WARATEK

Waratek Security Rules Best Practices

Document version: 2
Last update: Aug 20th, 2024
Edited By: Keegan Henckel-Miller

www.waratek.com sales@waratek.com +1 872 469 8605

Overview

Welcome to the Best Practices guide to the Waratek Java Security platform — the last and only security platform you will need to ensure your Java applications are fully protected against both known threats and zero days. Waratek's advanced Runtime Application Self-Protection (RASP) platform works by integrating your application code base with immutable rules that are individually designed to detect and remediate vulnerabilities without the need for changes to the code base or system downtime. This allows the platform to deliver complete coverage on your java systems with negligible effects on application performance (less than two percent.)

In this guide, we'll take a look at some of the most common avenues attackers take to exploit Java systems and explain how Waratek detects and remediates those threats before they ever come near sensitive systems or data. These include deserialization attacks, session fixation, CSRF, open redirects, path traversal and LFI, cross site scripting and other common vulnerabilities.

We'll also offer some tips to aid your setup process and help you get the most out of your Waratek protection. You'll learn why Waratek is the best choice for securing your applications, ensuring compliance, and keeping your data safe. So grab your coffee, settle in, and let's get started making your Java applications bulletproof.

About Waratek

Some of the world's leading companies use Waratek's Java Security Platform to patch, secure and upgrade their mission critical applications. A pioneer in the next generation of application security solutions, Waratek makes it easy for security teams to instantly detect and remediate known vulnerabilities with no downtime, protect their applications from known and Zero Day attacks, and virtually upgrade out-of-support Java applications – all without time consuming and expensive source code changes or unacceptable performance overhead.

Waratek is the Cybersecurity Breakthrough Awards 2019 Overall Web Security Solution of the Year, is a previous winner of the RSA Innovation Sandbox Award, and more than a dozen other awards and recognitions. For more information, visit www.waratek.com.

Table of Contents

Unsafe Deserialization of Untrusted Data

Vulnerability Overview

Deserialization of untrusted data (CWE-502) — also called insecure deserialization — occurs when applications deserialize data from untrusted sources without sufficiently verifying that the resulting payload will be valid and therefore the in-memory object will be safe to use. This is considered to be one of the most damaging types of attacks, potentially leading to a complete compromise of a vulnerable system. To make matters worse, deserialization attacks have become one of the most widespread security vulnerabilities to occur over the past few years. One study by the SANS Institute found that in a sample of 50 web applications, 40% were vulnerable to insecure deserialization attacks. The researchers also found insecure deserialization vulnerabilities present in 34 percent of Java applications.

Serialization is the process of converting an object in memory into a stream of bytes in order to store it into the filesystem or transfer it to another remote application. Deserialization is the reverse process that converts the serialized stream of bytes back to an object in memory. All main programming languages, such as Java and .NET, provide facilities to perform native serialization and deserialization. This makes them the most vulnerable to attack. A deserialization attack is designed to create a gadget chain that will reach these privileged platform functions and execute the payload on the system. The payload could abuse the filesystem, the operating system, or system resources.

Deserialization vulnerabilities have been a significant security concern in Java applications for over a decade. The issue gained prominence with the discovery of several high-profile vulnerabilities, notably in widely-used Java libraries like Apache Commons and the Spring Framework. Deserialization vulnerabilities occur when untrusted data is used to reconstruct an object in memory. If not properly validated, this process can be exploited by attackers to execute arbitrary code, resulting in serious security breaches.

Deserialization vulnerabilities are not limited to language deserialization APIs but also encompass libraries that make use of other serialization formats such as XML and JSON. The attack process can be summarized in the following steps:

1. **A vulnerable application accepts user-supplied serialized objects.**
2. **An attacker performs the attack by:**
    a. **creating a malicious gadget chain (sequence of method calls)**
    b. **serializing it into a stream of bytes using the serialization API**
    c. **sending it to the application**
3. **Deserialization occurs when the vulnerable application reads the received stream of bytes and tries to construct the object.**
4. **When a malicious object gets deserialized, the gadget chain is executed and the system is compromised.**

Recommended Security Controls

According to the CERT and MITRE recommendations, to be protected against Deserialization attacks, applications must:

- Minimize privileges before deserializing from a privileged context.
- Not invoke potentially dangerous operations during deserialization.

