WebGoat Penetration Test Findings

Group Member Responsible: Bradley Zust

Table of Contents

Purpose of Test

The purpose of this penetration test is to demonstrate and gain experience in running a penetration test on a Web Application, so that a future penetration test can be conducted on the Time To Study Web Application.

Test Setup and Legal Statement

No penetration test should ever be conducted without the express written consent of the system owners. In this specific case, the test was conducted on virtual machines on my personally owned systems, disconnected from the internet. The setup was one Attacker system, which was running Kali Linux, and the vulnerable system that is used to run OWASP WebGoat. The virtualization product used for this Test is VirtualBox, with a software version of 7.0.6. More information on the specifics of the technology used is visible in Table 1.

What is OWASP WebGoat?

OWASP WebGoat is a deliberately insecure application that allows developers to test vulnerabilities commonly found in Java-based applications that use common and popular open source components. WebGoat is developed by the Open Worldwide Application Security Project (OWASP) as a way to improve the security of software by increasing awareness and education of various common vulnerabilities found in applications. Previously, A document was submitted to Time To Study which described the OWASP Top Ten Document. As both the OWASP Top Ten and OWASP WebGoat are maintained by the same organization, it is not then a surprise that OWASP WebGoat serves as an example of those vulnerabilities in practice.

Penetration Test

Table 1, Technology/Systems used:

Name	Software Version	Purpose	Notes
VitualBox	7.0.6 r155176 (Qt5.15.2)	Virtualization Software	To create and manage the virtual machines
Ubuntu	22.04.2	Operating System for WebGoat system	
Kali Linux	2023.1	Operating System for the attacker machine	

Table 1.1 Detailed Virtual Machine Setup Information:

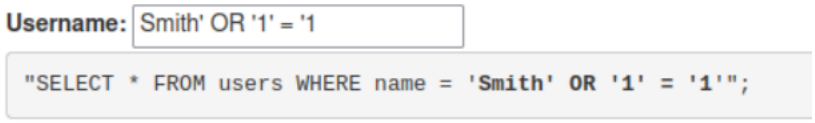
Label	Ubuntu Machine	Kali Linux Machine
Name	WG User	Kali-linux-2023.1-virtualbox-i386
OS	22.04.2	2023.1 (32 bit)
Username	guest	kali
Password	password	kali

Base Memory	4096 MB	2048
Processors	4 CPUs	2
Hard Disk	20 GB	80 GB

Injection

WebGoat begins with SQL injections. SQL injections are one of the most common web hacking techniques on its own, and the most popular type of injection. SQL Injections occur when user input is not properly sanitized, meaning that malicious SQL queries can be input to user input sections like a Username or Password box, and instead of being treated like text, the system will interpret it as an actual SQL query. Figure 1 is an example of just this within WebGoat:

Figure 1: User Input Field



In Figure 1, it shows what the user sees, and how it is read by the application. By putting that line into the username, it completes the SQL statement, and will return all entries from the users table. Another example would be if the following was typed into the 'username' field:

Smith'; DROP TABLE users; TRUNCATE audit_log; --

That example chains together multiple SQL commands to both drop the users table, and delete all entries from the audit_log table. Although not all databases support command chaining, it does show what is possible with a clever and knowledgeable attacker, making Defense-in-depth strategies even more important. Injections are particularly dangerous, as they can damage all 3 parts of the CIA triad: Confidentiality, Integrity, and Availability.

Confidentiality is fairly obvious from Figure 1, by being able to access information in the database without being authenticated, or without the proper authorization, destroys the confidentiality of the effected database. Attackers can not only view the data in the database, but also edit it, or delete it altogether, harming integrity. Finally, accounts can get edited or deleted within databases, making it so that the people who should have access, no longer do, harming availability.

