Setup and proof-of-concept development for select high-risk application vulnerabilities

Project-I Report submitted to
Indian Institute of Technology Kharagpur
for the Award of the Degree

Bachelor of Technology (Hons.)
In
Electronics and Electrical Communication Engineering

By
Shivam Saxena
(17EC10054)

Under the supervision of
Prof. Manoj Kumar Mondal

Rajendra Mishra School of Engineering Entrepreneurship
Indian Institute of Technology Kharagpur
Autumn Semester, 2020-2021

DECLARATION

I certify that

(a) The work contained in this report has been done by me under the guidance of my supervisor.

(b) The work has not been submitted to any other Institute for any degree or diploma.

(c) I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

(d) Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Date: November 24, 2020	Shivam Saxena
Place: Kharagpur	(17EC10054)

Contents

1 Introduction

Application security is the process of making apps more secure by finding, fixing, and enhancing the security of apps. Much of this happens during the development phase, but it includes tools and methods to protect apps once they are deployed. With more businesses and services moving their operations to app-based environments,it is becoming very important to develop ways to thwart malicious attacks by hackers.

Due to these reasons,application security is getting a lot of attention nowadays.The evolving nature of how business apps have been developed in the last few years has helped to create rapid growth in the application security sector. Gartner, in its report on the app security hype cycle has cautioned that IT managers “need to go beyond identifying common application development security errors and protecting against common attack techniques.” They illustrate an interesting approach of dividing products into different categories and describing where in their “hype cycle” they are located.

Nowadays most apps run on some type of APIs. API stands for application programming interface which is a concept that essentially refers to how multiple applications can interact with and obtain data from one another. APIs operate on an agreement of inputs and outputs. They help to offer a seamless integration of services from different parties. Naturally, due to their ubiquity, they are a big target for hackers to gain unauthorized access.

Another common feature of most applications today are integration with some form of back-end database. The most common mechanism to communicate with a database is SQL(Structured Query Language).SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access.

We look at some specific kinds of API vulnerabilities and SQL injections that are most prevalent. We try to develop different checks along the information pathway to make common system functionalities more robust.

2 Literature Survey

2.1 OWASP Top 10

The Open Web Application Security Project (OWASP) is an online community that produces freely-available articles, methodologies, documentation, tools, and technologies in the field of web application security. It regularly publishes articles and guides regarding new methodologies on how to best secure systems.It maintains a list called the OWASP Top 10,which is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications.

These security risks,in order are:

1. **Injection**. Injection flaws, for example, SQL, NoSQL, OS, and LDAP injection, happen when untrusted information is shipped off an interpreter as a component of an order or inquiry. The hacker's malicious code can fool the interpreter into executing unintended orders or getting to information without legitimate approval.
2. **Broken Authentication**. Application operations identified with session management and authentication are regularly formalized inaccurately, permitting assailants to tamper with passwords, keys, or meeting tokens, or to misuse other code inaccuracies to act as impostors incidentally or temporarily.

