USENIX Security '25 Artifact Appendix: Precise and Effective Gadget Chain Mining through Deserialization Guided Call Graph Construction

Yiheng Zhang[1,2], Ming Wen[1,2,*], Shunjie Liu[2], Dongjie He[4], and Hai Jin[1,3]

[1]*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST)*
[2]*Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, HUST, China*
[3]*Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, China*
[4]*Chongqing University, China*

A Artifact Appendix

A.1 Abstract

Our artifact, FLASH, is a tool designed to detect potential gadget chains in Java applications by constructing deserialization-guided call graphs. FLASH begins with a novel Deserialization-Driven Controllability Analysis, which determines whether variables can be restored through deserialization. Based on the controllability of variables, FLASH performs a series of downstream tasks, including hybrid dispatch and reflection analysis, thereby building a more comprehensive and precise deserialization call graph. Utilizing the deserialization call graph, FLASH applies an existing detection algorithm [1] to identify gadget chains.

Evaluation conducted on benchmarks demonstrates that FLASH achieves higher recall (with 30.8% lower false negative rate) and precision (with 25.9% lower false positive rate) than state-of-the-art methods in gadget chain detection, at the cost of limited overhead. Moreover, FLASH successfully discovers 90 new gadget chains.

A.2 Description & Requirements

This section provides essential information regarding the usage of FLASH, including its access link, required software dependencies such as the Java runtime environment or Docker, the location of the benchmark used in our experiments, and safety considerations when verifying the exploitability of detected gadget chains.

A.2.1 Security, Privacy, and Ethical concerns

It is important to note that the execution of a gadget chain ultimately triggers dangerous functions. Therefore, when verifying the exploitability of a gadget chain, we replace the malicious payload with a benign command (e.g., opening a calculator or referencing a non-existent address) to ensure safety.

A.2.2 How to Access

We have open-sourced FLASH on Zenodo at `https://doi.org/10.5281/zenodo.15606159`, where it is publicly available for download.

A.2.3 Hardware Dependencies

None

A.2.4 Software Dependencies

If users choose to compile and run FLASH from source, a JDK 17 environment is required. Alternatively, we provide a pre-built executable version of FLASH and a Docker-based environment, allowing users to run the tool with only Docker installed. Additionally, to detect gadget chains under a specific JDK version, users need to require the corresponding JDK dependency libraries, which are essential for FLASH to detect gadget chains and validate the exploitability of the identified gadget chains.

A.2.5 Benchmarks

To facilitate evaluation, we have open-sourced our benchmark of Java applications containing our detected new gadget chains in the `Flash_GC.zip` file. FLASH allows users to specify target Java applications for analysis and automatically detects gadget chains contained in these applications.

*Ming Wen is the corresponding author

A.3 Set-up

This subsection primarily introduces the set-up steps required before using FLASH.

A.3.1 Installation

We recommend using IntelliJ IDEA and JDK 17 to run FLASH from source code. Specifically, after downloading the source code, users can open the project directly in IntelliJ IDEA. Then, users can follow the configuration steps for the Tai-e framework [3] to set up the project, as described in the official documentation [2].

A.3.2 Basic Test

To verify that all required components for FLASH are properly set up, users can perform a functionality test by analyzing a sample Java application, which requires specifying runtime parameters in a YAML configuration file. We list and explain some of the key configuration parameters as follows:

- `appClassPath`. Users need to specify the target Java application to be analyzed.
- `sources`. Users need to specify the entry points for analysis. The keyword `serializable` can be used to include all deserialization-related methods, such as `readObject`, `readExternal`. Alternatively, users can specify other methods for analysis by providing their method signatures.
- `filterNonSerializable`. Users can configure to filter out methods whose declaring classes do not implement `java.io.Serializable`.
- `GC_MAX_LEN`. Users can configure the maximum length of a gadget chain that FLASH will search for during analysis.
- `priori-knowledge`. Users need to specify the file that contains manually created summaries for JDK built-in APIs (e.g., String and JavaBean).

We include a sample configuration file in our artifact to help users get started. Users can configure IntelliJ IDEA to pass this file to FLASH by specifying it with the `-options-file` parameter. We also recommend setting VM options such as `-Xss` or `-Xmx` to allocate additional memory to FLASH for better performance. If FLASH outputs runtime logs (e.g., the loading of sinks and the number of analyzed methods), it indicates that it functions correctly and is ready for use.

A.4 Evaluation Workflow

Since our primary focus is on gadget chain detection, this subsection mainly evaluates the functionality of FLASH in detecting gadget chains.

A.4.1 Major Claims

(C1): Compared to the state-of-the-art approaches, FLASH achieves higher recall (with 30.8% lower false negative rate) and precision (with 25.9% lower false positive rate) in gadget chain detection. In total, it identifies 90 new gadget chains. This is demonstrated by experiment (E1), as described in Section A.4.2. The results are also presented in Table 1 of our paper.

A.4.2 Experiments

(E1): [Gadget Chain Detection] [30 human-minutes + 1 compute-hour + 1GB disk]: This experiment focuses on detecting potential gadget chains within a specified application. The expected results are the concrete gadget chain call stacks. Note that the time estimation refers to the average time required for analyzing a single target.
Preparation: Users should configure the required parameters (e.g., the target application to be analyzed) in accordance with the procedure outlined in Section A.3.2.
Execution: FLASH can be launched by simply running the IntelliJ IDEA.
Results: The expected results of the experiment are concrete gadget chains, represented as method call sequences. For instance: `Source#readObject` → `Link1#callee` → `Sink#sink`. It is worth noting that manual validation is necessary for the detection results. To facilitate this, users may refer to the gadget chains and corresponding proof-of-concepts (POCs) examples included in our benchmark as a basis for constructing and testing new exploit POCs.

To simplify large-scale analysis, we provide a pre-configured Docker environment and an accompanying bash script that automates the gadget chain detection of all Java applications within a specified directory. If you run the script, it will naturally require more time.

A.5 Notes on Reusability

First, the deserialization call graph generated by FLASH can serve as a foundational component for other research directions, such as fuzzing-based validation. Users can also easily export and store the generated call graph for further analysis.

Second, FLASH can be easily extended to support other types of vulnerability detection, such as web injection attacks. By modifying the configuration file to specify a web API as the source, and disabling the deserialization-specific analysis in the code, users can adapt FLASH to new detection scenarios with minimal efforts.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodol-

ogy followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[1] Xingchen Chen, Baizhu Wang, Ze Jin, Yun Feng, Xianglong Li, Xincheng Feng, and Qixu Liu. Tabby: Automated gadget chain detection for java deserialization vulnerabilities. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 179–192. IEEE, 2023.

[2] Tian Tan. Tai-e Reference Documentation, 2025. https://tai-e.pascal-lab.net/docs/current/reference/en/index-single.html#setup-tai-e-in-intellij-idea.

[3] Tian Tan and Yue Li. Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1093–1105, 2023.