How Waratek's Protection Works

The goal of deserialization is to convert a stream of bytes into an object in memory. The runtime platform (e.g. JVM) should allow this conversion but should not allow more privileged operations that are outside of the scope of the object deserialization API. Deserialization attacks depend on invoking API methods that are considered to be privileged, such as java.lang.Runtime.exec(), in order to perform an attack.

Securing against deserialization vulnerabilities is particularly challenging because these vulnerabilities often lurk deep within the application stack, making them difficult to detect and mitigate with perimeter-based solutions like Web Application Firewalls (WAFs). Unlike RASP (Runtime Application Self-Protection) solutions that operate within the application, WAFs primarily focus on monitoring and filtering traffic at the network edge. This external positioning limits their ability to detect and prevent attacks that exploit internal application logic, such as deserialization vulnerabilities.
In accordance with the CERT, MITRE and OWASP recommendations and observations, Waratek protects against deserialization attacks (CWE-502) by addressing the problem from a privilege escalation (CWE-250) and an API abuse (CWE-227) point of view.

The issue with serialization is that until a payload is reconstructed, there is no good way to know whether or not it contains harmful code. Our platform gets around this obstacle by executing the operation in a test environment to observe what happens without the risk of a full-on attack. On specific object deserialization operations (called boundaries), the Waratek agent constructs a dynamic restricted micro-compartment on the execution thread and continues the object deserialization inside it. Waratek de-escalates the privileged operations in the micro-compartment and monitors the usage of resources. If a privileged function is invoked inside the micro-compartment, the execution is terminated and the payload is not executed. The same logic applies for Denial of Service attacks, if resources are abused inside the micro-compartment, then the deserialization process will be terminated and the attack will be prevented before the system resources are exhausted. The micro-compartment is destroyed on a non-malicious object deserialization completion and privilege de-escalation is revoked on the executing thread.

Waratek's protection capabilities support popular deserialization APIs and formats that can be used across the application. Additionally, the Waratek agent is able to protect against attacks regardless of the untrusted source. For example, if the serialized data is coming from an HTTP client (such as an external web request) or data coming from another internal system (such as a message queue), it is terminated all the same before it is able to cause any harm within your application.

This process does not rely on previous knowledge of publicly available gadget chains and exploits. That key element saves our users a lot of time profiling their applications' new functionalities when changes are made. You can deploy new versions of your applications without having to make any adjustments to your protection mechanisms.

Waratek offers protection against Deserialization attacks via the deserial:harden rules. Currently, there are 2 deserial rules:

1. The deserial:harden:system rule, that protects against Remote Code Execution (RCE) deserialization attacks
2. The deserial:harden:dos rule, that protects against Denial-of-Service (DoS) deserialization attacks

Enabling these rules sets up the privilege de-escalation runtime micro-compartmentalization framework. This framework monitors and controls memory allocation, CPU utilization, circular dependency depths, code injection, and privilege escalation during deserialization operations.

Protective Action

When the deserial rule is enabled in deny mode and a deserialization attack is identified, the malicious deserialization operation is terminated and a Java exception is thrown back to the application, in accordance with the deserialization API.

For example:

```
com.waratek.AllowDeserialPrivileges=java.lang.SecurityManager.<init>(),java.lang.System.getenv()
```

Best Practices

Because of the criticality of the vulnerability as well as because users typically are unaware if there are components anywhere in their Java stack, Waratek recommends enabling both deserial rules in order to be protected against both remote code execution and denial of service deserialization attacks.

References

- https://cwe.mitre.org/data/definitions/502.html
- https://owasp.org/www-community/vulnerabilities/Insecure_Deserialization
- https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487787

Session Fixation

Vulnerability Overview

Session Fixation (CWE-384) is an attack that permits an attacker to hijack a valid user session. Authenticating a user, or establishing a new user session without invalidating any existing session identifier gives an attacker the opportunity to steal authenticated sessions.

