Insecure deserialization

What is insecure deserialization?

Insecure deserialization is a vulnerability that occurs in untrusted data that is used to abuse the logic of an application, perform a DOS attack, repay attacks, injections attacks, privilege escalation attacks or even execute arbitrary code when it has been deserialized. It's even feasible to swap out a serialized object with one of a completely different class. Objects of any class that is available to the website, regardless of which class was anticipated, will be deserialized and created, which is alarming. As a result, unsafe deserialization is frequently referred to as a "object injection" flaw.

An exception might be thrown if an object of an unusual class is used. However, the harm may have have been done by this point. Many deserialization-based assaults are accomplished before the deserialization process is complete. This implies that even if the website's own functionality does not interact directly with the malicious item, the deserialization process itself might start an attack. As a result, websites that use highly typed languages for their logic may be vulnerable to these approaches.

Insecure deserialization allows an attacker to remotely execute its code within a vulnerable application/software. Although getting to successfully exploit this code isn’t for your average hacker, if it is successful, an attacker can escalate an Insecure deserialization attack by changing the privileges on the currently run server that is running the possibly vulnerable web application. When this has been done the perpetrator can freely go through-out the internal network and commence even more attacks.

In the past few years, the most vulnerable applications that would succumb to these attacks would be java applications, as they’ve stolen the spotlight when it comes to these types of attacks although java applications are not the only targets when it comes to insecure deserialization, these attacks can also happen in applications such as PHP and Python based web applications.

Although insecure deserialization isn’t as big a problem as it used to be back in such as the likes of 2017 when it was at its peak in popularity reaching #7 and steadily staying at #8 in OWAPS top 10 web application security risks it’s still something that shouldn’t be underestimated and taken very seriously.

Why is insecure deserialization so common?

The most common cause of insecure deserialization is a widespread lack of knowledge of how risky deserializing user-controllable data may be. In an ideal world, user input should never be deserialized.

However, most website owners may believe they are safe because they apply some type of extra check on the deserialized data. Because it is nearly hard to design validation or sanitization to account for every possibility, this method is frequently useless. These checks are also faulty since they rely on inspecting data after it has been deserialized, which is sometimes too late to prevent an attack.

Deserialized objects are frequently expected to be trustworthy, which might lead to vulnerabilities. Developers may believe that consumers can’t read or handle data successfully, especially when employing languages that use a binary serialization structure. While it may take more work, an attacker can exploit binary serialized items in the same way that they can exploit string-based formats.

Because of the large amount of dependencies seen in current websites, deserialization-based attacks are also viable. A normal website could use a number of distinct libraries, each with its own set of dependencies. This results in a large number of classes and methods that are difficult to protect.

It’s difficult to anticipate which methods will be executed on the bad data since an attacker can build instances of any of these types. This is especially true if an attacker can chain together a long sequence of unexpected method invocations, delivering data to a sink that has nothing to do with the original source. As a result, predicting the flow of harmful data and plugging every potential weakness is nearly difficult.

What is the impact of insecure deserialization?

Because it gives an access point to a vastly enlarged attack surface, unsecured deserialization can have a significant impact. It allows an attacker to repurpose existing application code for malicious purposes, resulting in a slew of additional flaws, including remote code execution.

Insecure deserialization can lead to privilege escalation, uncontrolled file access, and denial-of-service attacks even if remote code execution is not feasible.

How to prevent insecure deserialization?

Deserialization of user input should generally be avoided unless absolutely essential. In many circumstances, the high severity of exploits it may enable, as well as the difficulties of guarding against them, exceed the benefits.

If you must deserialize data from an untrustworthy source, take precautions to ensure that the data has not been tampered with. You might, for example, use a digital signature to ensure the data's integrity. However, such tests must be completed before the deserialization procedure may commence. They aren't very useful otherwise.

You should avoid utilizing generic deserialization features if at all feasible. All characteristics of the original object are included in serialized data from these methods, including private fields that may contain sensitive information. Instead, you could write your own serialization methods for each class, allowing you to at least select which fields are accessible.

Finally, keep in mind that the vulnerability is caused by the deserialization of user input, not by the presence of gadget chains that handle the data afterwards. Don't count on attempting to get rid of device chains you find during testing. Due to the web of cross-library dependencies that very probably exist on your website, it is impracticable to try to plug them all in.

Publicly disclosed memory corruption attacks are also a consideration at any given time, which means that your program might be susceptible regardless.

For unsafe deserialization testing, you may manually verify your application from the source code site and in the running state.

A static code analysis tool can also be used. It would, however, require access to your code and would only see the "theoretical" perspective of your application in a non-executed state.

