2. Related Works

In recent years, the number of different vulnerabilities of different software products has been increasing. Software vulnerabilities are constantly growing, but the search for new ways and practices to improve software products is also growing. The emergence of vulnerable software over a period of 20 years is illustrated in Figure 1.

There are several definitions of software product vulnerability, including that of the IETF (IETF RFC 4949): "A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy" [5].

To overcome software development vulnerabilities that contribute to the creation of vulnerable software, different methods of software development have been identified in practice. Recently, most popular are the agile methods for developing software products that have incorporated the security issue in each stage of their SDLC. Namely, more and more software companies in the development of new software use secure coding practices and test the security of the software at every stage of development, while respecting the principles of the standard SDLC [6].

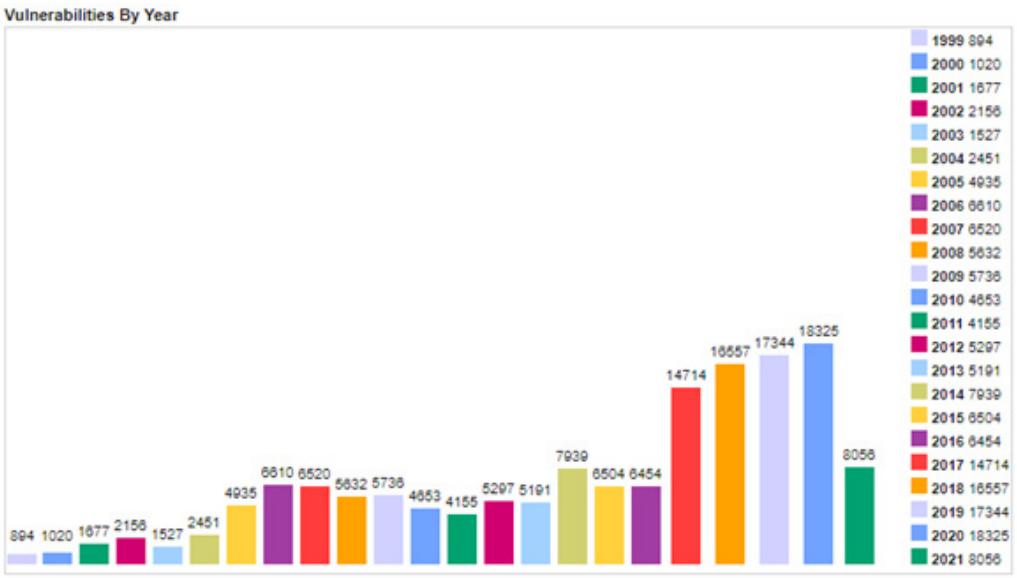

Figure 1. Number of reported vulnerable software in the CVE (Common Vulnerabilities and Exposures) database from 1999 to 2021. [4]

The term secure coding has attracted a great deal of attention in recent years. There are several ways to define this term. Most intuitively, we can define secure coding as a way of writing a secure program that will be as resistant as possible to illegal operations by malicious programs or people. By illegal operations we mean operations that compromise the security of the data and the application as a whole. If errors occur in the program that contributes to the program not fulfilling its functionalities, but these errors do not have a security implication, then we must not declare the program as unsafe.

It is necessary to distinguish between a functional application of sufficient quality that is not safe, and a safe but not sufficiently functional application. Secure coding helps protect user data from theft, corruption and malicious use. Security is not something that can be added to the software in the end as a finishing touch. In order to integrate security into the software itself, the natures of the threats must first be identified and accordingly security coding practices must be included during the software planning phase.

Without going into the reasons why malware attacks software applications, we must be aware that even the slightest vulnerability of a system is an open vector for attack and data theft. Attacks can be automated and replicated, but any vulnerability, no matter how small, is a real threat to the system as a whole, noting that no platform is immune to cyber attacks today.

It should be noted that secure coding is important for all types of software, from even the small everyday scripts that developers often write for themselves, to the largest commercial applications intended for public use.