Session fixation attacks have been a well-known security issue in web applications for many years, particularly in environments where session management is handled incorrectly. This type of attack occurs when an attacker fixes a user's session ID before the user authenticates — the HTTP Session ID remains the same before and after user logs-in. This permits an attacker to “fix” a specific Session ID and hijack a valid user session. In other words, the Session ID can be controlled by the attacker.

Once the user logs in, the attacker, who already knows the fixed session ID, can hijack the session and gain unauthorized access to the user’s account. These attacks exploit weaknesses in how sessions are managed and are especially dangerous because they can occur without the user ever knowing.

Traditionally, preventing session fixation attacks has relied on perimeter-based solutions like Web Application Firewalls (WAFs) or relying on developers to implement session management best practices, such as regenerating session IDs upon authentication. However, WAFs have limited effectiveness against session fixation because they operate at the network level and lack the context needed to monitor session management processes within the application. Similarly, relying on manual coding practices to secure sessions can be error-prone, particularly in large, complex applications where sessions are managed across multiple components.

Recommended Security Controls

According to the OWASP and MITRE recommendations, to be protected against Session Fixation, applications must:

1. **Invalidate any existing session identifiers prior to authorizing a new user session**
2. **Regenerate the session ID after any privilege level change within the associated user session**

How Waratek's Protection Works

Waratek's advanced runtime protection offers a more robust and comprehensive solution to mitigating session fixation attacks. Waratek's security policies are embedded directly within the runtime environment, allowing for real-time monitoring and enforcement of session management best practices. By leveraging Waratek's Session rules, administrators can ensure that session IDs are regenerated securely upon user authentication and that sessions are properly managed throughout the application's lifecycle.

The SessionFix rules can be safely enabled across all Java applications to protect against session fixation attacks. These rules work by enforcing secure session management practices at the runtime level, ensuring that session IDs cannot be fixed or reused by an attacker. Importantly, Waratek's SessionFix rules require minimal configuration and can be deployed without disrupting your existing application infrastructure.

Waratek offers protection against Session Fixation attacks via the session:protect:regenerateSID rule. This rule hooks into the session authentication mechanism of the Servlet API and monitors user authentication processes. When a user successfully authenticates then the Waratek agent regenerates the Session ID of the user's HTTP Session. This proactive security control remediates the vulnerability and eliminates the attack surface for Session Fixation attacks. Because no Session Fixation attacks are possible, this rule does not log any security events for attacks.

Let's examine a high-level description of the Session ID regeneration workflow that Waratek performs:		1 User enters the correct credentials.
2 System successfully authenticates the user.	3 Existing HTTP session content is moved to temporary cache.	4 Existing HTTP session is invalidated (HttpSession.invalidate()).
5 A new HTTP session is created for the user. (HttpSession.invalidate()).	6 The previously cached session data is restored into the newly created HTTP session.	7 The user goes to a successful login landing page using a new session ID.

Protective Action

The Session Fixation rule is a proactive rule. It is triggered proactively before a potential attack and eliminates the attack vector for Session Fixation.

Best Practices

We recommend that users enable the XSS security rule even in blocking mode to be protected against XSS attacks. Applications vulnerable to XSS attacks are still vulnerable to Session Fixation or Session Hijacking attacks even if the Session Fixation rule is enabled.

References

https://owasp.org/www-community/attacks/Session_fixation
https://cwe.mitre.org/data/definitions/384.html

Cross-Site Request Forgery

Vulnerability Overview

Cross-Site Request Forgery or CSRF (CWE-352) is an attack that occurs when the web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent HTTP request was intentionally provided by the user who submitted the request. When this happens, an attacker can trick a user into executing unwanted actions on a web application where the user is authenticated.

These attacks work because browser requests automatically include all cookies including session cookies. Therefore, if the user is authenticated to the site, the site cannot distinguish between legitimate requests and forged requests. This is particularly dangerous because it can lead to unauthorized transactions, data theft, or even full account takeover, all without the user's knowledge.
CSRF attacks have been a persistent threat in web security for over a decade. Traditionally, risk in this area has been mitigated by implementing anti-CSRF tokens, validating referer headers, requiring re-authentication for sensitive actions, or some combination of these techniques.

