CALL FOR BRIEFINGS (formerly Call for Papers) **Submission Samples**

Explore these high-scoring, accepted Briefings to inform your submission

Reach out to cfp@blackhat.com with questions

BLACK HAT EUROPE 2025

Title The Forensic Trail On GitHub: Hunting For Supply Chain Activity

Speakers
Rami McCarthy, (Principal Security Researcher, Wiz)
Amitai Cohen, (Attack Vector Intelligence Lead, Wiz)

Abstract
Ultralytics. tj-actions. Grafana. GitHub Actions are increasingly targeted by attackers and implicated in industry-impacting incidents. Thankfully, GitHub's public surface offers numerous threat intelligence sources for the discerning defender. This talk covers a comprehensive methodology for investigating and tracking real-world supply chain attacks exploiting GitHub Actions, inspired by our work responding to the aforementioned incidents. It adds a new dimension and set of tools to threat intelligence research.
We'll expose the wealth of intelligence available directly from both GitHub and the underlying Git plane. Through concrete demos, we'll show how to effectively pivot on user metadata and behavioral heuristics, uncover attacker forks, and recover deleted gists and commits. We'll also demonstrate how to trace attacker aliases, identify targets of reconnaissance, and unmask attackers and researchers in real-time. Attackers are hiding in the complexity of this ecosystem, but with automation we can peel back the noise, empowering detection and investigation.
This approach is practical, repeatable, and relies exclusively on publicly available data, ensuring accessibility for all defenders without the need for private threat intelligence feeds.

Presentation Outline
I. GitHub in the Crosshairs: A Recent History of Escalating Attacks
* Why GitHub Actions are Prime Targets:
* Role in CI/CD pipelines (automation, permissions, access to sensitive assets).
* Trust in open-source components and ripple effect of compromise.
* Case Studies: Anatomy of Recent Compromises
* 1\. Ultralytics: Supply chain attack exploiting GitHub Actions for malicious PyPI package injection.
* 2\. `tj-actions`: Multi-step supply chain attack leveraging mixed techniques with specific and broad impact.
* 3\. Grafana: Evidence of attackers racing bug bounty hunters in this space.
* Broader Context: Other Reasons to Investigate GitHub:
* [DPRK Contagious Interview campaign](https://unit42.paloaltonetworks.com/north-korean-threat-actors-lure-tech-job-seekers-as-fake-recruiters/).
* [GitHub usage in attacks (C2, data exfil)](https://go.recordedfuture.com/hubfs/reports/cta-2024-0111.pdf).
* [Attackers leaking intelligence (e.g., APT37)](https://www.zscaler.com/blogs/security-research/unintentional-leak-glimpse-attack-vectors-apt37).
II. Under the Hood: The GitHub & Git Ecosystem for Defenders
* Fundamental Concepts for Forensic Investigation:
* GitHub is built on Git.