Practices and tips for secure coding can be suggested by any experienced software team. Whether they will be implemented in software solution development usually depends on the management team leading the software development project.

Guided by the idea of avoiding the additional financial implications that would result from additional steps in the classic SDLC, the implementation of security in the early stages of development is often avoided. Practice, on the other hand, has shown that saving on security compromises application data protection. Namely, it has happened many times that the financial loss caused by a cyber attack on a web application is much greater than the finances that would be needed to include security in all phases of the SDLC. The SDLC security proposed by Microsoft is a model that includes 12 practices that need to be implemented and with their proper implementation, the security of the application is achieved in the most economical way.

These are the following practices for secure SDLC by Microsoft [7]:

- Provide Training.
- Define Security Requirements.
- Define Metrics and Compliance Reporting.
- Perform Threat Modeling.
- Establish Design Requirements.
- Define and Use Cryptography Standards.
- Manage the Security Risk of Using Third-Party Components.
- Use Approved Tools
- Perform Static Analysis Security Testing (SAST).
- Perform Dynamic Analysis Security Testing (DAST).
- Perform Penetration Testing.
- Establish a Standard Incident Response Process.

3. Top 10 Web Application Security Risks

Every development team would like to know in advance what the possible attack risks are for the application they are developing. There are several methods to anticipate possible security risks that, if not addressed in a timely manner, could result in a vulnerable and unsafe application. The security risk analysis, according to the OWASP① Methodology, is treated with four metrics to determine the level of risk for the software solution - Usability, Frequency, Lightness and Technical Impact [8]. Risk analysis provides recommendations and tips that can successfully detect if an application is vulnerable, as well as tips and suggested practices on how to protect ourselves from these risks. The tips and recommendations that we will point out are of great importance for the development teams that are trying to develop secure code. If they are implemented and followed at all stages of SDLC when developing a software product, it is very likely that you will get Secure SDLC. In that way, the end goal of highly secure and functional software can be achieved in the most economical way. In essence, by creating your own model for secure coding according to the advice and methodology of OWASP, each organization can achieve optimization of secure coding in the development of software solutions in both the private and public sector. The followings are the top 10 web application security risks:

Injection. Injection occurs when untrusted data (usually provided by the user) is sent to command or query interpreter. Commonly used vectors are databases (SQL Injection) and operating system shells (OS Command Injection). The malicious agent can use a specially crafted input that will be sent to the underlying interpreted, executing unintended commands or accessing unauthorized data [9,10].

① The Open Web Application Security Project® (OWASP) is a non-profit foundation that works to improve the security of software.

Broken Authentication. Authentication and overall session management is easy to implement in a functional and insecure manner. This means that while the application is functioning correctly for regular users, it is possible for malicious agents to compromise passwords, keys, or session tokens [11].

Sensitive Data Exposure. Many web applications use sensitive and personally identifiable information, so those data must be stored on the server, transferred to the browser, and used during the browser session. Each of those sites is a possible exposure risk, a place where attacker can steal or modify data. This can result in credit card fraud, identity theft and other crimes.

XML External Entities (XXE). XML processors provide the option of evaluating external entity references. While this is a useful feature, it can be an attack vector if used with untrusted data. Possible issues include disclosure of internal server files and file shares, scanning and access of internal ports, remote code execution, and denial of service attacks via XML bombs.

Broken Access Control. Another issue that is commonly implemented in a functional and insecure way is authorization and access control. Abuse of these features would enable a malicious user to escalate its privileges – which could lead to access of other user's data, and in extreme cases, changing of access rights and overtaking of the system [12].

Security Misconfiguration. This is not a single issue, but a result of a flawed application deployment process. The most common issues are insecure default configurations, incomplete or ad hoc configurations, open cloud storage, verbose error messages, etc. Misconfiguration can happen at any level of an application stack, as well as on all the interfaces between different levels of the stack, both technical and human. It is common to have a vulnerability because of miscommunication of responsibilities.