While these methods can be effective, they are not foolproof. Anti-CSRF tokens, for instance, require proper implementation and can be bypassed if the token is not validated correctly or is exposed. Additionally, referer header validation is not reliable, as some browsers and privacy-focused users may block or omit referer headers, rendering this method ineffective. These perimeter-based solutions also rely heavily on developers correctly implementing security measures across all relevant parts of the application, which can be a challenge in large, complex systems.

Recommended Security Controls

According to the OWASP and MITRE recommendations, there are a few approaches to mitigate CSRF attacks. Each of these approaches is suitable for specific types of web applications.

The most commonly used and recommended solution is via the Synchronizer Token Pattern. Using this security control, CSRF tokens are generated on the server-side. They can be generated once per user session or for each request. Per-request tokens are more secure than per-session tokens as the time range for an attacker to exploit the stolen tokens is minimal. However, this may result in usability concerns. For example, the "Back" button browser capability is often hindered as the previous page may contain a token that is no longer valid. Interaction with this previous page will result in a CSRF false positive security event at the server. In per-session token implementation after initial generation of a token, the value is stored in the session and is used for each subsequent request until the session expires.

When an HTTP request is issued by the client, the server-side component must verify the existence and validity of the token in the request compared to the token found in the user session. If the token was not found within the request, or the value provided does not match the value within the user session, then the request should be aborted, the user session terminated and the event logged as a potential CSRF attack in progress.

CSRF tokens prevent CSRF attacks because without knowing the correct CSRF token, attackers cannot create valid HTTP requests to the backend server.

Another security control is the validation of the HTTP request's origin via standard HTTP request headers. There are two steps to this mitigation, both of which rely on examining an HTTP request header value:

1. Determining the origin the request is coming from (source origin) which can be achieved via Origin or Referer headers.

2. Determining the origin the request is going to (target origin). On the server side, if both are verified as matching, the request is accepted as legitimate (meaning it's the same origin request) and if not we discard the request (meaning that the request originated from cross-domain). Such headers are deemed reliable and

trustworthy as they cannot be altered programmatically (using JavaScript with an XSS vulnerability) since they fall under the forbidden headers list, meaning that only the browser can set them.

How Waratek's Protection Works

Waratek offers protection against CSRF attacks via 2 different rules:

1. The CSRF:STP rule (rules version 1.0)
2. The CSRF Same-Origins HTTP ARMR rule (rules version 1.3)

Users can enable either one of these rules or both. OWASP recommends using both security controls, however not all application environments are applicable for both types of security controls.

The CSRF Synchronizer Token Pattern (STP) rule

At a high-level, the CSRF:STP rule stops the processing of the JSP/Servlet if the received HTTP request is missing or carries an incorrect CSRF token. The CSRF:STP rule enables the Synchronizer Token Pattern protection, which instructs Waratek to inject CSRF tokens in specific HTML elements. The HTML elements covered are:

- <form> elements in which the token is injected as a hidden input field.
- <a> elements in which the token is injected in the URL specified by its href attribute.
- <frame> and <iframe> elements in which the token is injected in the URL specified by their src attributes.

Enabling the default CSRF STP rule ensures all HTTP POST requests will be protected by validating the CSRF token present in the requests. HTTP POST requests are the most important types of requests to protect because they are typically state-changing, whereas HTTP GET requests are typically not.

Using the CSRF:CONFIG rule, users have the option to:

Enable protection for HTTP GET requests
By default only HTTP POST requests are protected. If protection for HTTP GET requests is also required, use the csrf:config:methods rule.

Exclude / whitelist specific HTTP endpoints from protection
By default all HTTP endpoints are protected. If protection for specific HTTP endpoints must be disabled use the csrf:config:whitelist rule.

Exclude AJAX requests from protection
AJAX requests are not supported by the CSRF:STP rule because the CSRF token is not injected into client-side Javascript code that generates dynamic requests such as AJAX. AJAX requests typically carry the X-Requested-With header. If the application uses AJAX requests use the csrf:config:validateXRequestedWithHeader=false rule to disable validation for AJAX requests.

Use a different CSRF token for each HTTP method (POST / GET)
By default a different CSRF token is used for POST requests and yet another is used for GET requests. The benefit of this is to protect the CSRF token for POST requests in case the CSRF token for GET requests gets leaked. To disable this and use a single token instead then use the csrf:config:singleToken=true rule.

