WARATEK

WHITEPAPER

Security-as-Code: The Solution to Deserialization Vulnerabilities

This whitepaper provides background on the deserialization vulnerability, describes the limitations of the existing mitigation techniques & explains why Security-as-Code is ideal for solving this problem.

The Deserialization Problem

The problem when applications deserialize data from untrusted sources has been one of the most widespread security vulnerabilities over the last couple of years.

A Brief Background

Serialization converts a memory object into a stream of bytes to store it in 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 the machine's memory. Most programming languages provide facilities to perform native serialization and deserialization, and most of them are vulnerable.

Recent research by Gabriel Lawrence, Chris Frohoff, and Steve Breen demonstrated working deserialization attacks on popular Java applications and frameworks that allow Remote Command Execution.

To show their findings, they created the ysoserial tool, a proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization. The main driver for their research was finding a dangerous class in the Apache Commons Collection library. The name of that class is InvokerTransformer.

This finding gained much attraction mainly because of the popularity of the Apache Commons Collection (ACC) library. Even CERT issued a Vulnerability Note for the vulnerability in the ACC library. Any application, server, or framework that depended on the Apache Commons Collection was potentially vulnerable. JBoss, WebLogic, IBM WebSphere, and Jenkins were only just a few of the affected systems.

Explaining all the internal details of the deserialization exploit goes beyond the scope of this article, and many others do that very well. However, a few critical things need explaining.

What is the functionality of the InvokerTransformer, and why does it allow an attacker to exploit the system?

The InvokerTransformer's goal is to transform objects in a collection by invoking a method using reflection. Attackers abuse this functionality and manage to invoke any method they want. The deserialization PoC exploit tool ysoserial abuses InvokerTransformer, and instead of transforming a collection object, it invokes the `Runtime.exec()` that executes arbitrary commands on the target system.

To abuse the InvokerTransformer and make it invoke arbitrary, dangerous methods, such as the `Runtime.exec()`, a specially crafted method sequence needs to be created by the attacker. Each method in the sequence is called a "gadget," and the malicious sequence of method calls is called a "gadget chain." In the case of the Apache Commons Collections, the InvokerTransformer is a gadget in the malicious gadget chain. ysoserial expands every time it identifies a new gadget chain to include it in its available payloads.

Some robust gadget chains are often called golden gadget chains. Golden gadget chains contain no third-party gadgets - only JRE gadgets! They are just a vulnerable version of the JVM required for such chains to exploit the system.

The attack works in the following steps:

1. A vulnerable application accepts user-supplied serialized objects.
2. An attacker creates a malicious gadget chain, serializes it into a stream of bytes, and sends it to the application.
3. The vulnerable application reads the received stream of bytes and tries to construct the object. This operation is called "deserialization."
4. During deserialization, the gadget chain executes, and the system gets compromised.

What is the Impact of Such a System Compromise?

It depends on the creativity of the attacker. Depending on the payload, a gadget chain can perform exploits such as Remote Code Injection, Remote Command Execution, and Denial of Service. In other words, deserialization vulnerabilities are critical vulnerabilities with a CVSS score from 7.5 to 10, depending on the environment.

Where exactly is the vulnerability in this scenario, and what

WARATEK

makes the above scenario vulnerable to attacks?

There are two criteria for introducing a deserialization vulnerability to an application:

1. The application must accept and deserialize serialized data from a location to which an attacker has access.
2. Vulnerable classes must exist in the classpath of the application, meaning that it is not enough for an application to use a "vulnerable" version of the Apache Commons Collection to be vulnerable. It must also deserialize data from unsafe locations.

Moreover, this is precisely why the Apache foundation claims no blame on the InvokerTransformer and other such classes that implement a specific functionality for this vulnerability. It is the combination of both these criteria that introduces the vulnerability.

The InvokerTransformer by itself is *not* vulnerable. How did Apache react to the finding of this vulnerability? From the Apache side, even though they stated that the InvokerTransformer was not to blame for this vulnerability, they hardened the InvokerTransformer by removing its ability to be serialized.

However, such a change breaks backward compatibility, and any application dependent on serializing the InvokerTransformer would break.

To overcome this limitation, the Apache community decided to introduce a system property that will restore the previous, potentially unsafe behavior of the InvokerTransformer. Therefore, with Apache's fix, a system cannot both use the InvokerTransformer for deserialization and be protected at the same time. It must be either one or the other, based on a system property.