Cross-Site Scripting XSS. XSS is an injection-based attack that focuses on the front-end of a web application. It happens when user-provided data is not properly validated and sanitized. Commonly it is used in a stored way, with the attacker injecting data in the web-site's database, which is later viewed by a victim. This can lead to session hijacking, data leaks or redirects to malicious sites.

Insecure Deserialization. A common approach to passing information between the client and the server is to exchange a state object, e.g. in a cookie. This state object is serialized and encoded using some scheme when in transit and is then deserialized on the client and on the server. If an attacker is able to deserialize the serialized state, he can access the application data inside, or even tamper with it. This could lead to remote code execution, privilege escalation, session hijacking, and other breaches.

Using Components with Known Vulnerabilities. Any non-trivial application will use third-party libraries, frameworks, packages, and other software modules as part of its code base.

All the parts run with the same privileges the application itself is running with, which means that any vulnerability of a component is a vulnerability of the application. These kinds of attacks are also lucrative for the attackers, as finding a vulnerability inside a heavily used component can allow access to multiple sites.

Insufficient Logging & Monitoring. If despite all our efforts, a breach does occur, it is extremely important that we have the necessary tools to detect it and mitigate it. With some attacks, like DDoS, proper detection is crucial in the defense of our site. Also, a common scenario is that once an attacker successfully overtakes a system, that system can be used as a foothold in attacking connected systems.

4. Example and Tips for Secure Coding

4.1 Injection into SQL Expressions

Technique overview. Most RDBMS② are using SQL as the querying and command language and the application build over them communicate with the database by constructing and sending SQL commands. The database does not know if the queries are malicious or not, and if they are valid, they will be executed. SQL Injection is an attack technique that will trick the application server into constructing a malicious command and getting the database to simply execute it. One often used type of attack is on applications where the construction of an SQL command is done with string concatenation. If the application concats unverified and unsanitized user input, the user can basically short-circuit the SQL Expression, and attach another of his own [13].

Example. In a PHP-based application, the following code is used to select values from a table called Items.

```
$sql = "SELECT Name, Status
        FROM Items
        WHERE Status != " . ITEM_DELETED_
        STATUS . "AND ID = " . $item_id . ";
```

This command is constructed with concatenating the fixed text of the command with two code-level parameters, ITEM_DELETED_STATUS and $item_id. The

② Relational Database Management System

ITEM_DELETED_STATUS is a constant that is defined in the code, so this cannot be used as an attack vector. On the other hand, $item_id is taken from a parameter of the request, using the following code:

```
$item_id = filter_var($_GET[“id”], FILTER_SANITIZE_
           STRING)
```

While it seems that the input is sanitized, the sanitization used is targeted to prevent XSS attacks. It does nothing to prevent SQL injection attacks - so from a database perspective, the value of the user input is completely raw. This endpoint could be accessed using something like the following URL:

```
http://server/item-info.php?id=123
```

In that case, the value of the $item_id variable will be “123”. The actual value of the $sql variable will be:

```
SELECT Name, Status
FROM Items
WHERE Status != 0 AND ID = 123
```

This is a valid SQL expression that when executed by the database will return the Name and Status of the item with and ID of 123. One variant of SQL injection changes the value of this parameter to an expression that will return more data than the original expression. E.g., if we use the following URL:

```
http://server/item-info.php?id=123%20OR%201=1
```

the value of the $sql variable will become:

```
SELECT Name, Status
FROM Items
WHERE Status != 0 AND ID = 123 OR 1=1
```

Since 1=1 is a condition that is always true, this command will effectively return the names and statuses of all items in the database. Another variant is to use the specifics of SQL to attach an additional statement after the intended statement. E.g. the following link includes a destructive DDL statement:

```
http://server/item-info.php?id=123;%20DROP%20
TABLE%20Items
```

The value of the $sql variable will become:

```
SELECT Name, Status
FROM Items
WHERE Status != 0 AND ID = 123; DROP
TABLE Items
```

This actually changes our SQL statement into two statements. One is the original query, while the other is a destructive command, and will delete the Items table itself. Once this request is processed, the application will no longer have such a table, which means that, at best, the application is nonfunctional, and at worst, a major, and potentially unrecoverable data loss.

