Finding Gadgets Like it's 2026

Finding Gadgets Like it's 2026 - Author not stated, Atredis Partners.

- Published: date not stated
- Original: <https://www.atredis.com/blog/2026/3/12/findings-gadgets-like-its-2026>
- Current location: <https://www.atredis.com/blog/2026/3/12/findings-gadgets-like-its-2026/>
- Preserved from: https://www.atredis.com/blog/2026/3/12/findings-gadgets-like-its-2026/ (live) on 2026-08-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

+ All posts

Introduction

Java deserialization vulnerabilities have been of interest to me for nearly a decade. In 2016, my team published a blog post titled "What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability." which kicked off a firestorm of vulnerabilities in high profile enterprise applications. Since that time, the Java ecosystem has taken steps to both eliminate the prevalence of deserialization of untrusted data, and to reduce the number of deserialization gadgets that can be used for malicious purposes.

Tools such as ysoserial have been treated in the penetration testing community as a largely complete catalog of gadgets due to the complexity of discovering new gadget chains. While new chains do surface occasionally, the research process has remained to some degree manual, complex, and unapproachable for those not steeped in the research. The release of new gadgets has slowed significantly, for example, ysoserial has not had new gadgets added for years. A quick review of the research shows very few newly proposed gadget chains. All of the widely used application servers ship in configurations that are rarely vulnerable to any public gadget chains.

As consultants who primarily do platform security assessments, our time is not best spent pursuing novel gadget chains given the complexity of doing so. We wondered if an LLM could automate the task of gadget discovery; exactly the sort of task an LLM should be well suited for. Over the course of just two days, we were able to implement a new methodology for gadget discovery and demonstrate its effectiveness by finding several new chains, including one novel

chain that works against WildFly application server and may affect other popular application servers and frameworks.

Background: Gadget Chains in 60 Seconds

For readers not immersed in the weeds of Java deserialization, a Java deserialization gadget chain is a sequence of method calls that kicks off when methods like `ObjectInputStream.readObject()` processes an attacker-controlled input and ends in a dangerous `sink` method that executes some unintended action that is useful to an attacker. The `sink` methods can trigger things like command execution, file writes, XML External Entity Injection (XXE), or other internal Java methods that have unintended consequences. The deserialization chain relies on Java classes that already exist on the application server classpath and chains them together through various techniques.

There are several key difficulties to identifying useful deserialization chains. A typical application server classpath can contain thousands of classes that implement `Serializable`, all of which are potential entry points that can be sent to the initial `readObject` call. Each `Serializable` class can hold arbitrary subtypes and many times these subtypes are abstract and can therefore map to many different classes. Combine that with dynamic language features such as reflection, and the search space becomes huge. Existing tools like GadgetInspector use static analysis to try and reduce the search space to something that can be manually validated but may miss certain types of chains. This also still requires manual analysis to weed out most of the results which will be false-positives.

The idea behind this project was that an LLM, given the right structure, could use a combination of static analysis, reasoning, and implementation skills to build gadget chains from start to finish. LLMs perform exceedingly well in scenarios where a feedback loop is constructed where the LLM can validate its hypotheses.

Architecture Overview

The tooling architecture was designed through back and forth prompting with the LLM. It has some interesting properties that cater to the strengths of LLMs. The tool can be broken into 5 separate components:

- Call Graph Builder - Tooling to build out a graph of sources, sinks, and connections. This is currently stored in a SQLite Database.
- Graph Query Server - A small HTTP server with REST APIs that expose endpoints which query the call graph database. The LLM agents will use this to search for paths.
- LLM Agents - Claude Code acts as the LLM agent. It queries the graph, discovers potential chains, and evaluates the chains optionally using the “Dynamic Runner” to help with debugging.
- Dynamic Runner - Tools for executing and debugging candidate test harnesses. The LLM agent will not always use these, but they are available for it.
- Evaluation - First evaluate using a test class compiled against the target classpath. Once a chain is confirmed locally, test against a minimal web application deployed in the target environment.

Call Graph Builder

The goal with building the call graph is to create a graph with every possible source and sink, and all possible paths between the two. The reality is that we need to make some compromises because the number of possibilities is too large. For construction of the call graph, we start with the IBM Watson Libraries for Analysis (WALA) Class Hierarchy Analysis (CHA). CHA is technique for constructing call graphs that is fast but imprecise; not all edges in the CHA call graph will actually be traversable in a real deserialization chain.