By not allowing the serialization of InvokerTransformer, attackers will not be able to use the InvokerTransformer anymore to craft malicious gadget chains.

Does this Solve the Problem at its Core?

A Greek expression says, "cutting off your head will not solve your problem if you have a headache," which is what happens with the InvokerTransformer. Disabling its serializability is not the proper way to solve the problem; it might break the application and not automatically make the system safe.

The InvokerTransformer is not the only known gadget. Researchers identified several other gadgets and will identify many more in the future. Disabling classes found useful as a gadget only creates a never-ending Whack-a-Mole game.

The ysoserial exploit kit is a good example that demonstrates this conundrum. Currently, it contains 27 gadget chains that utilize several distinct gadgets. Disabling the InvokerTransformer does not solve the problem since more than 21 other gadget chains do not use the InvokerTransformer and could potentially compromise systems.

To make things even worse, golden chains that contain only JRE gadgets cannot be blindly disabled or removed because, most probably, the application will break because of the missing required functionality.

Additionally, the transitive dependencies of third-party components create a library sprawl which makes the problem of identifying and disabling "dangerous" classes even more complicated.

Variations of Deserialization Attacks

At this point, it is crucial to introduce three variations of the deserialization attacks to understand their impact better. There are:

1. Blind deserialization attacks aim to extract data from the target system in environments where the system is behind a network firewall that blocks outgoing connections or when strict Security Manager policies are in place.
2. Asynchronous (or stored) deserialization attacks store the gadget chains in a database or a message queue. The gadget chains execute when the target system reads data from the database or the message queue and deserializes them.
3. Deferred-execution deserialization attacks do not execute the gadget chains during but after deserialization is completed. Deferred-execution usually leverages the finalize() method.

The situation gets even worse because all the known DoS deserialization attacks were classified as "will not fix" by Oracle [1] [2] or a few other vendors, such as Red Hat.

Having a production infrastructure with vulnerable software whose vendors refuse to provide a fix is the worst situation any enterprise wants!

What is the Proper Fix?

Is there a solution that solves the problem and stops all the various types of deserialization attacks?

According to CERT, "Developers need to re-architect their applications." Such a fix requires significant code changes, time, effort, and money. If changing the source code and the application's architecture is an option, this is the preferred approach.

However, remember that even if an application does not perform any deserialization in its components, most servers, frameworks, and third-party components do. So, it is challenging to be 100% certain that the whole stack does not and will never perform deserialization without breaking existing required functionality.

Enterprise solutions need protection fast and without requiring source code changes. Any solution requiring code changes and more than a few minutes of deployment time is not acceptable for production environments. Especially for enterprise production environments with hundreds of deployed instances, making any source code changes is almost not feasible to implement. The margin of tolerance decreases significantly for critical vulnerabilities like deserialization.

CERT suggests that blocking the network port using a firewall might sometimes solve the problem. However, in most cases, this is not applicable. For example, the deserialization exploits in JBoss, WebLogic, and WebSphere run on the HTTP port of the web server, which means that blocking that port will render the server useless. Therefore, blocking the network port is not a viable option.

How did the Vendors of the Affected Systems Solve the Issue?

Without going into much detail about every affected software, the following list shows how some other vendors handled the issue:

Vendor	Effort
Spring	Hardened the dangerous classes
Oracle WebLogic	Blacklist
Apache ActiveMQ	Blacklist
Apache BatchEE	Blacklist + Whitelist
Apache JCS	Blacklist + Whitelist
Apache OpenJPA	Blacklist + Whitelist
Apache OWB	Blacklist + Whitelist
Apache TomEE	Blacklist + Whitelist
Atlassian Bamboo	Disabled deserialization
Jenkins	Disabled deserialization + upgraded ACC
IBM WebSphere	Upgraded ACC

Source: Alvaro Muñoz and Christian Schneider

Also, note that there were cases where the vendors refused to create a fix for the issue either because they did not acknowledge the problem as their own or the affected system is an old version that is no longer supported.

How can Security Teams Protect Against Deserialization Attacks?

How can such production systems be protected if the vendors cannot provide patches and the customers cannot make source code changes?

First, there are the Web Application Firewalls. WAFs are not helpful because they have no application context since they can only examine the application's input and output. Applying heuristics to incoming requests is guaranteed to produce false positives and negatives. Any security solution with no application context and operating outside of the application cannot adequately solve the deserialization vulnerability.