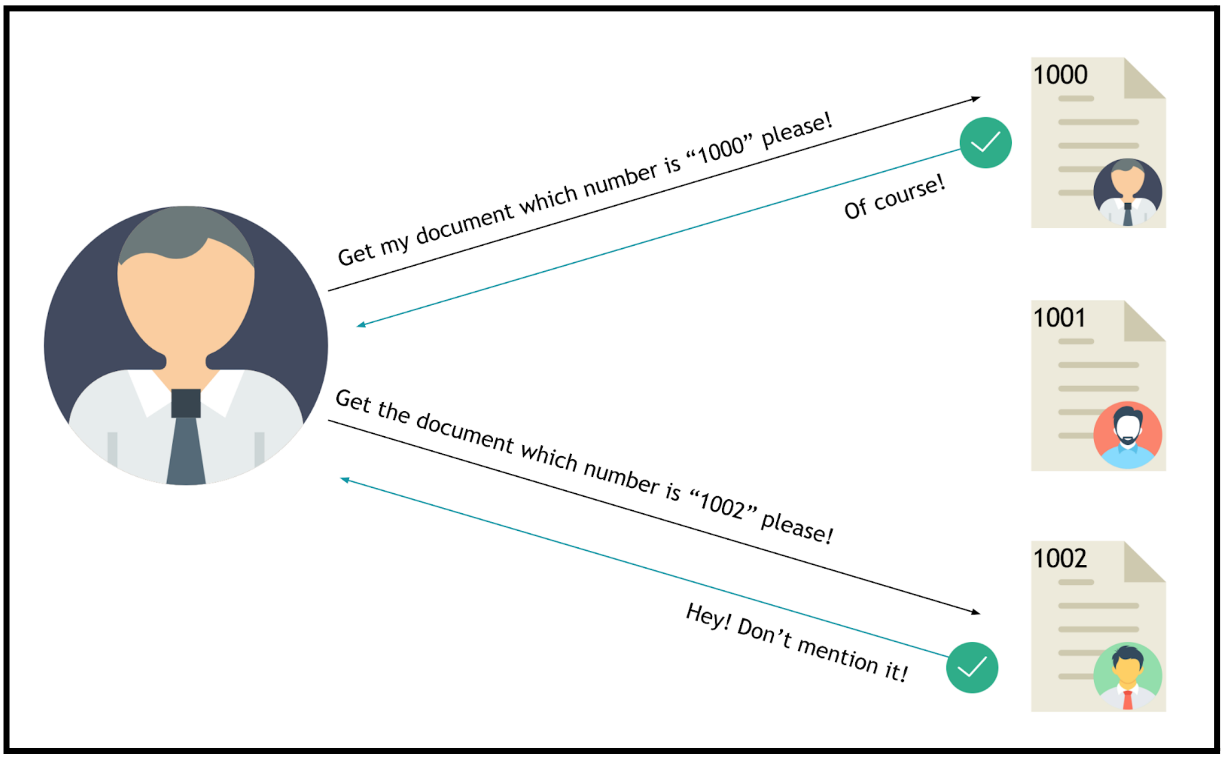
3. **Sensitive Data Exposure**. Many web applications and APIs don't appropriately secure delicate information, for example, monetary, medical care, and personally identifiable information. Attackers may steal or alter such poorly secured information to direct fraud in online transactions,identity theft etc. Important information might be undermined without additional assurance, for example, encryption at rest or in transit,, and requires extraordinary safety measures when traded with the program.
4. **XML External Entities (XXE)**. Numerous outdated or ineffectively compiled XML processors assess external element references within XML documents. External elements can be utilized to uncover sensitive records utilizing the document URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
5. **Broken Access Control**. Limitations on what validated clients are permitted to do are regularly not appropriately authorized. Attackers can misuse these defects to get to unapproved capabilities and/or information, for example, access other clients' records, see delicate documents, alter other clients' information, change access privileges, and so on.
6. **Security Misconfiguration**. Security misconfiguration is the most often observed issue. This is regularly an aftereffect of uncertain default designs, deficient or impromptu setups, open distributed storage, misconfigured HTTP headers, and expansive error messages containing touchy data. Not only should every single OS, framework, library, and application be safely designed, they should also be fixed/redesigned in a punctual manner.
7. **Cross-Site Scripting XSS**. XSS flaws happen at whatever point an application incorporates untrusted information in another page without legitimate approval or escaping, or refreshes a current page with client provided information utilizing a browser API that can generate HTML or JavaScript. XSS permits hackers to execute code in the victim's browser which can commandeer client sessions, alter web sites, or divert the client to malevolent sites.
8. **Insecure Deserialization**. Insecure deserialization regularly prompts remote code execution. Regardless of whether

deserialization imperfections don't bring about remote code execution, they can be utilized to perform other assaults, including replay attacks, infusion attacks, and privilege escalation attacks.

