Assignment

WASP Software Engineering Course Module 2024

Bruno Kreyssig
Umeå University
Department of Computing Science

June 25, 2024

Contents

Introduction

Our research focuses on the analysis of deserialization vulnerabilities in Java. Specifically, we are concerned with the code reuse attack vector in deserialization gadget chains. The Java serialization API provides convenient functionality for converting object state in runtime into a transfer format, to later be reconstructed (deserialized) by a receiving JVM. During deserialization custom callback methods may be invoked to ensure the reconstructed object can be used within the new environment. For instance, a *HashMap* is required to recalculate the hash values of its keys according to the native (platform dependent) hash function implementation. An attacker can leverage this knowledge to find classes overriding deserialization callback methods (e.g., *HashMap*), which themselves invoke further methods (i.e., gadgets). The idea is then to chain such gadgets together to reach a security sensitive method.

A trivial instance of a gadget chain is shown in the three code snippets below. Going from a deserialization entry point (upper left) one could assume that only instances of *MyClass* can be retrieved from the serialized data stream. Yet, before the object is cast to its intended type, the object is reconstructed. An attacker could supply a *HashMap* containing an object of type *Sink*. Then, upon deserialization and before the cast is resolved, an arbitrary command is executed (lower left).

```
ObjectInputStream s;
// ...
MyClass c = (MyClass) s.readObject();
```

```
class Sink implements Serializable {
  public int hashCode() {
    Runtime.getRuntime().exec(param);
}}
```

```
class HashMap implements Serializable {
  Object key;
  void readObject(ObjectInpusStream s) {
    //...
    key.hashCode();
  }
}
```

We are primarily interested in the development of novel algorithms to detect deserialization gadget chains in Java projects. In addition, we consider evaluation strategies for such algorithms and platform specific questions, e.g., the impact of insecure deserialization on Android.

Related Principles

0.1 Robert's Lectures

Our research project is directly related to software engineering in the sense that we aim to research and develop specific, security-related **software verification** solutions. Indeed, when it comes to detecting gadget chains current research relies heavily on **static analysis** techniques [1, 2, 3, 4, 5, 6], which may be enhanced through the usage of a **fuzzer** [7, 8, 9, 10]. The trend leans into using both techniques, due to the high imprecision of relying solely on static analysis. That is to say, findings through taint and callgraph analysis get piped into a fuzzer to verify, whether these gadget chains can actually be exploited.

In a recent project we developed a benchmark for the evaluation of gadget chain detection algorithms. The idea of the benchmark was to deliver a set of Java libraries (i.e, JAR files) which contain synthetic gadget chains which individually require tools to detect different techniques to chain gadgets together. To verify whether these gadget chains truly existed within the libraries, we wrote **unit tests** to check if the chain was executed. Our software artifact is maintained in a Github repository for **version control**.

0.2 Guest Lectures

There weren't any technically applicable principles in the first presentation by SAAB. However, I can relate to the general **reluctance to use AI** or **ignorance** towards it. So far AI was not used in my research area, even though it received abundant attention recently. Furthermore, speaking for myself, I keep the idea in the back of my head, yet lean into using traditional software verification techniques. My priorities right now are to learn frameworks for static analysis such as *SootUp* and fuzzers such as *Jazzer*. According to the literature review in [11], there is merit to applying machine learning to laborious manual stages of the fuzzing process (e.g., test case generation and exploitability analysis). While I don't believe that AI/ML will be the major contributing factor to my PhD thesis, I found the notion in the guest presentation to be a good reminder. Likely it is possible for me to replace some subtask in my research projects with machine learning. In continuation, the Volvo presentation brought up an interesting concept.

I can see a use case for applying **fuzzy lookups** in an upcoming research project. One of the unsolved problems in deserialization gadget chain detection is determining a holistic set of security sensitive call sites. *Rasthofer et al.* [12] made use of machine learning to determine sink methods within Android APIs through strict features. For instance, methods containing a certain string (*set*) are an indicator for a sink method. There may be an angle to loosen this definition and use fuzzy lookups to evaluate class and method names whether they are likely to be considered a sink method.