* Git Basics: Commits, branches, repositories, tags.
* GitHub Concepts: Forks, Pull Requests (PRs), Gists, the GitHub firehose.
* Complexity as Concealment & The "Open Source Problem":
* While attackers leave a public record, they leverage system complexity (decentralization, volume, rapid changes) to hide.
* Our methodology aims to automate and cut through this noise.
* The "[open source problem](https://cybersecpolitics.blogspot.com/2024/04/the-open-source-problem.html)": Complex contributor graphs, many with "suspicious" or "low reputation" profiles.
III. Challenges in Public Investigations on GitHub
* Lack of public comms from Github: No public details on their investigations, hindering definitive confirmation of public assessments.
* Data Opacity:
* Not all data is public (e.g., IP addresses).
* Not all events are publicly logged (e.g., updating Tags).
* Commit data is spoofable (e.g Email Address).
* Evading the GitHub Firehose: Private account settings prevent direct logging.
* Evidence Burial by Cleanup:
* Examples from `tj-actions` incident:
* Nuked repositories hid incident context (e.g., issues).
* Deleted gists complicated payload investigation.
* Deleted forks obscured malicious commits.
IV. Public Intelligence Sources & Access Methods
* GitHub API: Programmatic access to public data.
* GitHub Archive: Access via BigQuery vs. ClickHouse.
* The Underlying Git Plane: Local cloning for deeper inspection.
V. Investigation Methodology
* Investigating Users:
* Techniques:
* Metadata via API: Followers/Following, Watched Repositories, Joined Organizations, Code snippets, Linked domains/social profiles, SSH public keys.
* Inferring Private User Activity: Observing interactions with public entities.
* Heuristics: Active hours/Timezone/Geography, Language.
* Retrieving User Emails from Commits: Distilling username history from `@users.noreply.github.com` emails.
* Demo: Investigating an Actor:
* Using a bug bounty hunter as an example.
* Show interactions with private/deleted accounts, deanonymizing aliases.
* Show deleted commits/PRs revealing research targets.
* Show evidence of reproduction by triagers and target remediation.
* Investigating Attacks and Attackers:
* Pivoting off Payloads:
* Payload code, Exfiltration URLs, Payload hosting (e.g., Gist).
* Demo: Recovering Deleted Payloads
* Deleted commits/forks via cross-branch references.
* Deleted gists via direct clone of dangling Git object.
* Exfiltration Analysis:
* Observation via public tools (e.g., `webhook.site`).

* Analysis of GitHub Actions logs or artifacts.
* Looking for Absence as Evidence ("Missing Stairs"):
* 404'd Users, Tip to find renamed ones: search GitHub for comments and interactions.
* Missing PRs/Issues in sequence.
* Missing action logs.
* Demo: Automated detection of suspicious absence
* Demo: Monitoring Attempted Pwn Requests
* GUI tool for highlighting potential payloads within the Github Firehose
* suspicious branch names, PR titles, etc
* Showing how this regularly highlights successful bug bounties.

What New Research, Concept, Technique, or Approach is Included?
We take a broad set of git internals, GitHub features, and known OSINT approaches, and demonstrate a novel and consolidated approach to apply them to threat hunt supply chain attacks and forensically investigate incidents involving GitHub. We also will cover case studies on real attacks, open-source new tooling for our demos, and prove out the research against in-the-wild activity.

What Problem Is Solved?
There has been a surge in Github based attacks, including the newsmaking Ultralytics, tj-actions, and Grafana incidents. Having worked the response to those incidents, we found that there is a knowledge and capability gap in threat hunting and investigation. This presentation improves the state of the art in the industry on investigating this class of attacks.

Provide 3 Takeaways
1. Attacker activity on GitHub allows differentiated investigation due to the predominantly public nature of the platform, with some critical limitations
2. By monitoring the public GHArchive firehose, it's possible to proactively identify attempted and successful attacks in the wild
3. GitHub's underlying `git`-based architecture opens up forensic capabilities, including recovery of "deleted" data (like gists and commits on forks), and incidental disclosure of useful information (like emails on commits)

Is This Content New? Yes, this content is new.

If this content is currently new, do you plan to publish/present it prior to Black Hat EU?
No, we plan to debut this content at Black Hat EU
Have You/Do You Plan to Submit This Talk to Another Conference? No

Is This a New Vulnerability? No

Will You Be Releasing a New Tool?
No, but we will be open-sourcing the code for our demos. This complements our presentation by making it practical instead of merely theoretical.
* investigating a threat actor on GitHub
* Detecting suspicious absence / deleted data
* Recovering Deleted Payloads
* Monitoring Attempted Pwn Requests

Will Your Presentation Include a Demo?
Yes, we plan multiple demos:
* investigating a threat actor on GitHub
* Detecting suspicious absence / deleted data
* Recovering Deleted Payloads
* Monitoring Attempted Pwn Requests

Does your organization offer a solution to this issue? No

BLACK HAT EUROPE 2025

Title Abstractions For Program Analysis

Speaker Kyle Martin, (Engineer, Vector 35)