9. **Using Components with Known Vulnerabilities**. Components, for example, libraries, systems, and other programming modules, run with similar access rights as the application. On the off chance that a weak segment is abused, such an assault can encourage serious information tampering or server takeover. Applications and APIs utilizing components with familiar weaknesses may sabotage application protections and empower various different attacks.
10. **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, combined with absent or insufficient integration with incident response, permits hackers to escalate attacks on frameworks,look after persistence, turn to more frameworks, and alter, extricate, or pulverize information. Most breach studies show time to recognize a break is more than 200 days, commonly identified by external parties as opposed to internal processes or monitoring.

2.2 Injection and BOLA

We focus our attention to the most pressing issues: injection and broken authentication. Specifically, we take a look at SQL injection and Broken Object Level Authorization(BOLA), discussed more thoroughly in the OWASP API Security Top 10. Essentially, in BOLA, a user accesses objects that he should not have access to by manipulating the IDs. Modern applications handle many resources and expose many endpoints to access them. Hence, modern APIs expose many object IDs. Those IDs are an integral part of the REST standard and modern applications.


Get my document which number is “1000” please!
Of course!
Get the document which number is “1002” please!
Hey! Don’t mention it!
1000
1001
1002

2.3 Case Studies

In recent times, a lot of high profile security breaches,including at Uber, Verizon,Facebook fell under this category.

- In Uber’s case, attackers were able to exploit an account takeover vulnerability on Uber which allowed them to take over any other user’s Uber account (including riders, partners, eats) by supplying the unique user UUID(universally unique identifier) in the API request and using the leaked token in the API response to hijack accounts. They were also able to enumerate any other Uber’s user UUID by simply by appending their phone number or email address in another API request. Furthering that, it allowed an attacker to track the victim’s location, take rides from their account, etc. by compromising the account using the leaked access token of Uber mobile application. This also permitted takeover of Uber driver accounts and Uber Eats accounts.
- In Verizon’s case, an attacker found an improperly secured internal subdomain,which led to exposure of over 2 million Verizon Pay

Monthly contracts, which had all sorts of personal information and usage statistics.

- At Facebook, attackers exploited vulnerabilities in Facebook Portal, a video communication service from Facebook. It allowed hackers to:
    - disclose any attachment for all users that was sent through the chat, including all images, files, videos and audio messages
    - exploit this across all Facebook chat infrastructure(Facebook

      main chat, messenger, portal chat and workplace chat)

Many more such cases occur frequently. Fortunately, in these cases, the bugs got reported before any hackers were able to compromise these systems. Given the sheer volume of transactions handled by big tech companies, these vulnerabilities could potentially lead to billions of dollars' worth of damage. So it is absolutely essential to secure apps against BOLA vulnerabilities and SQL injections.

We make use of the Spring Framework for Java to realize and mitigate these API vulnerabilities. Spring is the most popular application development framework for enterprise Java. Spring is lightweight when it comes to size and transparency. So it can be used to create high performing, easily testable, and reusable code.

3 Scope and Objectives

1. Literature survey and understanding of the relevant software concepts.
2. Securing against BOLA API vulnerability using features available in Spring Framework to create an “authorization engine”.
3. Securing against SQL injection by solving common code inefficiencies.

4 Work Progress

4.1 BOLA

Solution with Spring: Method Security

Spring Security supports authorization semantics at the method level. Typically, we could secure our service layer by, for example, restricting which roles are able to execute a particular method – and test it using dedicated method-level security test support.

- Using *@Secured* Annotation

The *@Secured* annotation is used to specify a list of roles on a method. Hence, a user only can access that method if she has at least one of the specified roles.

```
@Secured({ "ROLE_VIEWER", "ROLE_EDITOR" })
public boolean isValidUsername(String username) {
    return userRoleRepository.isValidUsername(username);
}
```

In this case, the configuration states that if a user has either *ROLE_VIEWER* or *ROLE_EDITOR*, that user can invoke the *isValidUsername* method.

- Using *@PreAuthorize* and *@PostAuthorize* Annotations

Both *@PreAuthorize* and *@PostAuthorize* annotations provide expression-based access control. Hence, predicates can be written using SpEL (Spring Expression Language).

The *@PreAuthorize* annotation checks the given expression before entering the method, whereas, the *@PostAuthorize* annotation verifies it after the execution of the method and could alter the result.