In this specific application, all database queries are run under a user that has full privileges not only on the database, but on the database server as well, so even more drastic privilege escalations are possible.

Mitigation of the example code’s vulnerability. There are multiple approaches available to this piece of code. One of the most basic ones is to limit the destructive power of an intruder, even if a successful attack occurs.

Database user privileges. The user that accesses the database should have the minimum permission that are sufficient to execute their intended operations. Usually, the user needs only to have data manipulation permissions (selecting, inserting and modifying data). This would mean that while the attack via the http://server/item-info.php?id=123%20OR%201=1 URL will succeed and leak data, the attack via the http://server/item-info.php?id=123;%20DROP%20TABLE%20Items URL will fail. An outside attacker will not be able to destroy our database, but they will still be able to extract data they should not be able to. In this specific case, the SQL command

```
REVOKE ALL ON `Database`.* FROM
‘user’@’localhost’;
GRANT SELECT, INSERT, UPDATE ON
`Database`.*
TO ‘user’@’localhost’;
```

was used to modify the accessing users’ privileges. First, all privileges were revoked, and then the user was explicitly granted only the SELECT, INSERT and UPDATE privileges. Since the application uses a technique known as soft delete, the DELETE permission was not required, so it was not granted. This approach should be used in all scenarios, as the application user should not have any extra permissions that those that are actually needed.

Type validation on user input. Another way to defend against SQL injection attacks is to ensure that the user input does conform to the type requirements of the query. In this case, the input should be an integer, so if we test the input for that, we can detect this attack and stop it before it gets to the database.

```
$item_id = filter_var($_GET[“id”], FILTER_SANITIZE_
                        STRING)

    if (!is_numeric($item_id)) {
       logError(“Invalid item_id received from client “
       . $item_id);
        die;
    }

    $sql = “SELECT Name, Status
            FROM Items
            WHERE Status != “ . ITEM_DELETED_
    STATUS
            . “ AND ID = “ . $item_id . “;”
```

This code uses the library function is_numeric to check whether the $item_id variable is either a valid number, or a string containing a valid number. If it's not, then an error is logged, and the processing of the request stops immediately. The malicious SQL is neither generated nor sent to the database.

This approach is effective, but it's not systemic. It's hard to check all the options for every single query, and the burden of implementation is on the developer.

Query parametrization using PDO. A better approach is to avoid manual generation of the SQL string completely. We can use a technique called prepared statements that is supported by most databases. In this case, we send the query using parameters, i.e. the text of the query is defined once, with placeholders at the variable parts. The values that need to specify the parameters are send separately. Since the database engine knows that is should execute a specific query format, it knows the types of the parameters, so it will not allow for any insertion of SQL statements.

In PHP there is a PDO library that supports using prepared statements. The code in our case would look like this

```
$item_id = filter_var($_GET[“id”], FILTER_SANITIZE_
            STRING)
   $statement = $pdo->prepare(“SELECT Name, Status
   FROM Items WHERE  WHERE Status != :status
   AND ID = :item_id”);
   $statement->bindValue(“:status”,ITEM_DELETED_
STATUS,
   PDO::PARAM_INT);
   $statement->bindValue(“:item_id”, $item_id,
   PDO::PARAM_INT);
   $statement->execute();
```

Since the statement knows that the :item_id parameter should only have an integer value, it will not allow any insertion of SQL inside the value.

Implicit parameterization using ORM. Instead of hand-crafting our SQL, it's quite possible to use a tool to map it for us. These kinds of tools are called Object-Relational Mappers (ORM). The most popular ORM for PHP is called Eloquent and it is part of the Laravel framework. Using it, we can describe the shape of our database using a model. Then, instead of creating SQL statements, we use regular language concepts to specify the data we need, and the SQL query is generated by the ORM automatically. This has the benefit that we are protected from SQL injection attacks by design, as there is no SQL to concatenate in an unintended way [14-16].