Abstract
We have an abundance of representations for code; from source code, to assembly, to decomplied code, to dataflow, block diagrams, and so much more. But why? Simply, because there is a best abstraction for every problem you want to solve. But knowing what abstractions are available and when to choose one over another is hard! In this talk, we'll delve deep into intermediate languages, discuss why you should be using them to solve the tasks at hand, and show you how to choose the best abstraction for the job. In doing so, we'll reveal the secrets that power compilers and decompilers (and even processors!) alike to perform analyses that find bugs, vulnerabilities, optimize code, and so, so much more...all without having to read a single line of code!

Presentation Outline
Brief overview:
- Definitions
- Normal Forms
- SSA
- SSI
- Demos

While I consider the concepts of what I'll be covering to be rather simple, the verbage often isn't. I'll define semantics as "the meaning of code" and walk through assembly, C, and Rust examples to showcase how different contexts, different implementations, different abstractions have different amounts of recoverable meaning. As we're abstracting, we're not always losing information! Often, in fact, we're gaining it! To that end, I define programs as mechanisms for moving and manipulating data, for which we can use Types to describe those data and their interactions. Continuing to zoom out, we can define entire programs as types (they have data, they have private/internal functions that move and manipulate those data, and they have public/external interfaces for accepting and sending those data), at which point we need to define "semantic scope;" depending on the assumptions about a system or rules we impose on it, how zoomed in or out we are, we have different abstractions.

Normal forms are our first step: once we believe that we can represent the same code in multiple ways, let's look at the simplest! I define normal forms (as instructions which can be reduced no further), a-normal form, conjunctive normal form, and others. They each have their own set of rules; these rules define their semantics; and their semantics allow for different properties to become obvious. In a-normal form, we can track dependencies and side effects, removing or ignoring instructions that we don't need to know. CNF, used by Z3 and other equation solvers, reveals various algebraic identities and enables us to solve extremely complex numerical problems efficiently…including defeating obfuscations. I'll include or remove additional forms here as I get initial feedback from practicing my talk.

SSA brings us to cutting-edge compiler and decompiler research, and it's the most important one for people to understand; it's used by nearly every program analysis tool out there, and more than likely you've read it before (and possibly been confused by how it works!). It has just two rules: all variables are read only, and future reassignments of a variable are assigned new versions instead. For the sake of brevity, I won't quote the entirety of LLVM's analysis catalogue to demonstrate the capabilities of SSA: LLVM's entire core analysis is performed exclusively on an SSA form of the code. But what about branches? What about loops? If you can't define a variable in two places, then what happens if it's true on one side and false on the other? What do you return? And what about variables being incremented in loops? Thankfully we don't need to unroll all our loops - otherwise we'd also have to solve the halting problem, but SSA form provides us the ability to track variables through loops without having to worry about redefinitions. I'd explain it more here in text, but it's best defined in my slides with visualizations. Once defined, however, I walk through several examples of analyses that would be nearly impossible in non-SSA form, but very easy in SSA.

One of my motivating examples is CVE-2021-31956; the bug is simple: a simple buffer underflow in the Windows kernel, in NTFS, caused by not checking that the right hand side of a subtraction is less than the left hand side; modeling that behavior abstractly for discovery over normal forms is also easy! Verifying, however, becomes hard: you need to find the last time the variable was defined, then trace all possible uses of that variable between those definitions and the subtractions you care about to see if any of those instructions properly constrain the variable. If we instead use SSA form, we already know exactly which assignment is being used at the location of the subtraction, and we can check all other uses of that specific version to determine if it's properly constrained. While that might not be completely sound (this would be a flow insensitive analysis for a problem that is flow sensitive…it turns a subset of true positives into false negatives), it enables us to find bugs quickly by eliminating all false positives from our simple initial discovery. I show all this in my slides with some great visualizations of the actual code paths…showing you the multiple definitions you'd need to reconcile in non-SSA, and the singular definition site you get in SSA that allows this to be nearly trivial.