Second, some RASP vendors and some Java agents either disable deserialization altogether or apply blacklisting / whitelisting on the classes.

These solutions break the Oracle Binary Code License Agreement and illegally deploy in production. Even if that was legal, completely disabling deserialization system-wide, this is guaranteed to break all except the most trivial Java applications. So, this is out of the question for enterprise environments.

Now let us examine here why blacklisting and whitelisting are inadequate solutions to the problem.

Any security solution that depends on the blacklisting of dangerous classes requires profiling of the application to verify that the application does not utilize these classes. Without first profiling the application, it is impossible to blacklist a class because the risk of breaking the application's functionality is significant.

Additionally, a negative security model means never being sure everything is blacklisted. The list of blocked signatures has to be maintained constantly and frequently, and by definition, it does not protect any unpublished, zero-day exploits.

Any security solution that promotes a blacklisting strategy as a solution to deserialization attacks is doomed to fail since it plays the Whack-a-Mole game.

Whitelisting is a much better approach than blacklisting. However, to apply to the whitelist, profiling of the application is again required. In this case, the whitelist will be an extensive list of classes.

Such extensive lists are complicated to manage, especially for enterprise environments. In addition, every time the application upgrades to a newer release, the profiling must be performed again, and a new whitelist needs to be created. Therefore, this considerably complicates the deployment of new releases in production.

This friction usually leads to whitelists that are not updated and, in turn, produces false positives. Finally, even if an enterprise accepts the effort to profile its infrastructure and maintain whitelists constantly, they are still vulnerable to golden gadget chains and Denial of Service deserialization attacks.

Another suggested mitigation is blindly blocking process forking and file/network IO.

Even though this approach will reduce the impact of a deserialization attack, it does not protect against blind attacks for data exfiltration or Denial of Service deserialization attacks.

Finally, some researchers suggest that an ad-hoc Security Manager can help mitigate these attacks. However, the truth is that even though it is a good first mitigation step, it is insufficient because of its many limitations.

1. Security Managers are known to be easily bypassed.
2. Security Managers do not protect deferred attacks where the execution of the payload is executed after deserialization, for example, via the finalize() method.
3. Security Managers will not mitigate any DoS deserialization attacks.
4. Security Managers require another type of whitelist to be created and maintained.

A new solution: Declarative Security-as-Code Deserialization Rules

Based on the above discussion, it is clear that there is a need for a better security solution to address this critical vulnerability. Here is a list of requirements to describe the ideal security solution for deserialization attacks:

1. Must work with zero source code changes
2. Must work with no application profiling
3. Must work with no configuration or tuning (no blacklists or whitelists)
4. Must allow applications to be updated/upgraded without redeployment effort
5. Must work with existing hardware and application stack
6. Must allow applications to use the "dangerous" classes/gadgets for legitimate functionality
7. Must not break existing application functionality or binary compatibility
8. Must not produce any false positives or false negatives
9. Must protect against all known gadget chains (all ysoserial payloads)
10. Must protect even against unpublished, zero-day gadget chains with no configuration
11. Must protect against golden gadget chains
12. Must protect against gadget chains that have been classified as "will not fix" by the vendors
13. Must protect against blind deserialization attacks
14. Must protect against Denial of Service deserialization attacks
15. Must protect both the Serializable and the Externalizable interfaces
16. Must protect against attacks via any end-point (such as HTTP, RMI, JMS, and JNDI)
17. Must protect the entire application stack (the JRE, the server, the framework, the application, and all the dependencies)
18. Must protect against deferred deserialization attacks (such as via the finalize() method)
19. Must protect against lateral / stored deserialization attacks (such as via databases)
20. Must not depend exclusively on the Security Manager
21. Must support all versions and releases of Java

Lastly, it is essential to note that all the above must complete without incurring any noticeable performance overhead and without fail. In other words, it must be production-ready.

The above list of requirements can benefit anyone who might want to evaluate the effectiveness and usability of a deserialization mitigation solution.

Waratek offers a new security feature that remediates Java object deserialization attacks and fulfills all the above requirements.

Using Waratek's Security-as-Code platform and turning on the declarative "Deserial" rule wholely and automatically protects the entire application stack against Java deserialization attacks, known or unknown (zero-day).

Waratek achieves this level of protection by creating a dynamic, restricted compartment inside its platform. This restricted compartment is active for the duration of each deserialization operation and afterward, such as during garbage collection. The restricted compartment allows any legitimate functionality to run normally but prohibits any gadget chain from abusing and compromising the system. The feature allows the InvokerTransformer to be used generally by systems that depend on this functionality without compromising the system by any malicious gadget chains.

