Runtime Prevention of Deserialization Attacks

François Gauthier
francois.gauthier@oracle.com
Oracle Labs
Brisbane, Queensland, Australia

Sora Bae
sora.bae@oracle.com
Oracle Labs
Brisbane, Queensland, Australia

ABSTRACT

Untrusted deserialization exploits, where a serialised object graph is used to achieve denial-of-service or arbitrary code execution, have become so prominent that they were introduced in the 2017 OWASP Top 10. In this paper, we present a novel and lightweight approach for runtime prevention of deserialization attacks using Markov chains. The intuition behind our work is that the *features* and *ordering* of classes in malicious object graphs make them distinguishable from benign ones. Preliminary results indeed show that our approach achieves an F1-score of 0.94 on a dataset of 264 serialised payloads, collected from an industrial Java EE application server and a repository of deserialization exploits.

KEYWORDS

Deserialization, Markov chains, Runtime protection

ACM Reference Format:
François Gauthier and Sora Bae. 2022. Runtime Prevention of Deserialization Attacks. In *New Ideas and Emerging Results (ICSE-NIER'22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510455.3512786

1 INTRODUCTION

In programming languages, serialization is the process of converting an in-memory object or data structure into a persistent format; deserialization works the opposite way. An attacker accessing the serialized form of an object can thus influence the object that will be created upon deserialization. In recent years, security researchers discovered various ways of abusing deserialization to achieve denial-of-service or arbitrary code execution in various languages like Java, [1] C#, PHP, Python, and Ruby using various serialization formats like binary, XML, JSON, and YAML [3, 8, 11, 12]. Deserialization issues are now so prominent that they are now included in the OWASP Top 10 Web Application Security Risks list [2]. In this paper, we focus on detecting attacks against native Java deserialization that uses byte streams as the serialization format. To help combat the threat posed by native Java deserialization vulnerabilities, deserialization filters were introduced in Java 9 and back-ported to Java 6, 7, and 8 [1]. Upon deserialization of, the filter is invoked after resolving the class from the byte stream and before creating an object of that class in memory, giving the filter an opportunity to inspect the class and stop the deserialization process altogether if a forbidden class is detected. However, the onus of developing the filters is on developers and the manual effort involved is not trivial. In this paper, we propose an approach to automatically build probabilistic models from *benign* and *malicious* byte streams to detect malicious deserialization at runtime.

[1]Java and Java EE are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

ICSE-NIER'22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9224-2/22/05…$15.00
https://doi.org/10.1145/3510455.3512786

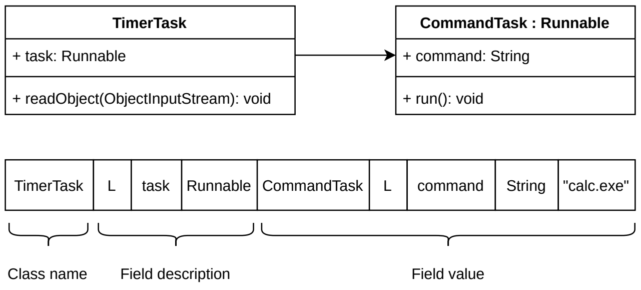

Figure 1: A simple class diagram and its corresponding byte stream

2 BACKGROUND ON JAVA DESERIALIZATION

In Java, any object from a class that directly or indirectly (i.e. through inheritance) implements the `Serializable` interface can be serialized and deserialized using Java native serialization. During the serialization process, starting from the root object, references to other objects (e.g. through class fields) are resolved and serialized deterministically in a recursive manner until the entire object graph has been converted to a byte stream. During deserialization, the serialized object graph is read sequentially from the byte stream, one object at the time. When an object is deserialized, the first information that is extracted from the stream is its class name, at which point deserialization filters are invoked to give the application an opportunity to introspect the class and interrupt deserialization if desired. Figure 1 shows an example of a class diagram (top) with a simplified representation of its corresponding byte stream (bottom). Assuming that instances of those classes have been constructed, serializing the corresponding object graph would result in the byte stream at the bottom of Figure 1. The first part of the byte stream contains the name of the class of the first object to be deserialized, followed by field descriptions. The last part of the stream contains the field values, which can be objects themselves. If this stream was deserialized, the deserialization filter would receive the `TimerTask` class first, followed by the `CommandTask` class. Detailing the exact mechanisms that an attacker can use to exploit Java deserialization is beyond the scope of this paper, but the interested reader can refer to [8, 9, 11] for more information. The key takeaway is that object graphs are serialized, deserialized, and filtered sequentially; a property we leverage to abstract them as Markov chains.