The code would look like this:

```
use Illuminate\Database\Eloquent\Model;
class Items extends Model
{
   protected $primaryKey = ‘ID’;
   protected $fillable = [
      ‘Name’,’Status’
   ];
}
const item = Items::where([
   [‘Status’, ‘!=’, ITEM_DELETED_STATUS],
   [‘ID’, ‘=’, $item_id],
])->first()
```

A major drawback to this approach is that, as part of a framework, it can't be easily used in isolation, as it requires significant setup effort.

Discussion. After evaluation of the different approaches we decided to implement explicit parametrization of the code. While with Eloquent any parametrization is implicit and easier to use, it requires a major refactor of the application. It was decided not to proceed with such a change. Instead, the database privileges were fixed, and in addition, all the vulnerable SQL statements were transformed into a parametrized format. In specific places, where it made sense from a user perspective, type checks were added as well, in order to be able to return user-friendly errors.

4.2 Broken Authentication

Technique overview. One of the most trivial, yet persistent security holes are authentication leaks. The issue is that quite often, they are not a purely technical problem, i.e. it's not enough to solve them through code, but they require user discipline and education.

Quite often, especially with systems with automated deployment, it is common to have a set of hardcoded system users with total access to the system. It is assumed that once the system is deployed, the operator should change the default credentials to custom ones, so that those users cannot be used by unauthorized persons. However, this is not always the case, so there are plenty of cases where an otherwise secure system was compromised using the default set of credentials.

Example. A large, distributed team of developers are developing a large, distributed system. The authentication of the system is done with a regular username/password combination, with optional two-factor authentication. Because of business reasons, it's not possible to enforce two-factor authentication across the board. The process of registration of a user involved email verification.

Since there are many changes being done to the system at a given time, using a single testing and staging environment is not practical. A procedure was developed for automated generation of ephemeral testing/staging environments. These environments are a point-in-time replica of the production environment, including databases, storages, services, cloud resources, etc.

Any subteam is able to generate such an environment, use it to test and stage a feature, and once ready, push it to production. The authorization pool that is generated per environment used a single hardcoded user with a global super-admin role. The user was preset as verified. The password for the user was stored inside a secret of the continuous integration tool, so it was not directly accessible to the developers. The intention was that only an enumerable list of people will have access to it, and that after generation of an environment, there should be a manual intervention to change the password.

That was not always the case, and, in time, most of the developers knew and used the default password.

Mitigation of the example code's vulnerability. To address the vulnerability before it became an attack, several solution scenarios were proposed.

Enforce scrubbing of data. Since the default account was used only on the ephemeral environments, the leakage will be much smaller if the data are scrubbed of any personally identifiable information. This approach would trivialize the problem, however it has some issues of its own – mainly that it's hard to guarantee and enforce a proper scrubbing procedure on a system that changes often.

These issues were considered, and it was decided that this approach, for the specific system, will create more problems than it solves, so it was not implemented.

Limit the access of the environments. Another proposed option was to limit the physical access of the environments to users within the company, instead of the general internet. This solution had the benefit that it is easy to enforce via network policies, even for remote workers, using VPN filtering or similar approaches. Also, this kind of solution was already used for things like cloud service or direct database access.

However, the business requirements are that the ephemeral environments had to be accessible to specific stakeholders who are outside of the company. While it is possible to expose the environments in a controlled manner, it would have created additional workload for the operations team, as well as disrupting the user experience of external stakeholders.

Since this approach was determined to create additional workloads, without solving the underlying problem, it was not implemented. It was decided that we might implement limited access to some ephemeral environments if we know they won't be used from outside the organization.

Code-based limitation of the hard-coded account. Since the hardcoded user should ideally be used only to create the real users that will actually use the environment, a possible solution would be to add such restrictions to the default user. For example, we could add a rule that the default user is only active some preset time after creation, or that it can only do specific actions, or we can disable it once a real super-admin user is generated, etc.

While these actions will effectively solve the problem, they would require changes and specific checks in the authentication/authorization code. This means that the hardcoded account will not only be hardcoded in the configuration of the ephemeral environments, but also in the service that processes the users.