Rename the CSRF token used in the HTTP requests
By default the name of the CSRF token used by Waratek is "_X-CSRF-TOKEN". In the rare case where this name is used by a different HTTP parameter, then use the csrf:config:tokenName rule to rename the HTTP parameter that Waratek uses to carry the CSRF token.

The CSRF Same-Origins Rule

At a high-level, the CSRF Same-Origin rule checks if the received HTTP request is coming from a source origin different from the target origin. The source origin is determined by the Origin, Referer, or X-Forwarded-For headers. The target origin is determined by the Host or X-Forwarded-Host headers or by the hosts configured in the Waratek rule.

If the origin validation fails then the rule strips out all HTTP parameters, cookies and payloads from the HTTP request, rendering it harmless.

If none of the Origin headers are present, the origin validation cannot be performed and the rule blocks the HTTP request, according to the OWASP recommendations.

When enabling the default CSRF Same-Origins rule then all HTTP POST requests will be protected by validating the standard Origin HTTP response headers that should be present in the requests. Following is an example of the default CSRF Same-Origins rule:

```
app("CSRF Same-Origins"):
  requires(version: ARMR/1.6)
  http("Deny HTTP requests with invalid origin header (for
        all HTTP endpoints)"):
    request()
    validate(csrf: ["origins"])
    action(detect: "HTTP origin validation failed", severity: 7)
  endhttp
endapp
```

If protection is needed for specific HTTP endpoints, the specific relative URIs of the HTTP endpoints must be supplied in the request declaration of the Waratek CSRF rule. For example:

```
app("CSRF Same-Origins"):
  requires(version: ARMR/1.6)
  http("Deny HTTP requests with invalid origin header (for
        specific HTTP endpoints)"):
    request(uri: ["/path/to/vulnerablePage.jsp",
                  "/path/to/vulnerableServlet"])
    validate(csrf: ["origins"])
    action(detect: "HTTP origin validation failed", severity: 7)
  endhttp
endapp
```

Protective Action

When the CSRF:STP rule is enabled in protect mode and a CSRF attack is identified then the malicious HTTP request is terminated and an HTTP 403 response is returned to the client.

When the CSRF Same-Origins rule is enabled in protect mode and a CSRF attack is identified then the malicious HTTP request is not terminated but all its HTTP parameters and cookies are considered malicious and are therefore stripped from the request, rendering it safe.

Best Practices

We recommend you also enable the XSS security rule in blocking mode to be protected against XSS attacks. If the application is vulnerable to XSS attacks then stealing the CSRF tokens would be possible via XSS attacks. This would allow attackers to bypass the CSRF protection.

Because of the fact that the CSRF:STP rule might require some configuration, users are advised to first enable the ARMR CSRF Same-Origins rule as the first layer of defense against CSRF. Then, consider enabling the CSRF:STP rule only in relevant applications first in monitoring / allow mode and later in blocking mode after the rule has been properly configured.

Given that the CSRF Same-Origins rule depends on the presence of the Origin HTTP header, it is recommended that the CSRF Same-Origins rule is enabled only after ensuring all users are on an up-to-date browser version.

It is also recommended that users enable the ARMR CSRF Same-Origins rule only for the vulnerable HTTP endpoints reported by their vulnerability scanners.

References

- https://owasp.org/www-community/attacks/Session_fixation
- https://cwe.mitre.org/data/definitions/384.html

Open Redirect

Vulnerability Overview

Open Redirect (CWE-601) is a vulnerability that occurs when a user-controlled input is used to construct a link to an external site, and the application uses that link in an HTTP redirect without properly validating or sanitizing that input. The flaw commonly occurs when the application sets the Location HTTP response header with an unsafe value from the HTTP request. In other words, the HTTP redirect URI can be controlled by the attacker. When exploited, this flaw allows attackers to redirect users to malicious websites, typically by manipulating the redirection link within a legitimate site. This can be particularly dangerous in the context of phishing attacks, where an attacker lures users into clicking on a seemingly legitimate link that, unbeknownst to them, redirects to a malicious site designed to steal their personal information.