The ideal way is to utilize a dynamic application security testing tool to test your executed, live program. It may scan your web application or API on its own.

How was insecure deserialization discovered?

“The Insecure Deserialization Vulnerability was first reported on January 12th, 2019 by researchers at Tencent Security Xuanwu Lab. This vulnerability allows attackers to execute arbitrary code on the target server when they are able to inject malicious data into the JSON string being passed to the deserialize method.” –

[https://crashtest-security.com/insecure-deserialization/#:~:text=The%20Insecure%20Deserialization%20Vulnerability%20was,passed%20to%20the%20deserialize%20method.](https://crashtest-security.com/insecure-deserialization/#:~:text=The%20Insecure%20Deserialization%20Vulnerability%20was,passed%20to%20the%20deserialize%20method)

An example of insecure deserialization:

The Jenkins cryptominer,

The Jenkins cryptominer was probably the largest ever illegal cryptomining operation that was ever discovered. The crypto-miner operator exploits the Jenkins Java deserialization implementation's reported CVE-2017-1000353 vulnerability by submitting two successive queries to the CLI interface. The vulnerability arises from the serialized object's lack of validation, therefore, enables whatever serialized object to be accepted.

And it wasn’t just Jenkins’s servers but Tesla too, the hackers got access to Amazon’s web service account owned by tesla and exposed sensitive data involving Tesla’s telemetry, mapping and vehicle service data and then took advantage of the open server and started running their cryptocurrency mining malware.

This attack happened due to misconfigured and unsecured servers and was found by a security firm called RedLock, specifically They directed users to an insecure Kubernetes interface, an open-source program for managing cloud-based resources and tools.

"The hackers had penetrated Tesla's Kubernetes console, which was not password secured," RedLock continued. Access credentials to Tesla's AWS environment were exposed within one Kubernetes pod, which held an Amazon S3 (Amazon Simple Storage Service) bucket containing sensitive data such as telemetry." Which was a huge blow to Tesla for their incompetence.

The hackers then gained access to Tesla's AWS server and set up a cryptojacking operation based on the Stratum bitcoin mining technology. There's no word on how long the operation lasted or how much money was extracted. According to the security assessment, the hackers employed a variety of techniques to avoid discovery and continue their activities invisibly, including disguising their malware behind a CloudFlare-hosted IP address and reducing the CPU resources necessary to mine.

RedLock researchers submitted the issue to Tesla's bug bounty program as soon as they discovered it and got a reward of almost US$3,000. Tesla spokespeople responded to the revelation by saying that the vulnerability was fixed within hours of the report being made public, and that the information stolen exclusively related to internally utilized engineering test cars. Apparently, their initial investigation found no evidence of a breach of customer privacy or vehicle security not that they would tell us anyway as that would affect their company and trust within its customers.

Organizations must take the necessary safeguards as more cryptojackers attack servers. To protect its data and resources, a corporation must build a fortified environment. The following are some guidelines to follow:

Implement stricter server credential management and keep complete visibility over third-party programs that use these credentials.

Make use of all privacy options and set up servers securely, especially those that store sensitive information.

To meet the demands of both the company and their consumers, implement access controls and guarantee effective encryption.

Use and follow suitable patching procedures.

References

**OWASP**

[OWASP Cheat Sheet: Deserialization](https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html) [OWASP Proactive Controls: Validate All Inputs](https://owasp.org/www-project-proactive-controls/v3/en/c5-validate-inputs) [OWASP Application Security Verification Standard](https://owasp.org/www-project-application-security-verification-standard) [OWASP AppSecEU 2016: Surviving the Java Deserialization Apocalypse](https://speakerdeck.com/pwntester/surviving-the-java-deserialization-apocalypse) [OWASP AppSecUSA 2017: Friday the 13th JSON Attacks](https://speakerdeck.com/pwntester/friday-the-13th-json-attacks)

<https://www.acunetix.com/blog/articles/owasp-top-10-2017/>

<https://www.acunetix.com/blog/articles/what-is-insecure-deserialization/>

https://www.waratek.com/wp-content/uploads/2019/06/WP-Deserialization-20190610.pdf

**External**  
**\*** [CWE-502: Deserialization of Untrusted Data](https://cwe.mitre.org/data/definitions/502.html)**\*** [Java Unmarshaller Security](https://github.com/mbechler/marshalsec)**\*** [OWASP AppSec Cali 2015: Marshalling Pickles](https://frohoff.github.io/appseccali-marshalling-pickles/)

* https://crashtest-security.com/insecure-deserialization/#:~:text=The%20Insecure%20Deserialization%20Vulnerability%20was,passed%20to%20the%20deserialize%20method.