3 BACKGROUND ON MARKOV CHAINS

A Markov chain represents a system that has a finite number of states: $S = \{s_1, s_2, \dots, s_n\}$ and that transitions between states with some probability p at each step t. The probability of the system starting in a state $s_i \in S$ is captured by its initial state probability vector: $p_{init} = (p_1, p_2, \dots, p_n)$, where each probability p_i corresponds to the probability of the chain starting in state s_i and where the probabilities in p_{init} sum to one. In Markov chains, the probability of transitioning from a state s_i to another state s_j depends on s_i only, and is captured by a transition probability matrix, where rows correspond to the state at step t, columns correspond to the state at step $t+1$, and each row sums to one:

$$p_{tr} = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{pmatrix}$$

Given a Markov chain and a sequence of states $(x_1, x_2, \dots, x_n)$, one can calculate the probability that the chain *generated* the sequence with a simple product of probabilities:

$$P((x_1, x_2, \dots, x_n)) = p_{init}(x_1) \cdot \prod_{i=2}^{n} p_{tr}(x_{i-1} x_i) \qquad (1)$$

4 MODELLING JAVA DESERIALIZATION WITH MARKOV CHAINS

In this section, we explain how we model deserialization as a Markov chain. Our choice of Markov chains over other learning approaches was motivated by two main factors: 1) previous failed experiments with a sequence-agnostic classifier, and 2) the need to make predictions based on small datasets. We indeed experimented with naïve Bayes classifiers first and only achieved marginal improvement over random classification. Then, we chose Markov chains (MC) over more complex sequence-based learning approaches that require large datasets like RNN, and LSTM because in our setup, deserialization is a relatively uncommon operation, leading to a small dataset. In section 6, we show how MC generated from Bayesian inference, which accounts for the inherent uncertainty of small datasets, lead to significantly more precise predictions than MC that are derived directly from empirical data.

4.1 Abstracting Java classes as states in a Markov chain

Classes in a stream and their various features determine the code that is executed during deserialization. For example, in Figure 1, because the `CommandTask` class implements the `Runnable` interface, the `CommandTask` constructor will be invoked to create a `Runnable` object for the `task` field. Also, because the `TimerTask` class overrides the `readObject` method, it can alter default deserialization, which is a feature that is often exploited in deserialization attacks.

To exploit a deserialization vulnerability, attackers will seek to *craft* a byte stream consisting of specific classes with specific features in a specific order that will typically lead to denial-of-service or arbitrary code execution. The key insight here is that the class's intended use is largely irrelevant to attackers, who are instead solely interested in the very specific features that will lead to successful exploitation. For this reason, we studied all the deserialization exploits in `ysoserial` [7] and manually identified some of the features that make them *useful* for exploitation purposes. The non-exhaustive list of features we identified are listed in Table 1.

Given the set of features in Table 1, we can abstract each class as a Boolean feature vector. Given n Boolean features, the number of possible feature vectors is finite and equal to 2^n. In our setup, the set of states in the Markov chain is thus the set of possible feature vectors. Because the maximum number of states grows exponentially with the number of features, in practice, we use the set of feature vectors that are observed in the training set, which cardinality tends to be much smaller than 2^n. To account for new feature vectors in the testing set, we create a generic state to which all unobserved states map to.

4.2 Estimating probabilities from data

We estimate the various probabilities in our Markov chain directly from dynamic observations. Specifically, given a set of byte streams, we deserialize them, extract the resulting sequences of classes, and abstract all classes to feature vectors. This results in a set of concrete Markov chain instances (i.e. sequences of states) from which we can estimate the initial and state transition probabilities. The most straightforward approach is to directly use empirically observed frequencies as probabilities. This works well in contexts where the sample size is large. When the sample size is small, however, statistical inference methods are generally preferable. In this work, we use Bayesian inference to estimate the probabilities of a Markov chain where empirical observations are used to guide the inference process. Bayesian inference models the variables to infer as random variables issued from specific probability distributions. Then, through a guided random process (e.g. Markov Chain Monte Carlo), it infers the parameters of those probability distributions that maximise the likelihood of the empirically observed values. Consider, for example, the transition probability matrix of a Markov chain. Our goal is to estimate the transition probabilities that best explain the observed sequences of states. A typical way of modelling such a matrix is to represent each row as the outcome of a Dirichlet distribution, which is parameterised with a vector of concentration parameters $(\alpha_1, \dots, \alpha_K)$ where $\alpha_i > 0$ and produces as output a vector of K real numbers that sum to one:

$$(x_1, \dots, x_K), \text{ where } x_i \in [0,1], \text{and } \sum_{i=1}^{K} x_i = 1$$