The consequences can be severe — users may be tricked into divulging login credentials, financial data, or personal identification details. Additionally, the reputation of the compromised website can suffer significantly, as users lose trust in its security. Moreover, attackers can use open redirects as a stepping stone for more sophisticated attacks, such as session hijacking or the distribution of malware.

Open Redirect has been a persistent issue since the early days of web development, coinciding with the rise of dynamic web applications. As websites began to rely more heavily on user-generated content and complex redirection logic, the opportunities for exploitation increased. Despite being a well-known issue, open redirects remain a common vulnerability today, as developers often overlook the importance of validating redirect URLs.

Recommended Security Controls

According to the OWASP and MITRE recommendations, to be protected against Open Redirect, applications must:

1. Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does.

2. If user input cannot be avoided, ensure that the supplied value is valid, appropriate for the application, and is authorized for the user.

The most common scenario of an open redirect attack is when the attacker redirects the user to an external, malicious, domain.

How Waratek's Protection Works

Waratek Secure for Java offers protection against Open Redirect attacks via the redirect:servlet:external rule. This rule uses the tainting engine to track all user input, hooks into the Servlet API and monitors server-side HTTP redirect operations. When an HTTP redirect operation occurs, the Waratek agent deems the redirect unsafe if both the location URI is user-controllable (tainted) and if it is external to the application's domain. Tainted redirect locations to external root domains are not allowed. The parameter "external" means that the rule detects or protects against HTTP server-side redirects to an external root domain, different subdomain or IP address from the application's. For example, if the redirect:servlet:external:deny:info rule is enabled and assuming that the application is hosted on the domain www.example.com then user-controlled server-side HTTP redirects to the following domains will be deemed malicious and blocked: www.google.com, www.example.co.uk, test.example.com, test1.test2.example.com.

If the application depends on user-controlled HTTP redirects to different subdomains of the same root domain, then the exclude=subdomains parameter must be configured in the rule: redirect:servlet:external;exclude=subdomains. Only user-controlled HTTP redirects to different root domains will be considered malicious by the rule.

For example, if the redirect:servlet:external;exclude=subdomains:deny:info rule is enabled and assuming that the application is hosted on the domain www.example.com then user-controlled server-side HTTP redirects to the following domains will be deemed malicious and blocked:
www.google.com, www.example.co.uk.

Note that user-controlled server-side HTTP redirects to the following domains will be deemed safe and allowed:
test.example.com, test1.test2.example.com.

By default, when no taint source is specified in the rule, the open redirect rule protects against attacks coming from HTTP requests. Users have the option to also enable protection against open redirect attacks coming from other sources such as relational databases and/or deserialization-based protocols such as RMI.

Protective Action

When the Open Redirect rule is enabled in protect mode and an Open Redirect attack is identified then the malicious HTTP redirect operation is terminated.

Best Practices

Waratek recommends not to enable the Open Redirect rule in blocking mode if the application depends on user-controlled server-side HTTP redirect operations to external domains. Consider enabling the rule in allow mode to monitor the server-side HTTP redirect behavior of the application and keep track of external redirects.

References

- https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html
- https://cwe.mitre.org/data/definitions/601.html

Path Traversal & Local File Inclusion

Vulnerability Overview

Path Traversal (CWE-22 - CWE-40) has become more prevalent as dynamic content and user interactions become more common in web applications, revealing weaknesses in how input was sanitized and how file paths were managed. Over the years, numerous high-profile attacks have exploited these vulnerabilities

Path traversal, also known as directory traversal, involves exploiting insufficient security validation/sanitization of user-supplied input file paths, allowing attackers to access or manipulate files outside of the intended directory. Local file inclusion is closely related, where an application allows files to be included and executed as part of the request without proper validation, leading to unauthorized access or execution of files on the server.
Here's how they work: a user-controlled input is used to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory. When the software does not properly neutralize special elements within the pathname, malicious input can cause the pathname to resolve to a location that is outside of the restricted directory.

When attackers exploit a Path Traversal vulnerability, they can manipulate file paths to gain unauthorized access to files and directories outside the intended web root. This can lead to exposure of sensitive files such as configuration files, user credentials, or even system files that could be used to further compromise the server. LFI vulnerabilities take this a step further by allowing attackers to include and execute files from the local server. If an attacker successfully exploits LFI, they could potentially execute arbitrary code, gain full control of the affected server, or escalate their privileges within the system. The impact of these vulnerabilities ranges from data breaches and unauthorized access to complete system compromise.

