Justin Strelka

Exploit\_web\_server

7/9/20

**Ethically Hacking a Vulnerable Web Server**

**Table of Contents:**

Abstract…………………………………1

Motivation…………………………….1

Introduction………………………….1

Method/Measurement………….2

Results…………………………………..5

Conclusion…………………………….8

Bibliography………………………….9

Reflection……………………………..9

**Abstract:**

This paper begins with a brief explanation of the importance of ethically hacking owned web servers with permission. Next there is a description of the vulnerable web server we will be using for the purposes of this paper. Following is an outline of the top 10 application security risks along a brief description of each style of attack. The paper concludes with a demonstration of three attacks on the top 10 security list (2 SQLi and 1 XSS attacks) along with a description of what happened in each exploit and why the exploit is dangerous.

**Motivation:**

This paper is intended for students that are interested in understanding how applications are at risk for being exploited. You will be able to understand and describe the most common application security risks as provided by the Open Web Application Security Project (OWASP). Understanding how the 10 most common exploits will be conducted against your applications will help you implement robust design choices for development and assist in patching your application in the future when exploits are found. Lastly, you will be able to test exploits of your own against known applications that you have permission to test exploits against.

**Introduction:**

Understanding common vulnerabilities can help us design robust and secure applications that provide professional service to the end users. Vulnerable web applications can have catastrophic effects on a company’s reputation along with extensive costs for rectifying damages. When a web application is exploited it can affect more than just the accessibility and lost revenue for a business; it can affect the end users by leaking personally identifiable information (PII) which can have cascading effects to the individual. It is our duty as developers and cybersecurity professionals to provide safe secure services to the end users.

**Method/Measurement:**

Before we begin attempting to exploit a known vulnerable web application, we need to first understand how web applications are attacked. The Open Web Application Security Project (OWASP) tracks the top 10 application security risks and publishes them at the following link: <https://owasp.org/www-project-top-ten/>. This is a great place for us to begin our understanding of how applications can be vulnerable and how we can protect against these vulnerabilities.

The number one vulnerability on the OWASP top 10 list is exploitation through use of injection. Injection attacks are defined as the use of untrusted data being inserted into a command by a malicious user that goes unflagged as malicious code by the interpreter. [1] The malicious command can then be executed by the system which can have effects that disrupt data or extract sensitive data without proper authorization.

The second most common vulnerability outlined on the OWASP top 10 list is broken authentication which is the misconfiguration of authentication services. The misconfiguration of authentication services allows malicious user to compromise passwords, trusted keys, or session tokens to assume the identity of other users. [1] Broken authentication attacks allows malicious users to obtain the privileges granted to the authorized user which may provide access to other system resources to escalate or pivot an attack.

The third vulnerability listed is sensitive data exposure which is caused by a flaw in an applications API to properly protect sensitive data. When data is unencrypted at rest or in transit malicious users can exfiltrate sensitive data such as personally identifiable information, financial accounts, or health records. [1] This information is then later used to conduct other crimes against an individual or business. An obvious way to protect against such attacks is to ensure all sensitive data is encrypted at all times during rest and transit.

The fourth vulnerability is XML External Entities (XXE) which allows for malicious users to exploit poorly configured XML processors. An XML processor will parse XML documents and provide users access to their content and possibly other sensitive resources. [1] When poorly configured a malicious user can use the processor to gain access to internal file shares, port scanning, or remote code execution. When a malicious user can gain access to port scanning their likelihood of finding other vulnerabilities within the system to escalate the attack increases.

The fifth exploit is broken access control which is caused by flaws in directory access control. A common service that is used for directory access management is the lightweight directory access protocol (LDAP). The LDAP protocol allows administrators to create groups and roles that contain rules outlining the privileges a user assigned to specific group or role has. When these rules have flaws containing unauthorized privileges attackers can use theses privileges to escalate their attacks to other system resources.

The sixth attack is security misconfigurations which can appear in all parts of a system and is the most common cause of a successful exploitation. Security misconfigurations can appear due to insecure default configurations such as the default firewall settings in most Linux distributions (default accept). Other misconfigurations can appear from old packages that have not been updated and patched allowing for malicious users to use exploitation frameworks such as Metasploit to easily exploit a system. [1]

The seventh common exploit is cross-site scripting (XSS) which occurs when a malicious user imports untrusted data into a new or existing web page. The untrusted data can then be validated within an end user’s browser to redirect the user to a malicious web site of the hackers choosing or import code onto the end user’s system. [1] Code that is imported into an end user’s system is then later executed and can cause undesired behavior of the system for malicious purposes.