The drawbacks are that the code will have to behave differently for different users, and that would make the system inconsistent. It will dramatically increase the need to test and verify that the authentication/authorization process is operational and secure.

This approach was dismissed because, while effective, it will increase the complexity of an already complicated system.

Implement two-factor authentication. A fourth approach was to turn on the two-factor authentication for the hardcoded account. Since this would be used by multiple

people, we will need to use an application for sharing TOTP verification codes.

This lowers the security value of the hardcoded password, since even if a malicious user knows the password, they cannot use it to login to the system, unless they have access to the shared TOTP application. And since most tools for secret sharing include centralized administration, this transfers the problem to management of the secret sharing system. This will dramatically reduce the number of people who can tamper with the system.

The only downside of this solution is that it requires a centralized secret sharing tool, but there are plenty reasonably priced solutions for that.

Add account verification. Another option was to avoid hardcoding the password for the super-admin account at all. Instead of generating a pre-verified account with a set username and password, only the username can be hardcoded, and then use an email to verify the account and set a password. Since the environment deployment process already generated an email specific to the environment, this was easy to implement, and the implementation would only change the deployment process.

The downside is that the environment is not immediately useful, as it will require a manual step of verifying the account. However, since the environment generation process is usually monitored, the person responsible for the specific environment can easily verify and set a password. If needed, they can set up two factor authentication for the specific account, or even disable the account altogether. And since the password will be generated by them, it will be unknown even to the system administrators.

Discussion. After evaluation of the different options available, it was deemed that the last two options will systematically solve the problem. Taking in mind the specific organization of the development teams, it was decided to use the last approach, as it transferred responsibility for password management on a specific environment to the team itself. A part of the solution was a training session for the team leaders on how to set and secure the password of the super-admin user.

4.3 Broken Access Control

Technique overview. Once an application knows who the user is, it's imperative to know what the operations are the user can do, and, just as important, what operations should be prohibited. Authorization is extremely complicated problem to solve, and quite often is tricky to validate. This kind of attack misuses that complexity to make the attacked system think that the malicious user has more capabilities that they should actually have.

One vector of attack, on applications that use JSON Web Tokens for authorization and access control, is to tamper with the data present in the token. A JSON Web Token consists of three parts: header, payload and signature. The signature can cryptographically verify the contents on the header and payload, so that it can be detected whether the data of the payload was tampered with. However, unless it is being done automatically, there is potential for error, and an attacker can target the endpoint of the system that does not implement correct verification.

Example. A JavaScript based server-side application is using JWT tokens for authorization and access control. It uses an external service for token generation, and the verification used the same external service. That means that the process of verifying the token's integrity was slow, and developers tended not to use it, as it was making the application sluggish.

An example of a JWT would be:

eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ik-
pvaG4gRG9lIiwiZW1haWwiOiJlbWFpbEBleG-
FtcGxlLmNvbSIsImlhdCI6MTUxNjIzOTAyMn0.
kEmXw9lLw3tO1HloDZQoORejF1RiwVFSv-
73VGkbCy7Cu91ZSyuW1b7LayrNWcknl5wP3JH-
9kH1err0Mx96kbrA1uHpu0RXoRmLraTY-
f40krmSVLO1czYZB69BtQEkWIG3wup_
wlbhDZLiKkJgyLSPx6gnhTQibSw9U7rW07Wm-
CPu36-KyfgXedX--Mk-MsJqyiSBVHlhbMJmjlD-
ABWJJ1fQRF2lsirug9D-16MEYFkzOshvPI1nczLH-
8CBk-ls-VL5c67JPUpmOqYczEGvOth50Bymloc2Jf_
l8pJUWjZzejF-Hsg4AGRHkDrYNbQELHbfGYrNKhyr_
vF0j4BpquYw

It consists of three parts that are separated by the dot(.) characters. The first two sections are simply base-64 encoded strings. If we decode them, we can get their plain text quite easily. This specific token has the header of

```
{
 "alg": "RS256",
 "typ": "JWT"
}
```

and a payload of

```
{
 "sub": "1234567890",
 "name": "John Doe",
 "email": "email@example.com",
 "iat": 1516239022
}
```

We can note that both header and payload are simple json objects, and, in the payload, the data of the user is

plainly visible. The frontend application stores this token in a cookie called ident and sends it on every request to the backend service.