Broken Authentication

One way that attackers can utilize Broken Authentication is through authentication bypasses. Authentication bypasses normally take advantage of some kind of flaw in the configuration or logic and can be tampered with to achieve the correct conditions for success. This can be from guessing/brute forcing, for example, Kali Linux uses kali/kali as its default

username/password, so if I were trying to access a Kali machine, I could try that. Another way that attackers can bypass authentication is by removing the parameter blocking them altogether, and seeing how the system reacts. Finally, the simplest is if the webpage has a reliance on a hidden input that is in the web page/DOM. WebGoat provides an example of how PayPal's two-factor authentication was bypassed, by removing all parameters using a proxy, including the security questions, and was able to get in.

Another way that broken authentication can be taken advantage of, is through JSON Web Tokens (JWT). JWTs can be used to give attackers information, such in Figure 2, where I decoded the JWT token to find the username.

Figure 2: JWT Token decoded:

JWT tokens should be signed to verify that the tokens are not changed, otherwise an attacker can simply change the tokens to whatever they want to allow access to administer or other valuable accounts. Best practice with JWT tokens is to:

- Not allow a client to switch the algorithm.
- Ensure an appropriate key length is used for signing the token when a symmetric key is used.
- Ensure the claims added to the token do not contain personal information. If more information is needed, encrypt the token.
- Add sufficient test cases to ensure that invalid tokens do not work, even when integrated with third party.
- Ensure your usage aligns with other industry best practices.

Another important part of the authentication process that can be used to bypass authentication is the password reset functionality. On many sites, there are different messages if the email exists, or if the email doesn't exist. This is a security flaw, as the attacker can use the password reset functionality to test various emails to see if there is an account associated

with it. It would be better to have the same message no matter if the email is registered or not, so that attackers are unable to use it to test email addresses. Additionally, many websites use security questions that are fixed, and have a limited number of answers. The example given in WebGoat is the security question: “What is your favorite color” which although you do not know what the admin’s favorite color is, there are a limited number of answers, making it easy to guess. In fact, I was able to guess it correctly on the first try with “Green” showing how dangerous this type of security question can be. Finally, it is important to ensure that password reset links are unique, only usable once, and expire after a limited amount of time, to reduce the chance of a Denial-of-Service attack occurring by abusing the ‘reset password’ link.

Furthermore, when it comes to passwords themselves, it is important to utilize the NIST password standard to implement secure password systems. Some recommendations made by NIST are:

- No composition rules. Do not request the user to for example use at least one upper case letter and a special character on their password. Give them the opportunity to, but do not force them!
- No password hints. If you wanted people to have a better chance at guessing your password, write it on a note on your screen.
- No security questions. Knowledge-based Authentication (KBA) are outdated and is insecure.
- No unnecessary changing of passwords. If you want a user to choose long, hard-to-guess passwords, you should not make them change those passwords unnecessarily after a certain period of time.
- Minimum size of 8 characters. A secure password nowadays should be at least 8 characters to 64 characters long.
- Support all UNICODE characters. You should allow all kind of UNICODE characters in a password, including emojis and whitespaces.
- Strength meter. Add a strength meter on the password creation page to help the user to choose a strong and secure password.
- Check the password against known bad choices.

Finally, it is important to properly secure stored passwords. Encryption and a protected channel for requesting passwords should be used.

Sensitive Data Exposure

Data should be encrypted. Otherwise, software such as packet sniffer can be used to obtain the data in transit, and if not encrypted, can easily provide attackers with sensitive materials.

XML External Entities

An XML External Entity (XXE) injection is a type of attack against an application that parses XML input. An XXE injection attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser, which can lead to the disclosure of confidential data, denial of service, server-side request forgery, port scanning from the perspective of the parser's machine, and other system impacts. There are 3 main kinds of XXE attacks, which are:

- Classic: an external entity is included in a local DTD
- Blind: no output or errors are shown in the response
- Error: try to get the content of a resource in the error message.

As Time To Study does not utilize XML, I will not spend as much time on this section, but it is still important, and I would encourage the rest of the group to explore this part of WebGoat, as well as all the other sections.