CAIN Conference Publications

0.3 Task Description

1. Describe the core ideas
2. How does it relate to your research
3. How would your research in combination with the paper fit into a larger AI project
4. How could your project be changed/adapted to make AI engineering in your project easier (based on the ideas in the paper)

0.4 FuzzSlice: Pruning False Positives in Static Analysis Warnings through Function-Level Fuzzing (ICSE 2024) [13]

Usually fuzzing is used to confirm potential warnings discovered through static analysis. I.e., it outputs true positives for which a crashing input could be generated. Given an exhaustive list of warnings from a static application security testing tools (SAST), it however becomes infeasible to fuzz all warnings within reasonable time [14]. In contrast, *FuzzSlice* [13] describes the novel concept of using fuzzing to prune false positive warnings generated by static analysis.

Instead of fuzzing the entire program from an entry point (e.g., the *main* method), *FuzzSlice* determines the enclosing function of the SAST warning's location. Then, the tool recompiles a function slice including all dependencies required for the execution of a minimal program containing only this function. The idea ultimately is to fuzz only the inputs of this minimal function slice. While observing a crash at the statement flagged as a warning provides no proof of a true positive (it may not be possible to pollute the function with the fuzzed parameters), should the fuzzer fail to generate a crash within a time frame the warning most likely is a false positive. Thus, *FuzzSlice* focusses specifically on the reduction of false positives in SAST reports by elimination of warnings, rather than confirmation of these. *FuzzSlice* showed promising results both on the synthetic *Juliet* benchmark and on three open-source, real-world applications.

This line of work is of high relevance to my field of research. As outlined in Section 0.1, the research trend for Java deserialization gadget chain detection leans towards the usage of hybrid approaches. This makes sense, since in isolation, static analysis and fuzzing are either lacklustre in precision or soundness. More importantly, the idea of analyzing function slices in isolation could potentially be

ported to the verification of gadget chain findings. Given a statically observed gadget chain (e.g., using control flow, class hierarchy and taint analysis), there may be merit in fuzzing each gadget individually and if for any gadget within the chain the fuzzer is incapable of reaching the statement invoking the following gadget, then the gadget chain likely is a false positive.

To apply the publication to my research I first need to evaluate the requirements for minimal function slice creation in Java (*FuzzSlice* is used for projects based on C). The naive approach would be to invoke the function directly from a *fuzzerTestOneInput()* call (e.g., using *Jazzer* [15]) and replace the statement containing the call to the next gadget with an exception. The idea of instrumenting Java bytecode prior to fuzzing in such a way was already used in [9] and [7]. However, instead of only replacing the known sink method with an exception, I would replace each linking gadget within the gadget chain with such an exception and then fuzz all these gadgets individually. The instrumentation step is easiest achieved using the ASM bytecode manipulation framework.

Also, in the publication fully-fledged SAST tools (*RATS* and *INFER*) are employed to generate the static analysis report. The research towards static deserialization gadget chain detection has not reached a level of maturity, where one could take a previous tool and expect a high degree of soundness within the results. For example, finding gadget chains relying on reflection or dynamic proxies remains an unsolved challenge. Thus, the static analysis proportion of the project would have to be developed by us as well, likely using and extending *Soot* in process.

Let us consider how this work relates to AI/ML projects. Larger AI projects are certain to contain sophisticated ETL pipelines for data evaluation and cleansing. Popular Big Data frameworks such as *Apache Hadoop*, *Flink* and *Spark* are all based on the Java platform. Deserialization vulnerabilities were discovered recently in Hadoop (CVE-2023-46674). Consequently, the usage of SAST and fuzzing for the detection of deserialization gadget chains (my research) or any type of vulnerability in general ([13]) is invaluable for keeping AI/ML pipelines secure.

0.5 Automatically Resolving Data Source Dependency Hell in Large Scale Data Science Projects (CAIN 2023) [16]

The authors describe an approach to automatically determine data source dependencies in machine learning from Azure activity graphs. Comprehensive visibility of data source dependencies (including transitive) is crucial to the development of stable ML pipelines in order to keep track of the effects a change in a singular data source has on which parts of the overall model (see pre-reading paper [17])

While the implementation is described specifically for Azure, the concept is generic when substituting the terms **activities** with **machine learning code** and **activity graphs** with **ML pipeline blocks**. For instance, an ML project could be composed of two activity graphs (G_1, G_2), each containing an activity (A_1, A_2) to train a model and a set of dependant data sources. Further the graph G_2 could depend on the output of G_1. From here the authors develop a high level procedure to determine the data source dependencies of an activity graph.