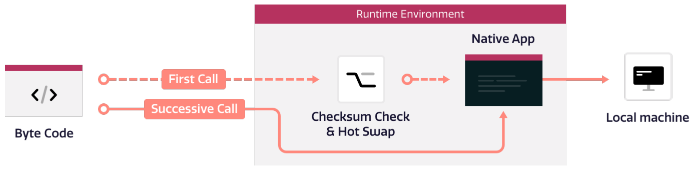
Waratek achieves all the above without making code changes, profiling, blacklisting, or whitelisting with no false positives or negatives and without breaking existing functionality.

The feature also remediates golden gadget chains (JRE-only gadgets), blind attacks, Denial of Service, asynchronous/lateral attacks, and attacks with deferred execution.

The protection is achieved immutably with a minimal performance overhead & can deploy without redeploying the application.

Author: Apostolos Giannakidis, Product Security, Microsoft

How it Works



When an action is performed on your applications for the first time, and an attempt is made to execute vulnerable code, Waratek Secure performs a checksum check and tells your application to ignore the code.

A healthy version of the code is returned in real-time as defined in your Policy Config file or the Waratek Portal. Only the healthy version will be made available on any additional call to that same piece of code, resulting in even faster execution.

Technical Specs

Feature	Notes
Agent Size	3MB
CPU Utilization	< 2%
Memory utilization	25MB
Network Utilization	Negligible at scale

See first hand how Waratek can help you

Immutable Security
Say goodbye to regressions after deployments. Once your policy is defined, no code added to the codebase can supersede the policy.

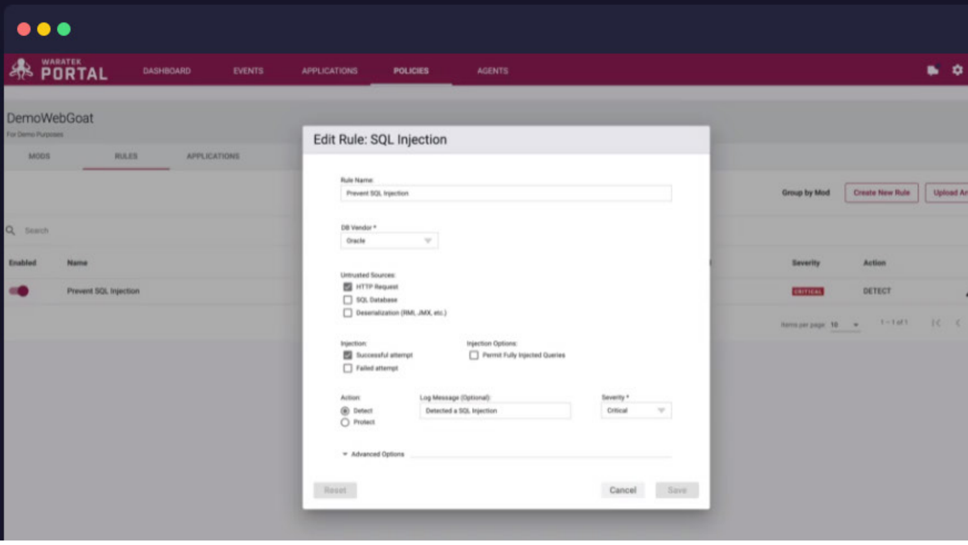
Deployment Agnostic
Securing your runtime instead of your codebase or CI/CD pipeline enables critical patches to be applied instantly instead of during the next deployment window.

Economically Scalable
Remove the toil of false positives and long feedback loops between security and engineering to transform the economics of AppSec.

Ready to Scale Security with modern Software Development?

Take a guided tour to learn how to accelerate your adoption of Security-as-Code to deliver AppSec at scale. No email required.

Start tour



Edit Rule: SQL Injection

Rule Name: Prevent SQL Injection

DB Vendor *: Oracle

Untrusted Sources:
- [✓] HTTP Request
- [ ] SQL Database
- [ ] Deserialization (RMI, JMX, etc.)

Injection:
- [✓] Successful attempt
- [ ] Failed attempt

Injection Options:
- [ ] Permit Fully Injected Queries

Action: [✓ Detect] Protect

Log Message (Optional): Detected a SQL Injection

Severity *: Critical

Advanced Options

Reset Cancel Save

WARATEK