The eighth vulnerability on the OWASP list is insecure deserialization. The most common forms of serialized objects are JSON formatted files which are used to store or send data through communication channels. [1] Many programming languages offer customizable serialization techniques. However, custom techniques and even standard techniques for deserializing serialized data can be error prone. Such flaws can be repurposed for malicious attacks against a system for access control or even remote code execution.

The ninth vulnerability listed by OWASP is the use of components with known vulnerabilities such as libraries, frameworks and other pre-defined software packages. When buggy software packages are running inside of an application the package receives the same privileges as the application does. This means that if the application is ran with root or administrator privileges the buggy software package can also access root or administrator files and commands. [1]

The tenth and final vulnerability on the OWASP top ten list of security risks is insufficient logging and monitoring inside of an application. Log monitoring is an effective way of identifying and mitigating vulnerabilities. However, log monitoring is only as good as the logs provided for a given application. When attackers go unnoticed within the logs of an application this allows them time to gain a more secure foothold to pivot other attacks from and remain within a system.

These ten are just the most common and generic ways web applications are and can be attacked. This is not a comprehensive list of all the possible ways a web application can be exploited. Though now that we have an idea of the ways a web application can be vulnerable we can begin trying to implement these exploits against a known vulnerable web application of our own. For the purposes of this paper we will be using the WebGoat application provided directly by OWASP. The application can be retrieved from the following link: <https://github.com/WebGoat/WebGoat>.

We will be running WebGoat through docker on a Kali Linux VM running on VirtualBox. Docker can be installed on our Kali Linux VM using the documentation provided by kali.org at the following link: <https://www.kali.org/docs/containers/installing-docker-on-kali/> . Once docker has been installed we will clone the github repository for the WebGoat application onto our Kali Linux VM. As shown in **Figure 1** after installing Docker and cloning the WebGoat github repo be sure to set the network adapter for the Kali VM to host only adapter since the WebGoat application is designed to be vulnerable.

A screenshot of a social media post

Description automatically generated

Figure 1: Change Kali VM VirtualBox network adapter setting to host-only for security purposes.

We will now initiate the WebGoat application using docker a container to run and manage the application. First we must navigate to the directory we cloned the WebGoat repository to. Once we have reached the directory containing the WebGoat file we can run the following command to build the docker image: docker run -d -p 80:8888 -p 8080:8080 -p 9090:9090 -e TZ=America/Denver webgoat/goatandwolf:latest. The vulnerable web application WebGoat will be running at 127.0.0.1:8080/WebGoat and the WebWolf application will be running at 127.0.0.1:9090/WebWolf as shown in **Figures 2 & 3.** Go ahead and create a user profile on the WebGoat server and the same credentials will be used on the WebWolf server as well.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

Figure 3: WebWolf Server login page at 127.0.0.1:9090/WebWolf

Figure 2: WebGoat Server login page at 127.0.0.1:8080/WebGoat

Once you have created a WebGoat account on login to the application and on the left you will see a column with a list vulnerabilities that correspond to the top ten list we covered previously. We will be conducting two SQL injection attacks that can be researched under the A1 Injection tab and one Cross-Site Scripting (XSS) attack found under the A7 Cross-Site Scripting (XSS) tab. We will begin by performing two SQLi attacks against the SQL structured database provided by WebGoat.

**Results:**

A screenshot of a cell phone

Description automatically generatedThe first SQLi exploit we will conduct will demonstrate the inherent vulnerabilities that dynamic queries present. The query in the example appends a number to the end of the query as a variable which allows us to build a vulnerable query with unintended results. The original query expects two values to be input in the provided text fields one for “Login\_Count:” and the other for a “User\_id:”. The vulnerable input is the “User\_id:” input since it is at then end of the query and we can manipulate the amount of data we place in the text field. The original condition makes use of an and clause meaning that both conditions need to be satisfied to return any data. We can use the “OR” clause in out input for “User\_Id:” accompanied by a tautology such as “1 = 1” to make the condition true for every line on the table and return all lines within the table. As you can see in **Figure 4,** we fill in the first value “Login\_Count:” with a valid input such as 3. Though at the “User\_id:” input we inject our tautology to manipulate the query by entering “0 OR ‘1’ = ‘1’”.

Figure 4: Successful manipulation of the Select data function returning entire table.