First, the nodes of an activity graph are taken in a FIFO structure, thus ensuring the analysis starts from the last node in the graph. Then, the algorithm propagates backwards until it reaches only initial data sources or a fixpoint, e.g., another connected activity graph. During this process activity nodes are statically analyzed to determine their data source dependencies.

This leads to the low level procedure of how to determine the data source dependencies of a single activity in isolation. Recall that an activity is equivalent to a block of ML code, usually a Python script. At this point the publication ties in with my area of research. While in my project, deserialization entry points or deserialization callbacks are used as a set of source methods, the publication takes a set of typical read input methods (e.g., *pd.read_ csv()*) as sources. Analogously, sink methods in my project are security sensitive call sites, whereas in the publication sink methods are model training statements. The task is then to determine a data flow path from the sources to the sinks.

In [17] data dependency debts were highlighted as the most costly debt in ML systems with the difference to traditional SE projects being that they are more difficult to detect than dependency

debts. Thus, in a larger AI project using program analysis helps to provide visibility on the actual data dependencies (including hidden and transitive) of different parts of ML code. The authors of [16] describe three use cases of their project:

- **Interactive dashboards:** consuming the data of the analysis and connecting it to the data source dependencies. Then, when changes in the upstream data sources are detected, developers can be alerted of these changes and take immediate responsive actions.
- **Knowledge graphs:** as a means to power the interactive dashboards and execute actions upon data source changes.
- **Feature stores:** to detect feature re-usability and thereby reduce storage requirements

Since I work with program analysis in my project, I can see the indirect benefit of improving techniques in this field to the research area of AI engineering, as was seen in the CAIN paper described here. In the context of a larger scale AI project my research could also prove directly useful in detecting vulnerabilities in certain parts of the project. To give a concrete example, Big Data frameworks such as Hadoop or Apache Flink rely on the Java platform. Due to the inherent interconnection of Big Data and ML, it seems very likely to find at least some blocks of an AI project relying on Java. Furthermore, in our recent research [18], we were able to confirm the existence of deserialization gadget chains within these Big Data frameworks. Indeed, within Hadoop a deserialization vulnerability was found very recently (CVE-2023-46674). Therefore, overall, I can see how my research is vital to keeping AI projects safe from malicious actors.

References

[1] Ian Haken. "Automated Discovery of Deserialization Gadget Chains". en. In: 2018. URL: https://i.blackhat.com/us-18/Thu-August-9/us-18-Haken-Automated-Discovery-of-Deserialization-Gadget-Chains.pdf.

[2] Luca Buccioli et al. "JChainz: Automatic Detection of Deserialization Vulnerabilities for the Java Language". In: *Security and Trust Management: 18th International Workshop, STM 2022, Copenhagen, Denmark, September 29, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, Apr. 2023, pp. 136–155. ISBN: 978-3-031-29503-4. DOI: 10.1007/978-3-031-29504-1_8. URL: https://doi.org/10.1007/978-3-031-29504-1_8 (visited on 01/21/2024).

[3] Xingchen Chen et al. "Tabby: Automated Gadget Chain Detection for Java Deserialization Vulnerabilities". In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. ISSN: 2158-3927. June 2023, pp. 179–192. DOI: 10.1109/DSN58367.2023.00028. URL: https://ieeexplore.ieee.org/abstract/document/10202660 (visited on 10/03/2023).

[4] Junjie Wu, Jingling Zhao, and Junsong Fu. "A Static Method to Discover Deserialization Gadget Chains in Java Programs". In: *Proceedings of the 2022 2nd International Conference on Control and Intelligent Robotics*. ICCIR '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 800–805. ISBN: 978-1-4503-9717-9. DOI: 10.1145/3548608.3559310. URL: https://dl.acm.org/doi/10.1145/3548608.3559310 (visited on 10/29/2023).