My last form will be SSI (single static information), a form used in some very niche program analysis tools, not GCC, Clang/LLVM, Binary Ninja, Ghidra, IDA, nor any of the other main tools. It follows similarly from SSA, but also propagates constraints on variables to their use-sites. So now, instead of having to walk backwards through use-def chains in SSA to gather them yourself, you just get them for free.

Reverse engineers, vulnerability researchers, malware analysts, and everyone else who reads code for security want these abstractions! But not everyone realizes they exist, and that our tooling can provide them to us. While it would be nice to have SSI and some of the others in our tools for us to use, engineers can create those representations themselves and reap the benefits of this talk. In the meantime, researchers and engineers can start to leverage the existing capabilities of normal forms and SSA to automatically find real bugs in the real world, using the same techniques often obfuscated by compiler researchers' program analysis white papers and tooling.

Depending on timing and feedback, I have several analysis scripts which perform the various analyses I've discussed here which I may show, though I'm currently tending on the side of leaving my demos in the slides.

What New Research, Concept, Technique, or Approach is Included?

What's exciting about this talk, for reverse engineers and vulnerability researchers, is getting to see forms of code in action that you usually only hear about in abstract or in research. I'm presenting old concepts, but in an accessible way. This talk is motivated by my students begging me to give a talk on specially SSA form (single static assignment)….most researchers have seen LLVM's bitcode before, but don't understand how to read it, why it's able to perform so well, or what could be even better than SSA for reverse engineering certain tasks. I suppose I can claim this as "new research," in that it touches on a lot of current PhD research….including research efforts lead by DARPA in several programs….but fundamentally this talk aims to help bring old program analysis concepts from the 80s, 90s, and 00's to the modern engineer's toolbox and motivate them with the future capabilities of existing representations that are not yet integrated into our popular program analysis frameworks. So while I don't personally consider this research to be novel, others are working on research relating multiple abstractions, and there are many novel insights to be gained from how I relate them to each other and use them together. The best testimonial I've gotten is from a VR team's most senior engineer after going through some of these topics: "This is unbelievably useful. I wish I learned about all this five years ago!"

What Problem Is Solved?

Vulnerability discovery, point of interest identification, modeling program behavior, taint tracking, dataflow analysis, types analysis, exploitability verification, and the automation of every-day program analysis questions.

Provide 3 Takeaways

1. Why using abstractions for program analysis is better for modeling program behavior and discovering points of interest than classic "regex over bytes"
2. What you get for free from each abstraction that allow you reimplement those old heuristics more generally, so you can find them in more programs on any architecture of any platform
3. Then finally, how to approach writing automated analyses to find bugs and verify the exploitability of vulnerabilities practically without having to specialize them for each ta

---

Black Hat Asia 2025

Title JDD: In-depth Mining of Java Deserialization Gadget Chains via Bottom-up Gadget Search and Dataflow-aided Payload Construction

Speakers & Contributors

Bofei Chen, (Ph.D Candidate, Fudan University)
Yinzhi Cao, (Associate Professor, Johns Hopkins University)
Lei Zhang, (Assistant Professor, Fudan University)
Xinyou Huang, (Master's Student, Fudan University)
Yuan Zhang, (Professor, Fudan University)
Min Yang, (Professor, Fudan University)