Broken Access Control

One way that Access Control can break is through Direct Object References being insecure. Direct Object References are when an application uses client-provided input to access data and objects, and can look like:

https://bradstestwebsitepleaseignore.com/dor?id=12345

or

https://bradstestwebsitepleaseignore.com/images?img=123445

Direct Object Refences are considered insecure when the reference is not properly handled and allows for authorization bypasses or disclose private data that could be used to perform operations or access data that the user should not be able to perform or access. For example:

If your user account is:

https://bradstestwebsitepleaseignore.com/app/user/25678

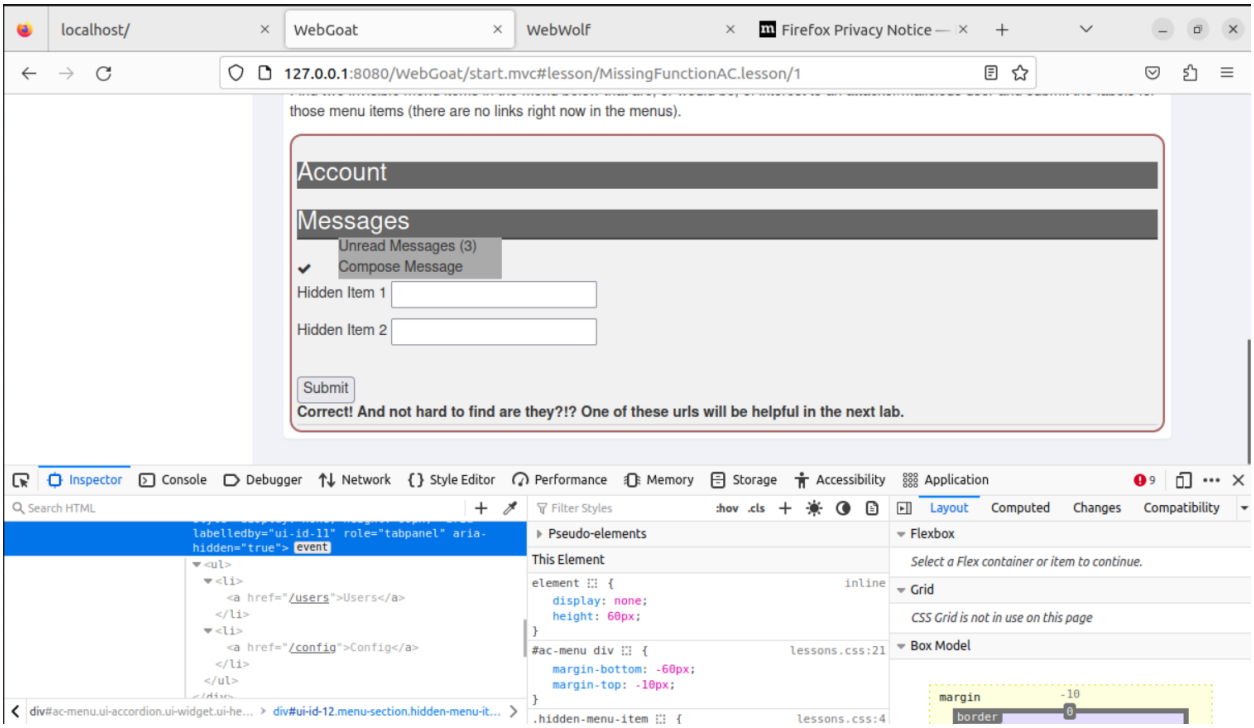
than you can type:

https://bradstestwebsitepleaseignore.com/app/user/25679

or any other number to view another' profile.

Some rely on HTML, CSS, or javascript to hide links that users don't normally access, but they are often not difficult to find. For example:

Figure 3: Finding Hidden Items



In Figure 3, you can see that the hidden links are easily found in the HTML file by just using the browser's inspect element functionality.