The concentration vector is used to initialise the distribution and captures our *prior* knowledge about transition probabilities. In our setup, all the concentration parameters are set to one, to represent that we have no prior knowledge. By repeatedly adjusting the concentration parameters, sampling from the per-row Dirichlet distributions, and evaluating the likelihood of the resulting matrix against our observations, Bayesian inference eventually converges to a set of *likely* transition matrices. It is important to note that through its inference process, Bayesian inference actually generates multiple distributions for each row, where the more recent ones are

Table 1: Common class features used in deserialization exploits

Id	Feature	Description
1	Uses reflection	True if the class calls any of the following from `java.lang.reflect`: - `Constructor.newInstance()` - `Field.set()` - `Method.invoke()`
2	Overrides readObject	True if the class overrides the method `Object readObject(ObjectInputStream ois)`
3	Overrides hashCode	True if the class overrides the `int hashCode() method.`
4	Has generic field	True if the class has a field of any of the following type: - `java.lang.Object` - `java.lang.Comparable` - `java.util.Comparator`
5	Implements Map	True if the class implements the `java.util.Map` interface.
6	Implements Comparator	True if the class implements the `java.util.Comparator` interface.
7	Calls hashCode	True if the class calls any of the following methods: - `int java.util.Objects.hash(Object... values)` - `int java.util.Objects.hashCode(Object o)` - `*.hashCode()`
8	Calls compare	True if the class calls any of the following methods: - `*.compare()` - `*.compareTo(...)`

expected to more precisely capture the *real* probabilities. In cases where the observations are few, the inference might not converge to a single solution, but rather to a set of plausible solutions. We later use metrics like standard deviation over the set of generated solutions to estimate the *confidence* in our predictions.

5 RUNTIME PREVENTION OF DESERIALIZATION ATTACKS

We now present our approach to infer Markov chains from benign and malicious deserialization examples and predict if a given byte stream is malicious. In our setup, the benign and malicious examples come from the application under test and from the `ysoserial` dataset [7, 11] respectively. To collect the classes and their associated features, we implemented a custom deserialization filter that uses ASM [4] to dynamically extract features from deserialised classes. To enable the classification of byte streams as benign or malicious, we create two Markov chains during the inference phase: one from benign examples, and one from malicious examples. Once the inference phase is complete, we use another custom deserialization filter to detect and prevent deserialization attacks based on the inferred Markov chains. A simplified filter is illustrated in algorithm 1. It takes as input the benign ($\mathcal{B}$) and malicious ($\mathcal{M}$) Markov chains as well as two threshold parameters t and l. Once deserialization starts, the Java runtime invokes the filter every time a new class is read from the serialized stream and passes it the class and a Boolean flag indicating whether the end of the stream has been reached. In practice, we must derive the *end* flag from other filter inputs, but we omit these details for clarity. The filter then uses ASM [4] to abstract the class as a feature vector and appends it to the current sequence of classes (lines 3-4). It then computes the mean probability that the sequence has been *generated* by the benign or malicious Markov chains (lines 5-6). Then, it computes confidence intervals of t standard deviations around the means and checks if the intervals are disjoint (line 7). If the end of the stream has been reached and the intervals are disjoint, the highest mean probability determines the outcome (line 9). If the end has been reached but the intervals are not disjoint, we do not have enough confidence in the results to reach a decision and conservatively reject the stream (line 11). If the intervals are disjoint, and at least l classes have been read from the stream, deserialization is aborted early if the stream is malicious (line 13). Otherwise, the filter postpones the decision and lets deserialization proceed to the next class in the stream by returning `undecided`.