The CHA produces a huge graph, especially when applied to the classpath of an application server or complex enterprise application. To prune the graph, we restrict it to classes that implement `Serializable` or `Externalizable` and then expand outward three hops via field types and method signatures. Anything outside of that scope will be excluded from the analysis.

The CHA graph is the base. On top of the CHA graph we do a pass over all target classes and inject additional edges to account for dynamic features in Java, this is done by examining class bytecode with the Java ASM library. A second pass with ASM is done to create edges for possible type confusion scenarios. For every `Serializable` class whose type is an interface or abstract class, it records every class that implements it.

Graph Query Server

The Graph Query Server is a simple Python-based FastAPI service that exposes HTTP endpoints for the LLM to query. The following endpoints are exposed:

- `/paths` finds all simple paths from an entry point to a sink, with edge types annotated
- `/type_confusion_targets` returns every concrete type injectable into a given field
- `/decompile` runs CFR on any class or method and returns decompiled source
- `/serialization_constraints` tells you which fields survive serialization, and what subtypes can go into each
- `/predecessors` works backward from a sink to find what calls it
- `/classes_implementing` lists all concrete implementors of an interface

These API endpoints let the LLM efficiently explore the call graph without holding too much of the various callchain structures in the context.

LLM Agent

The agent in this case is Claude Code. Claude Code essentially drives the tooling created in the other phases. To achieve somewhat repeatable results, a markdown file was created with specific instructions on the intended workflow and how to use each of the custom tools (`RUNBOOK.md`).

The runbook instructs the agent to build the call graph and use it to look for deserialization chains using the query server. For candidate chains, the agent builds a Java test harness to validate them; if they fail, it should use tooling that is part of the Dynamic Runner component to debug and try to fix them when possible. This process of identifying candidates and validating them defines a feedback loop that eliminates false positives and keeps the agent on track.

One drawback of this approach is that analysis is non-deterministic. You will get different results each time you run the tool. Multiple prompts may be required to ask it to continue to go deeper before a chain is found.

Dynamic Runner

The Dynamic Runner component is a set of tools in Python to help the agent with confirming candidate chains. It includes tools that help with JDWP-based debugging, DNS servers to monitor for DNS lookups, taint tracking tools, etc... The agent uses these tools at-will and as needed.

Evaluation

The LLM discovers and tests gadget chains autonomously using minimal test classes compiled against the target JAR files. For chains that fail, the LLM iteratively tries to fix the chain using Java debugging to gain detailed information about where and how the chain failed. For chains that pass, they can then be tested against a minimal web application deployed to the target application server. We used the LLM for this again because it often involves some debugging, but this was not part of the runbook workflow.

What We Tested

We initially planned on testing this workflow and tooling against a set of standard benchmarks. In other research, tooling is pointed at the ysoserial JAR file which includes many classes with known gadget chains. The idea is to benchmark by finding how many chains are discovered out of the total number of chains known to exist. Here is where we ran into our first problem: the agent was too smart. Claude Code quickly recognized that the gadgets on the classpath of the benchmark had known gadgets that were implemented in ysoserial and essentially cheated the benchmark.

Rather than try to solve the benchmarking problem, we decided to point it at something real, with no known public gadget chains. We tested against the following:

IBM WebSphere 9.0.5.24 WildFly 39.0.1 Payara (GlassFish fork) 6.2024.6

These are all modern application server stacks that have implemented mitigations for known gadget chains. WildFly and Payara used JDK21 where JPMS should make gadget chains that rely on the `TemplatesImpl` class impossible. WebSphere has a large classpath and runs on its own custom JDK.

What We Found

The tooling discovered many new chains, the most interesting of which may unlock remote code execution gadget chains in other common frameworks and application servers. It is notable that most of the new chains discovered are variations on known existing chains.

In total, the tooling discovered 17 confirmed gadget chains, 6 of which are thought to be novel. Only one chain directly resulted in remote code execution; this is unsurprising given the chosen targets and short timeframe for the project.

CB1-Shaded-RCE: TemplatesImpl Is Not Dead

This is our most interesting gadget chain because it is exploitable for remote code execution in common configurations of WildFly application server and opens up new interesting areas for research.

The 'TemplatesImpl' class is part of the Java JDK and is used in many deserialization gadget chains (CC2, CC4, Spring1, Rome1, Hibernate1, CommonsBeanutils1...). All of these gadget chains are considered dead in JDK versions 9 and newer because of a JDK feature called JPMS (Java Platform Module System). JPMS marks certain JDK classes as internal and does not allow user code to extend them; for example, this class definition would be illegal `class Evil extends AbstractTranslet` because `AbstractTranslet` lives in the non-exported package `com.sun.org.apache.xalan.internal.xsltc.runtime`. JPMS was essentially supposed to stop all `TemplatesImpl` based remote code execution gadgets.