The service in question is using the express framework to fulfill the requests, and an example endpoint is https://www.example.com/api/users/me which returns the full user profile for the currently logged in user, including personally sensitive information.

In order to avoid using user-specific parameters in the endpoint url, it uses the provided cookie to extract the user email, based on which the full profile is loaded from the database and returned to the client.

The code that handles the request is

```
router.get( '/users/me', (req, res) => {
    const email = getEmailFromCookie(req);
    if (!email) {
        res.sendStatus(http.forbidden);
        return;
    }

    const profile = await UserService.getProfile(email);
    return res.status(http.ok).send({ profile });
})
```

This code will extract the email from the request, and once found will query the database for the profile. This means that if a malicious user is able to successfully change the return value of the getEmailFromCookie function, he will successfully retrieve the full profile of another user.

The function getEmailFromCookie is:

```
const getEmailFromCookie = (request: HttpRequest)
=> {
    const ident: string = request.cookies.ident;
    if (!ident) {
        return undefined;
    }
    const parts = ident.split('.');
    const userData = decodeBase64(parts[1]);
    const { email } = userData;
    return email;
}
```

This code simply takes the payload from the JWT and, without running any verifications, decodes and returns the email.

This means that the user can, manually or automatically, change the value of the cookie to another with the same header and signature, but whose payload decodes to

```
{
 "sub": "1234567890",
 "name": "John Doe",
 "email": "victim@example.com",
 "iat": 1516239022
}
```

Any verification of this token would mark it as invalid, but since there is no verification, this will not be noticed. Once called, the API endpoint will treat this as a valid request from the user victim@example.com, and return the full user profile to the attacker.

Mitigation of the example code's vulnerability. The obvious solution to this issue, once identified, is to add token validation. However, this needs to fulfill some requirements:

- It should be done on every request, i.e. the developers should not be allowed to opt-out of the validation
- It should be done implicitly, with no effort on the developer side, so that it will be impossible to forget to use it
- It should be performant, as the added verification should not slow down the application more than absolutely necessary

The existing verification code failed on all three of these requirements since it was explicit and used an external service.

In order to solve the first two requirements, the verification was implemented as an express middleware that verified that the token in the ident cookie was a valid JWT token. Because of the specifics of the express framework, the middleware will be called before the actual route handler. The route handler will be invoked if and only if, the middleware ends its run with a call to the next function. If the verification is not successful, i.e. if the token has been tampered with, we stop the processing, returning a forbidden error

Note that the actual verification is done inside the verify JWT token function. That function takes the token as a parameter, and verifies it asynchronously. Once the promise is resolved, we have a single boolean with the verification result.

To satisfy the third requirement, the verify JWToken function used a two-pronged approach. It maintained a list of tokens that were already verified (along with their expiration dates), so that a known good token does not have to be verified all the time. This is needed because the verification of the signature itself takes a non-trivial processing time, even when running locally. The nature of the service is that a user will usually request several hundreds

of API endpoints in a small amount of time, so keeping a list of known good tokens can effectively short-circuit that verification, at a small memory cost.

The second prong was to run the verification process locally instead of using the external service. The service helpfully provided for a way to download the key that is being currently used for the client (along with its expiration details) in a JSON Web Key format (JWK). In order to run the validation locally, several external libraries were needed, like jsonwebtoken and jwk-to-pem. The code for the verification was:

```
const verifyJWToken = async (token:string) => {
  if (checkCache(token)) {
    return true;
  };
  const pem = jwkToPem(jsonWebKey);
  const result = await new Promise((resolve) => {
    jwt.verify(token, pem, { algorithms: ['RS256'] },
      (err, payload) => {
                if (err) {
          return resolve({success: false});
        }
        return resolve({success: true, payload});
      });
  });
  if (!result.success) {
    return false;
  }
  if (isExpired(result.payload.exp)) {
    return false;
  }
  tokenCache[token] = result.payload.exp;
  return true;
};
```

This code first checks the cache for the token. If the token is found, it returns success. If the token is not in the cache, we can verify it using the jwt library. If the verification is successful, we get the decoded payload as a result. Once we have that, we're checking for token expiration one more time, and if everything is ok, we are signaling that the verification is successful.