Recommended Security Controls

According to the OWASP and MITRE recommendations, to be protected against Path Traversal and Local File Inclusion, applications must:

1. Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does.
2. If user input cannot be avoided, ensure that the supplied value is valid, appropriate for the application, and is authorized for the user.

How Waratek's Protection Works

Waratek offers protection against Path Traversal and Local File Inclusion attacks via the path:traversal rule. This rule uses the tainting engine to track all user input, hooks into Java's File API and monitors file system operations. When a file system operation occurs, the Waratek agent checks if the file system path contains user-controllable (tainted) characters that traverse the filesystem.

The path:traversal:relative rule detects if user-controlled input is used to traverse the file system using relative file system sequences such as ".." that can resolve to a location that is outside of the current directory.

The path:traversal:absolute rule detects if user-controlled input is used to traverse the file system using absolute file system sequences such as "/path/to/file" that can resolve to a location that is outside of the current directory.

Both rules can be enabled at the same time to be protected against both relative and absolute Path Traversal attacks.

By default, when no taint source is specified in the rule, the Path Traversal rule protects against attacks coming from HTTP requests. Users have the option to also enable protection against path traversal attacks coming from other sources such as relational databases and/or deserialization-based protocols such as RMI.

Protective Action

When the Path Traversal rule is enabled in deny mode and a path traversal attack is identified then the malicious file system operation is terminated and a Java exception is thrown back to the application, in accordance with the File API.

The Path Traversal rule is applicable and can be safely enabled in all applications, apart from when applications depend on user-controlled file system paths that contain either relative or absolute file system sequences.

Best Practices

Waratek recommends not to enable the Path Traversal rule in blocking mode if the application depends on traversing the filesystem with user-controlled inputs. Instead, consider enabling the rule in allow mode to monitor such operations.

References

- https://owasp.org/www-community/attacks/Path_Traversal
- https://cwe.mitre.org/data/definitions/22.html

Cross Site Scripting (XSS)

Vulnerability Overview

Cross-Site Scripting (XSS) (CWE-79) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. This can lead to the theft of session cookies, defacement of websites, and redirection of users to malicious sites.

XSS vulnerabilities have been a significant concern in web security since the late 1990s. Despite advancements in security practices and tools, it remains one of the most common vulnerabilities in web applications. It consistently ranks in the OWASP Top Ten, largely due to the complexity of securing dynamic content and the diversity of attack vectors. Over the years, XSS has evolved from simple script injections to more sophisticated attacks targeting user sessions, data integrity, and even browser functionalities, highlighting the ongoing challenge of securing modern web applications.

The XSS flaw occurs when:

- Data enters a Web application through an untrusted source, most frequently a web request.
- The data is included in dynamic content that is sent to a web user without being validated for malicious content.

There are 2 primary types of XSS vulnerabilities:

1. Reflected XSS attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the HTTP request.
2. Stored XSS attacks are those where the injected script is permanently stored on the target servers, such as in a database.

Recommended Security Controls

According to the OWASP and MITRE recommendations, to be protected against XSS applications must:

- Understand the context in which the untrusted data will be used and the encoding that will be expected.
- Use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

How Waratek's Protection Works

Waratek offers protection against XSS attacks via the xss:html rule. This rule uses the tainting engine to track all user input, hooks into the web application's Servlet API and monitors all the write operations to the HTTP response. When a servlet write operation occurs, the Waratek agent uses a streaming tainted HTML 5.0 lexer and checks if any sequence of user-controllable (tainted) characters mutate the HTML syntax.

Using the default rule parameter (*), the XSS rule will:

- protect only against reflected XSS attacks (i.e. payloads coming from HTTP requests)
- protect the HTTP responses of all HTTP endpoints that produce HTML responses

To enable protection against stored XSS attacks then the source rule parameter must be configured with the value database. For example:

```
xss:html:*;source=database:deny:warn
```

To enable protection against both reflected and stored XSS attacks then the source rule parameter must be configured with the values httprequest, database. For example:

```
xss:html:*;source=httprequest,database:deny:warn
```

In most cases, defining the above XSS rule would provide the required level of protection.