Some examples are:

```
@PreAuthorize("hasRole('ROLE_VIEWER') or hasRole('ROLE_EDITOR')")
public boolean isValidUsername3(String username) {
    //...
}
```

```
@PreAuthorize("#username == authentication.principal.username")

public String getMyRoles(String username) {

    //...

}
```

Here, a user can invoke the getMyRoles method only if the value of the argument username is the same as the current principal's username.

A simple Spring App was set up to test these features. User information was set up in an embedded h2 database. These annotations were tweaked to our purposes. This was done to simulate cases often encountered in banking and e-commerce data, where users have access privileges to only certain fields like name,address,phone number etc.

while fields like credit scores, loan amounts, are accessible to only admin level users.

A differentiated login was set up, where different users are directed to different web pages depending upon their privileges. Normal users had the capability to alter certain fields only pertaining to them, while admin level users(bank employee etc) had capabilities to alter all fields for all users. This sort of setup can be further differentiated to account for multiple levels of access,e.g USER, CLERK, MANAGER etc.

Essentially, a sort of “authorization engine” was developed, which consisted of several Java Controllers in conjunction, through which all requests were passed. This system carefully analysed the roles available to each user, and so determined whether the resource requested can be furnished or not.

A similar approach was applied to the DayTrader app. DayTrader is a benchmark application built around the paradigm of an online stock trading system. Originally developed by IBM as the Trade Performance Benchmark Sample, DayTrader was donated to the Apache Geronimo community in 2005. This application allows users to login, view their portfolio, lookup stock quotes, and buy or sell stock shares.

Daytrader had been set up in such a way that any user could get access to other users’ information by carefully modifying the http request in the browser. The app was secured by employing a similar setup of a differentiated login and passing all requests through these controllers. It was observed that this way these flaws in the original app were rectified.

Further,Spring profiles were used to easily switch between different configurations. Every enterprise application has many environments, like: DEV,TEST,PROD. Each environment requires a setting that is specific to them. For example, in DEV, we do not need to constantly check database consistency. Whereas in TEST and STAGE, we need to. These environments host specific configurations called Profiles.

Spring profiles helped to easily switch between vulnerable and secured versions of the app. They were also used to specify multiple different configurations of access privileges.

4.2 SQL injection

The Vulnerable Web Application by OWASP was used to simulate an environment. Vulnerable-Web-Application is a website that is prepared for people who are interested in web penetration and who want to have information about this subject or to be working.

Vulnerable-Web-Application categorically includes Command Execution, File Inclusion, File Upload, SQL and XSS. For database-requiring categories, it creates a database under localhost with one button during setup.

The SQL section of the app is primarily based in PHP. So, the XAMPP web server solution was used to execute this app.

To guard the app against SQL injection,a number of steps were taken:

1. All user input was thoroughly validated and sanitized before passing into database query.
2. All queries containing user input were executed using prepared statements like parameterized queries.
3. PHP Data Objects(PDOs) were used to extend this approach for databases other than MYSQL. Database specific PDO drivers were

used to expose database-specific features as regular extension functions.

Following these steps, the app was secured against all direct SQL attacks.

5 Further Scope

- Integrating the 2 approaches so as to get a single framework for applications handling database queries
- Creating a standalone framework which could be used for any other app
- Extending these approaches to apps in other coding environments.

Bibliography

1. Ayal Tirosh,G00340359,Hype Cycle for Application Security,2018
2. OWASP Top 10 Web Application Security Risks
3. OWASP API Security Top 10 2019
4. Inon Shkedy,Medium,A Deep Dive On The Most Critical API Vulnerability — BOLA (Broken Object Level Authorization)
5. Anand Prakash,AppSecure,How I could Have hacked your Uber account
6. Sarmad Hassan,Medium,Disclose private attachments in Facebook Messenger Infrastructure
7. Rod Johnson , Juergen Hoeller,et al,Spring Framework Reference Documentation,5.0.0.M1.
8. Mehdi Achour, Freidhelm Betz,et al,PHP Manual 2020
9. OWASP SQL Prevention Cheat Sheet.