The Deserialization Problem

waratek Highly accurate. Easy to install. Simple to operate.

Why container-based RASP is the most appropriate solution

What is the Deserialization vulnerability and what are the challenges in providing a solution.

The problem that occurs when applications deserialize data from untrusted sources is one of the most widespread security vulnerabilities to occur over the last couple years.

This article will provide background on the deserialization vulnerability, describe the limitations of the existing mitigation techniques and explain why Waratek's RASP by Virtualization technology is ideal to solve this problem.

A brief background

Serialization is the process of converting a memory object 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 the memory of the machine. All main 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 demonstrate their findings they have 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 the finding of a dangerous class in the Apache Commons Collection library. The name of that class is InvokerTransformer. This finding gained a lot of 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 there are many other articles that do that very well. However, a few important things need to be explained.

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.

In order 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. Note that every time a new gadget chain is identified, the ysoserial tool gets expanded to include it in its available payloads.

However, note that there are some gadgets that contain no third-party gadgets! These gadget chains contain only JRE gadgets. Nothing more than just a vulnerable version of the JVM is required for such chains to exploit the system. Such gadget chains are often called golden gadget chains and they are powerful.

How the attack works can be summarized 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 is executed and the system gets compromised.

What is the impact of such a system compromise?

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

Where exactly is the vulnerability in this scenario and what is it that makes the above scenario vulnerable to attacks?

Only two criteria are required in order for a deserialization vulnerability to be introduced to an application:

1. The application must accept and deserialize serialized data from a location where an attacker has access to.

2. Vulnerable classes must exist in the classpath of the application.

This means that it is not enough for an application to use a "vulnerable" version of the Apache Commons Collection in order to be vulnerable. It must also deserialize data from unsafe locations.

And this is exactly why the Apache foundation claims that the InvokerTransformer and other such classes that implement a certain functionality cannot be blamed 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 cannot be blamed for this vulnerability, they hardened the InvokerTransformer by removing its ability to be serialized.

However, this means that such a change breaks backwards compatibility and any application that was depended 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.

What needs to be understood here is that by not allowing the InvokerTransformer to be serialized, then attackers will not be able to use the InvokerTransformer anymore to craft malicious gadget chains.

Does this *really* solve the problem at its core? No. It merely patches the problem.

There is a Greek expression that says that if you have a headache, cutting off your head will not solve your problem. This is exactly what happened here with the InvokerTransformer. Disabling its serializability is not the proper way to solve the problem; it might break your application and does not automatically make the system safe.

The InvokerTransformer is not the only known gadget. Several other have been identified and many more will be found in the future. Disabling a class every time it is found that it can be used as a gadget will only create a never-ending Whack-a-Mole game.

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

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 important to introduce three variations of the deserialization attacks in order to better understand the impact of these attacks. There are:

1. Blind deserialization attacks that 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 that store the gadget chains in a database or a message queue. The gadget chains will be executed when the target system reads data from the database or the message queue and deserializes them.

3. Deferred-execution deserialization attacks that do not execute the gadget chains during deserialization but after deserialization has completed. This is usually achieved via the finalize() method. Here is an example of this attack.

The situation gets even worse because all the known DoS deserialization attacks were classified as “won’t 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”. Obviously, such a fix requires significant code changes, time, effort and money to achieve this. If changing the source code and the architecture of the application is an option then this is the preferred approach.

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

Especially for enterprise production environments with hundreds of deployed instances making any source code changes is almost not feasible to implement. Typically, for enterprise production environments, any solution that requires code changes and more than

a few minutes of deployment time is not acceptable, especially for critical vulnerabilities such as the deserialization vulnerability. Enterprise solutions need protection fast and without requiring source code changes.

CERT alternatively suggests that blocking the network port using a firewall might solve the problem in some cases. However, in most cases this is not applicable. For example, the deserialization exploits in JBoss, WebLogic, WebSphere, etc 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 of every affected software, the following list shows how some other vendors handled the issue:

Spring	Hardened the dangerous classes
Oracle WebLogic	Blacklist
Apache ActiveMQ	Whitelist
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

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

How can customers with old or legacy versions of affected systems be protected against the deserialization attacks?