Abstract

Java serialization and deserialization facilitate cooperation between different Java systems, enabling

convenient data and code exchange. However, a significant vulnerability known as Java Object Injection (JOI) allows remote attackers to inject crafted serialized objects, triggering internal Java methods (gadgets) and resulting in severe consequences such as remote code execution (RCE). Previous works have attempted to detect and chain gadgets for JOI vulnerabilities using static searches and dynamic payload construction via fuzzing. However, these methods face two key challenges: (i) path explosion in static gadget searches and (ii) a lack of fine-grained object relations connected via object fields in dynamic payload construction.
First, we introduce a gadget fragment-based summary and bottom-up search approach to address the path explosion challenge.
Second, we then demonstrate how to infer the dataflow dependencies between injection objects' fields and use them to guide dynamic fuzzing to generate exploitable objects.
We evaluate JDD upon six popular Java applications (e.g., Apache Dubbo, Sofa-RPC, Solon, etc) in their latest version, which finds 127 zero-day exploitable gadget chains with six Common Vulnerabilities and Exposures (CVE) identifiers assigned (i.e., CVE-2023-35839, CVE-2023-29234, CVE-2023-39131, CVE-2023-48967, CVE-2024-23636, and CVE-2023-41331).
Each of these CVEs has a CVSS score of 9.8, indicating an extremely high risk of exploitation and the potential to cause significant security damage. Given the wide range of impacts and potential consequences of these vulnerabilities, the related developers patched all these gadget chains in a prompt and timely manner after we reported our findings.

Presentation Outline

/** Brief outline **/
Introduction
Background and Technique challenges
Challenge I: path explosion in static gadget searches.
Challenge II: complex object field relations.
Methodology
Stage I: Gadget fragment-based summary and bottom-up gadget searching approach. Step 1: Identifying deserialization entry points.
Step 2: Identifying gadget fragments with static taint analysis.
Step 3: Linking gadget fragments to construct gadget chains using a bottom-up approach. Stage II: Dataflow-aided injection object generation.
Step 4: Constructing IOCD based on Injection Object related Constraints. Step 5: IOCD-enhanced Directional Fuzzing.
Evaluation and Interesting zero-day gadget chains Takeaways + Q&A
/** Detailed outline **/
Part I: Introduction
In this (brief) part of the talk, we will highlight:
First, we will introduce the threat model for Java deserialization vulnerabilities and the importance of detecting and defending against this type of vulnerability.
Second, we will discuss the powerful evolutionary capabilities of Java deserialization gadget chains and the wide range of incompletely fixed patching problems they cause.
Java deserialization vulnerability is typically assigned extremely high-risk ratings and governance priorities given their broad reach and serious potential consequences. However, since deserialization is a fundamental mechanism, developers are often unable to directly block the entire gadget chain to ensure that normal business functions are not affected. Coupled with the dynamic features of Java and the widespread use of third-party components, the attackers could find many alternatives for the blocked