Our next SQLi attack will target the ability to manipulate the data within an SQL structured database. We can link queries together in the input fields by using the end of command SQL operator “;” and adding more commands to be executed at the end of the intended query. As cybersecurity specialists it is very important to understand that not all users will use the system as intended and ensuring that we sanitize our queries prior to deploying then is of upmost importance. When a single value is expected to be entered in a text field, we should never allow the user to execute a query containing the end of command character “;” as the intention is either accidental or malicious. As you can see in **Figure 5** the vulnerable field is yet again the concluding text box. We will enter our name into the Employee name field and our TAN number into the Authentication TAN field. However, we will append a second command to be executed by the database after the return of the first command by daisy chaining commands in the Authentication TAN field with the “;” operator. The command we will place in the Authentication TAN field will be: “3SL99A'; UPDATE employees SET salary = 99999 WHERE first\_name = 'John' AND last\_name = 'Smith”. As you can see in **Figure 5** the query executed successfully with unintended results, we were able to update our salary information without authorization. To defend against this we must sanitize our queries prior to execution or limit the privileges users have and the commands that can be executed by the user within the DBMS.

A screenshot of a cell phone

Description automatically generated

Figure 5: SQLi attack that updated table data without authorization compromising integrity.

Our final attack agains the WebGoat application will demonstrate the how a malicious user can extract or manipulate information inside a web browser using privileged functions in JavaScript. This kind of attack is known as Cross-Site Scripting (XSS) and it is dangerous because it can implement malicious code against an endusers device undetected. These kinds of attacks can infect a large amount of endusers very quickly depending upon how many users access the compromised site. Some of the common places these kinds of attacks spawn from are message boards, free form comments on blogs, and error messages that contain user supplied data which can be intercepted.

We will be attempting to conduct a DOM-Based XSS attack against our WebGoat server. To do so we will need to navigate a custom route within the application. Often times test routes are left inside of web applications and can be sniffed out by malicious users within the browsers develper tools. We will be producing this attack from within the Firefox browser (NOTE THIS IS NOT A BROWSER VULNERABILITY, IT IS ATTRIBUTED TO THE WEBGOAT APPLICATION). First we must open the “Developer Tools” within the browser, to do so with the Firefox browser we can simple press CTRL + SHIFT + K. The developer tools will automatically appear and we can navigate to the Debugger tab and then search for signs of a file that may handle routes. After searching we can see the GoatRouter.js file. Inside the GoatRouter.js file we can find a dictionary called “routes”. As seen in **Figure 6,** inside the routes dictionary we see test/:param labled as a testRoute which most likely does not have the applications security tools build A screenshot of a social media post

Description automatically generatedinto it.

Figure 6: Firefox web developer tools displaying targeted “testRoute”.

We can exploit this route by appending it to the start of the route in our web browser. The initial route for level 10 is <http://127.0.0.1:8080/WebGoat/start.mvc#lesson/CrossSiteScripting.lesson/10>. If we alter the route in a new browser window to access the test route we found earlier, we get the following URL to navigate to: <http://127.0.0.1:8080/WebGoat/start.mvc#test/params>. Where we see the params identifier in the URL we can replace it with malicious javascript given by the instructions in WebGoat on level 10. The newly formulated URL would be: <http://127.0.0.1:8080/WebGoat/start.mvc#test/%3Cscript%3Ewebgoat.customjs.phoneHome()%3C%2Fscript%3E>. As you can se we now injected a custom javascript function that will execute when the browser navigates to this path. As shown in **Figure 7,** inside the newly routed browser open the developer tools again and look at the javaScript console to see we have invoked the phone home function and we got an output of the random number required to complete level 10 of WegGoat.

A screenshot of a social media post

Description automatically generated

Figure 7: Display of the altered route executing customjs.phoneHome() function.

**Conclusion:**

We have discussed the 10 most common application vulnerabilities as provided by OWASP and demonstrated the exploitation of 2 kinds of exploits. We have displayed that databases can easily be attacked within an application to either extract or manipulate data. We have also shown that it is possible to execute malicious java script inside of an application from a standard web browser. Both of these kinds of attacks can be reduced or mitigated by ensuring the data going into our applications has been scanned for malicious looking code that can be flagged prior to execution. By doing so we can save our company and the end users future loss of revenue or other assets. It is our duty as cybersecurity specialists to understand how our systems and applications can and will be attacked so we can design our systems to be robust.

**Bibliography**

[1] <https://owasp.org/www-project-top-ten/>

**Reflection:**

Prior to writing this paper I was unaware of SQLi attacks even though I work with SQL databases almost daily at my internship. I have gained skills in writing complex queries however it did not occur to me that the language could be used in a malicious way. Obviously I knew dropping a table is very easy to do though my system admins I believe have been diligent on only allowing minimal necessary privileges. I was highly surprised to learn about XSS attacks and the ability to execute malicious java script from the browser. Moving forward I will be more cautious and diligent about identifying possible security risks in places that are not always clear. This demonstration has made me hesitant to browse the open web and supply any credentials anywhere. I will be creating unique passwords for every account I create and storing them in KeePass.