Optionally, Waratek can also protect against attacks coming from deserialized data. For example, from protocols such as RMI, JMX and the XMLReader that are based on Java or XML deserialization. To enable protection against deserialized payloads add the deserialization taint source in the rule's taint source parameter. For example:

```
xss:html:*;source=httprequest,database,deserialization:deny:warn
```

To enable XSS security control, the XSS rule with the default parameter must be specified. In some rare cases, users might need to specify additional XSS rules. There are 2 main reasons for this:

- an application might produce HTTP responses whose output is HTML but the content-type is incorrectly set by the application. For example, the HTTP endpoint generates HTML but its content-type is XML.
- different HTTP endpoints might require different taint sources to be configured.

In such cases, users can define additional XSS rules and specify in each additional XSS rule the relative path of the HTTP endpoint for which XSS protection must be enforced and optionally the source of tainted data. For example:

```
xss:html:/pathOne;source=database,deserialization:deny:warn
```

```
xss:html:/pathTwo;source=httprequest,deserialization:allow:warn
```

Finally, it is important to note that by default the XSS rule is enabled in a non-strict lexing mode. This means that the XSS rule will allow certain user inputs to be injected (i.e. to mutate the HTML syntax). These certain inputs have been vetted as non-malicious and are only formatting. This allows common WYSIWYG editors and markup languages such as markdown to be used. This feature is also called Safe XSS Injection. To disable the safe injection feature and enable the strict lexing mode, use the safeinjectionenabled rule parameter. For example:

```
xss:html:*;safeinjectionenabled=false:deny:warn
```

Protective Action

When the XSS rule is enabled in deny mode and an XSS attack is identified then the malicious HTTP output operation is terminated and no further writing to the HTTP response is allowed.

The XSS rule is applicable and can be safely enabled for web applications that use the Servlet API to handle HTTP requests and responses. Only reflected XSS and stored XSS for HTML is currently supported. Protection against pure JavaScript or CSS (Cascading Style Sheets) payloads are not yet supported.

Best Practices

In most cases, defining the following XSS rule would provide the required level of protection:

```
xss:html:*;source=httprequest,database:deny:warn
```

References

- https://cwe.mitre.org/data/definitions/79.html
- https://owasp.org/www-community/attacks/xss/
- https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

System Hardening against common vulns such as XXE & SSRF

Vulnerability Overview

XML external entity injection (also known as XXE) (CWE-611) is a web security vulnerability that allows an attacker to interfere with an application's processing of XML data. It often allows an attacker to view files on the application server file system, and to interact with any backend or external systems that the application itself can access.

In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other backend infrastructure, by leveraging the XXE vulnerability to perform server-side request forgery (SSRF) attacks. SSRF is a web security vulnerability that allows an attacker to induce the server-side application to make HTTP requests to an arbitrary domain of the attacker's choosing.

Note that SSRF (CWE-918) and XXE (CWE-611) are closely related, because they both involve web-related technologies and can launch outbound requests to unexpected destinations.

Recommended Security Controls

According to the OWASP and MITRE recommendations, the safest way to prevent XXE is always to disable DTDs (External Entities) completely.

How Waratek's Protection Works

Currently, Waratek does not offer a dedicated security control that remediates XXE or SSRF. However, using other security features available by the Waratek Java Security platform it is possible to significantly reduce the impact of both XXE and SSRF. For example, using the File and Network rules it can be possible to harden the system and prohibit the vulnerable application to access unwanted resources. Please refer to the Waratek User Guide for more information about the File and Network rules.

Protective Action

When a filesystem or network resource is accessed that is not allowed by a File or a Network rule then the IO operation is terminated and an exception is thrown according to the operation's API. The File and Network rules can be enabled on any Java application.

Best Practices

To safely enable the File and Network rules in an environment, users must first understand the filesystem and network activity patterns of the application. Identify the resources that are required to be accessed by the application and then define File and Network rules to whitelist these resources accordingly.

References

- https://cwe.mitre.org/data/definitions/611.html
- https://cwe.mitre.org/data/definitions/918.html