gadgets in the patch and bypass existing defenses, resulting in long-term security risks.
Part II: Background and Technique challenges
In this part, we will first introduce how to exploit the Java deserialization vulnerability through a motivating example.
Typically, an attacker would first find a gadget chain, which is an execution path that can invoke a security-sensitive method.
Then, the attacker constructs an injection object, and sends it to the target server to trigger the gadget chain and achieve attack consequences.
Next, we will briefly outline the detection methods employed in previous research. Finally, we will summarize the challenges that remain unaddressed by existing prior works, using this example as a basis.
Path explosion. Prior works often adopted a top-down static approach for checking gadgets extensively from a source to a sink for potential paths. However, during the search process, it is easy to detect many dynamic method calls that an attacker can control. As a result, top-down candidate search methods may grow exponentially with the search length and suffer from a severe path explosion problem. Thereby, the construction space of gadget chains cannot be fully explored in a limited time, leading to a large number of underreporting.
Complex object field relations. To trigger the attacker's expected method invocation sequence, the injected object usually has complex parallel and embedded object structure and field-related constraints and dependencies. Thus, it is difficult for random mutations to generate a valid payload to validate the gadget chains' exploitability.
Part III: Methodology
To address these two challenges, we designed an advanced automated JOI vulnerability detection tool called JDD, which was presented at IEEE Symposium on Security and Privacy 2024. It consists of two main stages with five steps. In Stage I, JDD detects possible gadget chains. Then, JDD generates injection objects to verify the gadget chains' exploitability.
Below, we will elaborate on the solution crafted to overcome the aforementioned technical challenges. Stage I: Gadget fragment-based summary and bottom-up gadget searching approach
Step 1. JDD extracts entry points from both the deserialization methods of Java language and popular Java deserialization protocols.
Step 2. JDD detects useful gadget fragments that could be chained together to construct gadget chains from entry points using static analysis. First, JDD starts from each source until the next dynamic method invocation and treats code in between as a fragment. Then, for each fragment, JDD performs data-flow analysis to record the taint mapping relationship between the last method and the first method parameters in the fragment, as well as the conditions on how to link the fragment.
Step 3. Upon completing the search of gadget fragments, JDD performs a bottom-up search to chain fragments together, thereby constructing all potential gadget chains. Our key idea is to fully reuse known sink knowledge (sink reachability, exploit conditions, etc.) to reduce repetitive analyses and thus reduce search complexity.
Stage II: Dataflow-aided injection object generation
Then, we will introduce how to generate injection objects for each gadget chain. and verify the exploitability of the chain by detecting whether the object can trigger the expected attack behavior.
Step 4. JDD follows the call sequence of the detected gadget chain to extract the execution paths, and then uncovers constraints that affect inputs. After extracting these constraints, JDD uses a novel data structure, termed Injection Object Construction Diagram (\textit{IOCD}) to model the object structure and dependencies of fields.
Step 5. JDD first generates an initial seed with a suitable structure based on the IOCD. Then, JDD utilizes IOCD to enhance the efficiency of fuzzing from three aspects: (i) select appropriate fields for mutation. (ii)

reduce the uncertain mutation space through constraint information. And (iii) ensure that the validity of the overall structure of the object is not compromised via a dependency-aware seed mutation.
Part IV: Evaluation and Interesting zero-day gadget chains
We evaluated the effectiveness of JDD on known datasets and popular Java components in the real world. Firstly, we employed JDD to analyze the latest versions of several popular Java open-source components, including Apache Dubbo, Sofa-RPC, Solon, Motan, etc. JDD discovered 127 zero-day exploit chains, earning recognition and acknowledgments from the respective vendors. Six Common Vulnerabilities and Exposures (CVE) identifiers were assigned to these discoveries, including CVE-2023-35839, CVE-2023-29234, CVE-2023-39131, CVE-2023-48967, CVE-2024-23636, and CVE-2023-41331. Each of these CVEs has a CVSS score of 9.8, indicating a very high risk of exploitation and the potential to cause serious security harm. Due to the extensive impact and severe potential consequences of these vulnerabilities, developers have patched all of them.
Additionally, we compared JDD with existing approaches such as ODDFuzz, GCMiner, and GadgetInspector using the well-known gadget chain dataset, i.e., ysoserial. The experimental results show that JDD detects 91 additional, previously undisclosed exploit chains compared to the current state-of-the-art tools.
In addition, we will also share some interesting zero-day exploitable gadget chains detected by JDD. Further explain the strong derivability of such vulnerabilities. Attackers could find many alternatives for the blocked gadgets in the patch and derive new exploitable gadget chains from the original ones; or use them to attack other Java projects that rely on different third-party components.
Illustrate the expanded attack vectors of some unknown gadget chains detected by JDD in comparison to the known chains. For example, if the gadget chain relies on fewer and more commonly used Java dynamic features, there may be more protocols it can exploit.
Part V: Finally, we will wrap up with key takeaways, leaving 5 minutes for questions.
We'll also summarize the talk's key points and the tools presented.
Two core technical challenges that previous state-of-the-art gadget chain detection tools failed to effectively address.
A fragment-based summary and bottom-up gadget chain detection approach that effectively tackles the challenge of static path explosion.
A technical framework that utilizes a lightweight static taint anal

Is This Content New (Not Previously Presented/Published)?
No. JDD was published at IEEE Symposium on Security and Privacy 2024, a prestigious conference in the cybersecurity field. Paper URL: redacted

Have You/Do You Plan to Submit This Talk to Another Conference?
Conference: IEEE Symposium on Security and Privacy 2024, a prestigious conference in the cybersecurity field.
/** What's new **/
In this talk, we will share more detailed schema design, tool implementation, and usage insights:
Provide more details on how to (1) generate summaries for gadget fragments; (2) link gadget fragments via a bottom-up search; and (3) generate exploitable injection objects based on IOCDs

We will share more interesting zero-day gadget chains detected by JDD.