If the vendors cannot provide patches and the customers cannot make any source code changes, then how can such production systems be protected?

First, there are the Web Application Firewalls. WAFs are not helpful here because they have no application context since they can only examine the input and the output of the application. Applying heuristics on the incoming requests is guaranteed to produce false positives and false negatives.

Any security solution that has no application context and operates outside of the application cannot adequately solve the deserialization vulnerability.

Second, there are some RASP vendors and some Java agents that either disable deserialization completely or apply blacklisting / whitelisting on the classes that are getting deserialized.

Some of these solutions break the Oracle Binary Code License Agreement and therefore are illegally deployed in production. Even if that was legal, by 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's examine here why blacklisting and whitelisting are bad solutions to the problem.

Any security solution that depends on blacklisting of dangerous classes requires profiling of the application in order to verify that these classes are not utilized by the application. Without first profiling the application, it is not possible to blacklist a class because the risk of breaking the functionality of the application is significant.

Additionally, adopting a negative security model means that you will never be sure that you have blacklisted everything. 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 whitelisting, profiling of the application is again required. In this case, the white list will be a really big list of classes.

Such big lists are very difficult to manage, especially for enterprise environments. In addition, every time the application needs to upgrade to a newer release, the profiling needs to be performed again and a new white list needs to be created. Therefore, this considerably complicates the deployment of new releases in production.

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

Another suggested mitigation is to blindly block (or whitelist) 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 nor Denial of Service deserialization attacks.

Finally, some researchers suggest that using 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.

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

A new solution: Virtual Container-based Deserialization Rule

Based on the above discussion, it's clear that there is a need for a better security solution to address this critical vulnerability. Here is a list of requirements for what could be described as 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 allow applications to use the "dangerous" classes / gadgets, as long as they are used for legitimate functionality
6. Must not break existing application functionality or binary compatibility
7. Must not produce any false positives or false negatives
8. Must protect against all known gadget chains (all ysoserial payloads)
9. Must protect even against unpublished, zero-day gadget chains with no configuration
10. Must protect against golden gadget chains
11. Must protect against gadget chains that have been classified as "won't fix" by the vendors
12. Must protect against blind deserialization attacks
13. Must protect against Denial of Service deserialization attacks
14. Must protect both the Serializable and the Externalizable interfaces
15. Must protect attacks via any end-point (such as HTTP, RMI, JMS, JNDI, etc)
16. Must protect the full application stack (the JRE, the server, the framework, the application and all the dependent libraries)
17. Must protect against deferred deserialization attacks (such as via the finalize())
18. Must protect against lateral / stored deserialization attacks (such as via databases)
19. Must not depend exclusively on the Security Manager
20. Must support all versions and releases of Java

Lastly, it's important to note that all the above must be achieved without incurring any noticeable performance overhead. In other words, it must be production-ready.

The above list of requirements can be very helpful to anyone who might want to evaluate the effectiveness and the 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 the Waratek RASP container and by turning on the “Deserial” rule, the full application stack is automatically protected against Java deserialization attacks both known or unknown (zero-day).

This is achieved by creating a dynamic, restricted compartment, inside the Waratek RASP container. This restricted compartment is active for the duration of each deserialization operations as well as after the deserialization has completed on specific events such as during garbage collection. The restricted compartment allows any legitimate functionality to run normally but prohibits any gadget chain to abuse and compromise the system. The feature even allows the InvokerTransformer to be used normally by systems that depend on this functionality, without risking the system to be compromised by any malicious gadget chains.

All the above is achieved without having to make any code changes, any profiling, any black or white listing 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, as well as attacks with deferred-execution.

The protection is achieved with minimum performance overhead and can be deployed on any Java release.

Author: Apostolos Giannakidis, Lead Security Engineer, December 2016