6 PRELIMINARY RESULTS

To validate our approach, we conducted an experiment on the Oracle WebLogic Server[2] [3] (WLS). In our setup, we collected 227 benign deserialization chains (avg. length of 38.96) from trusted runs of WLS. We also collected 37 malicious chains (avg. length of 16.68) from the deserialization payloads available in [7]. To measure the precision, recall and F1-score of our approach, we conduct a 5-fold cross-validation experiment where 80% of the examples are used for inference and the remaining 20% are used for prediction. Furthermore, to assess the benefits of using statistical inference to

[2]https://www.oracle.com/au/middleware/technologies/fusionmiddleware-downloads.html

[3]Oracle®WebLogic Server is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Algorithm 1: Deserialization Attack Prevention

```
    Input: $\mathcal{B}$, $\mathcal{M}$, $t$, $l$
    Output: status ∈ {accepted, rejected, undecided}
1   $seq$ ← new List()
2   Function MarkovFilter(class, end):
3       $features$ ← ExtractFeatures($class$)
4       $seq$.append($features$)
5       $\overline{P_{\mathcal{B}}} \leftarrow mean(P(seq \mid \mathcal{B}))$
6       $\overline{P_{\mathcal{M}}} \leftarrow mean(P(seq \mid \mathcal{M}))$
7       $disjoint \leftarrow ((\overline{P_{\mathcal{B}}} \pm t\sigma) \cap (\overline{P_{\mathcal{M}}} \pm t\sigma) = \emptyset)$
8       if $end$ and $disjoint$ then
9           return $\overline{P_{\mathcal{M}}} > \overline{P_{\mathcal{B}}}$ ? rejected : accepted
10      else if $end$ and $\neg disjoint$ then
11          return rejected
12      else if $disjoint$ and $|seq| \geq l$ and $\overline{P_{\mathcal{M}}} > \overline{P_{\mathcal{B}}}$ then
13          return rejected
14      else
15          return undecided
16      end
17  end
```

estimate probabilities, we also conduct the same experiment using Markov chains inferred directly from empirical data. We use PyMC3 for Bayesian inference [16] and Pomegranate for empirical inference [17]. Table 2 shows the results of our experiment with $t \in [0, 3]$ and $l = \infty$, drawing 5 000 samples from a Metropolis-Hastings sampler and using the last 500 samples for prediction. Bayesian inference performs significantly better, at the expense of inference times that are orders of magnitude larger. Note, however, that inference can be performed offline and that the actual runtime overhead, in the order of milliseconds, is similar for both approaches (i.e. extracting class features and resolving Equation 1). In algorithm 1, we let the filter stop the deserialization of malicious streams early if at least l classes have been read and the end of the stream has not been reached. Figure 2 shows the precision and recall achieved with different values of l, and $t = 2$. Reading more classes is beneficial to precision while recall remains largely unaffected. To our knowledge, this is the first study to use class features and ordering for *defensive* purposes against deserialization attacks. Considering our previous failed experiment with order-agnostic classifiers and our current F1-score of 0.94 using MC, our results suggest that class features and ordering do capture the *esssence* of malicious chains.

Table 2: Precision, recall, F1-score, and inference time of Bayesian and empirical Markov chains

	t	Precision	Recall	F1-score	Time (sec)
Bayesian	0	91.67±6.97	96.67±6.67	0.94±0.03	7163
	1	91.67±6.97	96.67±6.67	0.94±0.03	
	2	89.72±8.94	100.0±0.00	0.94±0.05	
	3	88.17±11.26	100.0±0.00	0.93±0.07	
Empirical	—	72.95±14.27	100.0±0.00	0.84±0.09	0.7

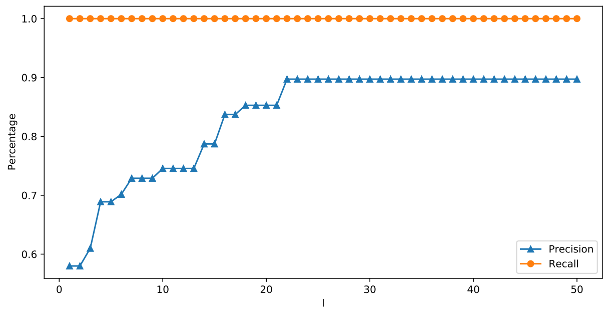

Figure 2: Average precision and recall in function of l

7 RELATED WORK

In the security community, the classes used in a deserialization payload are referred to as "gadgets" and the resulting byte stream is known as a "gadget chain". Many existing work tackled the problem of detecting gadgets chains in application and libraries [6, 10, 14, 15, 18] using techniques like debugger-assisted manual analysis, static analysis, and hybrid (i.e. static and dynamic) analysis.