What new research, concept, technique, or approach is included in your submission?
We design a novel gadget chain detection framework, called JDD, to statically detect possible gadget chains using a bottom-up approach and dynamically generate payloads, i.e., exploitable injection objects,

relying on dataflow relations between object fields. JDD addresses the two key technique challenges in Java deserialization gadget chain detection.
JDD solves the path explosion problem of static search for chained gadgets via a bottom-up approach that discovers intermediate gadget fragments and then chains them together from sinks to sources.
JDD constructs the so-called Injection Object Construction Diagram (IOCD) to better model dataflow dependencies between object fields and assist dynamic fuzzers for payload construction.

Provide 3 Audience Takeaways.
Two core technical challenges that previous state-of-the-art gadget chain detection tools failed to effectively address.
A fragment-based summary and bottom-up gadget chain detection approach that effectively tackles the challenge of static path explosion.
A technical framework that utilizes a lightweight static taint analysis engine to drive directed fuzzing.

If applicable, what problem does your research solve?
Lack of effective automatic tools for Java deserialization gadget chain detection and exploitable injection object generation. JDD could more comprehensively explore the construction space of gadget chains, thereby offering developers more insightful solutions for addressing Java deserialization vulnerabilities. For example, JDD could assist developers in designing and verifying blacklists and whitelists. This ensures they effectively block all potential exploit chains to the greatest extent possible.

Will You Be Releasing a New Tool? If Yes, Describe the Tool.
Yes. We released JDD, an automated tool for Java deserialization gadget chain detection and injection object generation.

Is This a New Vulnerability? If Yes, Describe the Vulnerability.
Yes. JDD detected 127 zero-day gadget chains and was assigned 6 CVE IDs.
(CVE-2023-35839, CVE-2023-29234, CVE-2023-39131, CVE-2023-48967, CVE-2024-23636, and CVE-2023-41331)
Their descriptions are listed separately below.
CVE-2023-35839: SOFARPC defaults to using the SOFA Hessian protocol to deserialize received data, while the SOFA Hessian (< 5.12.0) protocol uses a blacklist mechanism to restrict deserialization of potentially dangerous classes for security protection. But there is a gadget chain that can bypass the SOFA Hessian blacklist protection mechanism, and this gadget chain only relies on JDK and does not rely on any third-party components.
CVE-2023-48967: solon

If this is a new vulnerability, have you disclosed it to the affected vendor(s)?
Yes, the related developers patched all these vulnerabilities after we reported our findings.
For example: CVE-2023-39131
vendor response: Thanks for the feedback, we will evaluate whether to replace the native hessian with a

third-party hessian library such as SOFA-Hessian.
reporting status: reported
timelines: May 22, 2023 -- May 28,
2023 patch status: patched.
For example: CVE-2024-23636
vendor response: We are currently working on the fix and should be able to release the patch next week.
sofa-rpc 5.12.0 was released to fix the RCE attack. The CVE issued is CVE-

2024-23636. timelines: Nov 10, 2023 -- Jan 24
reporting status:
reported patch status:
patched.

Will Your Presentation Include a Demo? If Yes, Describe the Demo.

Yes. We plan to demo the process of JDD detecting Java deserialization gadget chains and generating exploitable injection objects for them. This will help attendees understand the JDD workflow more intuitively.

Does Your Company/ Employer Provide a Solution to the Issue Addressed? If Yes, Please Provide Details.

No. We are independent researchers.