This approach fulfills all three requirements, as it is both performant and transparent for the end user.

5. Conclusions

Secure coding and adherence to secure SDLC is quite a difficult task, both for the developers and the other members of the project team. The recommendations and tips outlined in this paper are intended to help software companies in the public and private sectors reduce the risk of application attacks in the most cost-effective way. This goal can be achieved exclusively by using the multitude of resources offered in the literature and empirically proven to have a positive impact on cyber defense, resulting in a functional and secure software product.

To create a secure application, you first need to define the term application security. In other words, frame all the answers to the question: What is security for a software product? Such a framework should guide the development of a secure application. Most of the answers to this question come from several factors such as user requirements, the environment in which the application will be developed, the production environment, and the social environment in which the application will be implemented.

Conflict of Interest

Authors declare no conflict of interests.

References

[1] ISO/IEC/IEEE International Standard, 2008. Systems and software engineering -- Software life cycle processes. IEEE STD 12207-2008. pp. 1-138. DOI: https://doi.org/10.1109/IEEESTD.2008.4475826

[2] Vale, T., Crnkovic, I., De Almeida, E.S., et al., 2016. Twenty-eight years of component-based software engineering. Journal of Systems and Software. 111, 128-148.

[3] Gorski, P.L., Acar, Y., Lo Iacono, L., et al., 2020. Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs. In-Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. pp. 1-13.

[4] CVE Details, Vulnerabilities By Year . Available from: https://www.cvedetails.com/browse-by-date.php

[5] Shirey, R., 2007. Internet Security Glossary, Version 2. DOI: https://doi.org/10.17487/RFC4949

[6] Baldassarre, M.T., Santa Barletta, V., Caivano, D., et al., 2020. Integrating security and privacy in software development. Software Quality Journal. 28(3), 987-1018.

[7] Microsoft SDL Practices. Available from: https://www.microsoft.com/en-us/securityengineering/sdl/practices

[8] OWASP Risk Rating Methodology. Available from: https://owasp.org/www-community/

[9] Alwan, Z.S., Younis, M.F., 2017. Detection and prevention of SQL injection attack: A survey. International Journal of Computer Science and Mobile

Computing. 6(8), 5-17.

[10] Sinha, S., 2019. Finding Command Injection Vulnerabilities. Bug Bounty Hunting for Web Security 2019. Apress, Berkeley, CA. pp. 147-165.

[11] Nadar, V.M., Chatterjee, M., Jacob, L., 2018. A Defensive Approach for CSRF and Broken Authentication and Session Management Attack. InAmbient Communications and Computer Systems. Springer, Singapore. pp. 577-588.

[12] Petracca, G., Capobianco, F., Skalka, C., et al., 2017. On risk in access control enforcement. InProceedings of the 22nd ACM on Symposium on Access Control Models and Technologies. pp. 31-42.

[13] Tasevski, I., Jakimoski, K., 2020. Overview of SQL Injection Defense Mechanisms. In2020 28th Telecommunications Forum (TELFOR). IEEE. pp. 1-4.

[14] Budiman, E., Jamil, M., Hairah, U., et al., 2017. Eloquent object relational mapping models for biodiversity information system. In 2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT). IEEE. pp. 1-5.

[15] Sinha, S., 2019. Database Migration and Eloquent. Beginning Laravel. pp. 113-166.

[16] Apress, B., Stauffer, C.A., Laravel, M., 2019. Up & running: A framework for building modern php apps. O'Reilly Media.