More closely related to our work are approaches aimed at detecting and preventing deserialization attacks [5, 13]. Cristalli et al. [5], present a two-phase approach that learns trusted execution paths and sandboxes the native Java deserialization mechanism to allow deserialization from those paths only. This approach does not generalise to previously unseen paths and the precision thus depends on the exhaustiveness of the training phase. To implement their system, the authors modified a Java Virtual Machine (JVM), and report overheads in the order of 20%-40%. Pan et al. [13], use heavyweight instrumentation (e.g. 100x slowdown) to dynamically collect execution traces and train deep learning models to detect malicious deserialization at runtime. To achieve an F1-score > 0.90, authors had to manually generate over 8 000 execution traces, which is highly unpractical. In contrast, our approach uses a native JVM and Java deserialization filters, and incurs a very small overhead.

8 FUTURE PLANS

The work presented in this paper is in the very early stages and warrants several caveats. Our evaluation is currently limited to one application (WLS), one technology (Java deserialization) and one sampler (Metropolis-Hastings) only. Other applications, samplers, and languages will have to be investigated. Despite these limitations, however, we have uncovered several avenues for further investigation. First, our results suggest that class features and ordering capture the essence of a malicious gadget chain. While attackers can obviously manipulate the stream to evade detection, successful exploitation *requires* specific features and ordering. We believe that this invariant could be key in achieving robustness in the face of adversarial attacks. Second, while our approach seems promising on small datasets, it remains unclear how well it scales up and down and how well it applies in various user scenarios. An empirical evaluation using different workloads, applications and user scenarios is needed to assess the practicality of our approach.

REFERENCES

[1] 2021. JEP 290: Filter Incoming Serialization Data. https://openjdk.java.net/jeps/290.

[2] 2022. Top 10 Web Application Security Risks. https://owasp.org/www-project-top-ten/.

[3] Moritz Bechler. 2017. Java Unmarshaller Security: Turning your data into code execution. https://www.github.com/mbechler/marshalsec/blob/master/marshalsec.pdf.

[4] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).

[5] Stefano Cristalli, Edoardo Vignati, Danilo Bruschi, and Andrea Lanzi. 2018. Trusted execution path for protecting java applications against deserialization of untrusted data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 445–464.

[6] Johannes Dahse, Nikolai Krein, and Thorsten Holz. 2014. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 42–53.

[7] Chris Frohoff. 2015. ysoserial. https://github.com/frohoff/ysoserial.

[8] Chris Frohoff. 2016. Deserialize My Shorts: Or How I Learned To Start Worrying and Hate Java Object Deserialization. http://frohoff.github.io/owaspsd-deserialize-my-shorts/.

[9] Brian Goetz. 2019. Towards Better Serialization. https://cr.openjdk.java.net/~briangoetz/amber/serialization.html.

[10] Ian Haken. 2018. Automated Discovery of Deserialization Gadget Chains.

[11] Gabriel Lawrence and Chris Frohoff. 2015. Marshalling Pickles: How Deserializing Objects Can Ruin Your Day. https://frohoff.github.io/appseccali-marshalling-pickles.

[12] Alvaro Muñoz and Oleksandr Mirosh. 2017. Friday the 13th JSON Attacks. https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf.

[13] Yao Pan, Fangzhou Sun, Zhongwei Teng, Jules White, Douglas C Schmidt, Jacob Staples, and Lee Krause. 2019. Detecting web attacks with end-to-end deep learning. *Journal of Internet Services and Applications* 10, 1 (2019), 1–22.

[14] Or Peles and Roee Hay. 2015. One class to rule them all: 0-day deserialization vulnerabilities in android. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*.

[15] Shawn Rasheed and Jens Dietrich. 2020. A hybrid analysis to detect Java serialisation vulnerabilities. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1209–1213.

[16] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic Programming in Python using PyMC3. *PeerJ Computer Science* 2 (2016), e55.

[17] Jacob Schreiber. 2017. Pomegranate: Fast and Flexible Probabilistic Modeling in Python. *The Journal of Machine Learning Research* 18, 1 (2017), 5992–5997.

[18] Mikhail Shcherbakov and Musard Balliu. 2021. SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web. In *Network and Distributed Systems Security (NDSS) Symposium 202121-24 February 2021*.