Cross-Site Scripting (XSS)

Cross-Site Scripting is the most prevalent and pernicious web application security issue. XSS is a vulnerability/flaw that combines the allowance of html/script tags as input that are rendered into a browser without encoding or sanitization. The most common locations for XSS are:

- Search fields that echo a search string back to the user
- Input fields that echo user data
- Error messages that return user supplied text
- Hidden fields that contain user supplied data
- Any page that displays user supplied data (message boards and free form comments)
- HTTP Headers

And XSS attacks may result in:

- Stealing session cookies
- Creating false requests
- Creating false fields on a page to collect credentials
- Redirecting your page to a "non-friendly" site
- Creating requests that masquerade as a valid user
- Stealing of confidential information
- Execution of malicious code on an end-user system (active scripting)

- Insertion of hostile and inappropriate content
- Add validity to phishing attacks (by using a valid domain in the URL)

The main types of XSS are Reflected, DOM-based, and Stored or persistent.

Reflected XSS requires social engineering, and results in malicious content from a user request being displayed to the user in a web browser, and then is written into the page after 'from' server response. This type runs with browser privileges inherited from user in browser.

DOM-based XSS is also technically reflected XSS, but client-side scripts are used to write HTML to its own page instead.

Stored or Persistent XSS is when Malicious content is stored on the server, within the database, file system or some other object, and later displayed to users in a web browser. Unlike Reflected XSS, social engineering is not required.

Insecure Deserialization

Serialization is the process of turning some object into a data format that can be restored later. This is most often done to save it to storage, or to send as part of communications. Deserialization is the reverse of that process, which takes data structured from some format, and rebuilding it into an object. The most common data format for serializing data is JSON. Many programming languages offer a native capability for serializing objects, which offer more features than JSON, but can be repurposed for malicious effect when operating on untrusted data.

Vulnerable Components

As applications use more and more components, most of which are open-source, it is difficult to sort through and figure out which components are vulnerable. It is not only important to keep track of what components are used, but also their version, and when to update the components. For open-source components, it is important to know that the component is an authentic copy, and if not, why it is modified, and if it is exploitable. Use of automated tooling is very important to not only keep track of what components are being used, but also which ones are vulnerable, and which must be updated.

Problems Encountered

The main problem I encountered early on had to do with my initial WebGoat machine. Initially I used a Debian installation, but over the course of trying to get WebGoat up and running, I kept on running into issues of various packages not installing correctly or being unable to be found. Eventually, I gave up on that installation when both my java installation ended up being outdated (although I had just installed it), and docker would not install. That is why the machine that uses WebGoat is listed as an Ubuntu installation, as I remade the VM with an Ubuntu iso instead.

Try it for Yourself!

OWASP WebGoat is a great tool to learn about these common vulnerabilities and goes far more in depth than I did within this document. I would highly recommend any developer to try it out. It can be used for no additional cost on existing hardware by using a virtual machine, as I did for my examples. It ranges from an introduction to the various vulnerabilities, to advanced lessons and mitigation strategies. It can be acquired from the OWASP Website, or the WebGoat Github page. I recommend using a Linux distribution and utilize docker to install and run WebGoat and WebWolf (the attacker variant of WebGoat). There are instructions on both the Website and Github page on how to do just that. However, do heed the warnings on both pages and the application itself about how to safely run the application, as running a deliberately insecure application does make your machine insecure. However, do not let that stop you, as with all security warnings heeded, it is a great tool to learn about application security.

Conclusion

WebGoat is a great tool to interact with and learn more about the OWASP Top Ten and application vulnerabilities. The fact that it is free to use, and easy to learn makes it a valuable tool to developers and cybersecurity practitioners alike. As with the OWASP Top Ten Document, this document should not be seen as the final word on security, or security issues that may be present within Time To Study, but as a stepping stone in developing habits and policies to continue to secure and maintain Time To Study.