[5] Yifan Luo and Baojiang Cui. "Rev Gadget: A Java Deserialization Gadget Chains Discover Tool Based on Reverse Semantics and Taint Analysis". en. In: *Advances in Internet, Data & Web Technologies*. Ed. by Leonard Barolli. Cham: Springer Nature Switzerland, 2024, pp. 229–240. ISBN: 978-3-031-53555-0. DOI: 10.1007/978-3-031-53555-0_22.

[6] Sicong Cao et al. "Improving Java Deserialization Gadget Chain Mining via Overriding-Guided Object Generation". In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, July 2023, pp. 397–409. ISBN: 978-1-66545-701-9. DOI: 10.1109/ICSE48619.2023.00044. URL: https://dl.acm.org/doi/10.1109/ICSE48619.2023.00044 (visited on 03/22/2024).

[7] Shawn Rasheed and Jens Dietrich. "A hybrid analysis to detect Java serialisation vulnerabilities". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. New York, NY, USA: Association for Computing Machinery, Jan. 2021, pp. 1209–1213. ISBN: 978-1-4503-6768-4. DOI: 10.1145/3324884.3418931. URL: https://dl.acm.org/doi/10.1145/3324884.3418931 (visited on 10/10/2023).

[8] Sicong Cao et al. "ODDFuzz: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing". English. In: IEEE Computer Society, May 2023, pp. 2726–2743. ISBN: 978-1-66549-336-9. DOI: 10.1109/SP46215.2023.10179377. URL: https://www.computer.org/csdl/proceedings-article/sp/2023/933600c726/1OXH0xA0Lrq (visited on 03/22/2024).

[9] Prashast Srivastava et al. "Crystallizer: A Hybrid Path Analysis Framework to Aid in Uncovering Deserialization Vulnerabilities". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1586–1597. ISBN: 9798400703270. DOI: 10.1145/3611643.3616313. URL: https://dl.acm.org/doi/10.1145/3611643.3616313 (visited on 01/21/2024).

[10] Bofei Chen et al. "Efficient Detection of Java Deserialization Gadget Chains via Bottom-up Gadget Search and Dataflow-aided Payload Construction". English. In: ISSN: 2375-1207. IEEE Computer Society, Feb. 2024, pp. 150–150. ISBN: 9798350331301. DOI: 10.1109/SP54263.2024.00150. URL: https://www.computer.org/csdl/proceedings-article/sp/2024/313000a150/1Ub248fSq9G (visited on 02/23/2024).

[11] Yan Wang et al. "A systematic review of fuzzing based on machine learning techniques". In: *PLOS ONE* 15.8 (Aug. 2020). Publisher: Public Library of Science, pp. 1–37. DOI: 10.1371/journal.pone.0237749. URL: https://doi.org/10.1371/journal.pone.0237749.

[12] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks". In: Jan. 2014. ISBN: 978-1-891562-35-8. DOI: 10.14722/ndss.2014.23039.

[13] Aniruddhan Murali et al. "FuzzSlice: Pruning False Positives in Static Analysis Warnings through Function-Level Fuzzing". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3623321. URL: https://doi.org/10.1145/3597503.3623321.

[14] Marcel Böhme and Brandon Falk. "Fuzzing: on the exponential cost of vulnerability discovery". In: Nov. 2020, pp. 713–724. DOI: 10.1145/3368089.3409729.

[15] CodeIntelligenceTesting. *GitHub - CodeIntelligenceTesting/jazzer: Coverage-guided, in-process fuzzing for the JVM*. URL: https://github.com/CodeIntelligenceTesting/jazzer.

[16] Laurent Boué, Pratap Kunireddy, and Pavle Subotić. "Automatically Resolving Data Source Dependency Hell in Large Scale Data Science Projects". In: *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*. 2023, pp. 1–6. DOI: 10.1109/CAIN58948.2023.00009.

[17] D. Sculley et al. "Hidden technical debt in Machine learning systems". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 2503–2511.

[18] Bruno Kreyßig and Alexandre Bartel. *Analyzing Prerequisites of known Deserialization Vulnerabilities on Java Applications*. en. 2024. URL: https://www.abartel.net/static/p/ease2024-javaDeser.pdf.