WildFly ships with a JAR called `jakarta.servlet.jsp.jstl-3.0.1-jbossorg-1.jar`. Inside it is a shaded copy of the Xalan XSLTC library, relocated to the package `org.eclipse.tags.shaded.org.apache.xalan`. A `shaded` copy of a JAR file is typically used to ship a certain version of a library without causing conflicts with other versions that might be on the classpath. The shaded copy includes all of the classes needed to re-enable the remote code execution gadget chain (`TemplatesImpl`, `TransletClassLoader`, `AbstractTranslet` etc). Since the classes live in an application JAR and are not part of the JDK, JPMS has no effect. When constructing a deserialization gadget, we can extend the shaded `AbstractTranslet`, the shaded `TransletClassLoader` defines it, and `Class.newInstance()` runs the malicious static initializer. This works on JDK 21.

```
PriorityQueue.readObject()
  BeanComparator.compare("outputProperties")
   PropertyUtils.getProperty(templates, "outputProperties")
    TemplatesImpl.getOutputProperties()        [shaded Eclipse Tags copy]
     newTransformer()
      defineTransletClasses()
       TransletClassLoader.defineClass(evilBytecodes)
        Class.newInstance()
         <clinit> static initializer
          Runtime.getRuntime().exec(cmd)    [RCE]
```

Previous research has exploited other shaded copies of Xalan but as far as we are aware, never for bytecode-loading remote code execution. Checking known blocklists, we have not found the `org.eclipse.tags.shaded` namespace listed.

#	Shaded Namespace	Source	CVE / Issue
1	`org.apache.xalan`	Apache Xalan standalone	jackson-databind #2469
2	`com.sun.org.apache.xalan.internal`	JDK internal	jackson-databind #2704
3	`oadd.org.apache.xalan`	Apache Drill shaded uber-JAR	jackson-databind #2688
4	`com.oracle.wls.shaded.org.apache.xalan`	Oracle WebLogic Server	CVE-2020-35728, jackson-databind #2999
5	`org.docx4j.org.apache.xalan`	docx4j document processing library	jackson-databind #3003

All of these exploited `JNDIConnectionPool` (a JNDI gadget) via Jackson's polymorphic type handling; none exploited `TemplatesImpl` for bytecode-loading RCE, and none bypassed JPMS.

Any server or application that ships a copy of Xalan's XSLTC library outside the `java.xml` module re-enables `TemplatesImpl` attacks.

We confirmed command execution on the latest version of WildFly using a small test application deployed to the server that has a classpath which includes Commons-BeanUtils and Commons-Collections. The dependency of `jakarta.servlet.jstl.api` is autoloaded by WildFly and does not need to be specifically added as a module when deploying the target application.

```
$ docker exec wildfly-deserialize-poc cat /tmp/rce-proof.txt
=== RCE via Shaded TemplatesImpl CB1 ===
Date: Tue Mar 10 12:53:28 AM UTC 2026
User: uid=1000(jboss) gid=1000(jboss) groups=1000(jboss)
Java: openjdk version "21.0.10" 2026-01-20 LTS
```

PriorityQueue-AttributeComparator-JNDI

This chain is interesting in that it appears to be totally novel and demonstrates the LLM's ability to go deep. The call chain required before getting to the JNDI lookup is quite long and uses classes from several different Payara JAR files.

JNDI lookup chains such as this do not yield code execution directly in modern JDK versions but can be a useful primitive.

```
PriorityQueue.readObject()
    heapify()    siftDownUsingComparator()
      AttributeComparator.compare(Attribute a1, Attribute a2)
        a1.getName().compareTo(a2.getName())  // same name    returns 0
        a1.getValue().toString()           // falls through to value comparison
          InjectableJMSContext.toString()
            delegate()
              isInTransaction()    false (no transaction context)
              requestedManager.getContext(id)    null (empty contexts map)
              requestedManager.getContext(ipId, id, metadata, getConnectionFactory(false))
                getConnectionFactory(false)
                  connectionFactory == null  (transient field, always null after deser)
                  new InitialContext().lookup(metadata.getLookup())
                    JNDI INJECTION with attacker-controlled URL!
```

The classes used in the deserialization chain come from the jar files `amx-core.jar` and `gf-jms-injection.jar`. The chain was tested against a live Payara instance with a test application deployed and we received the RMI handshake as expected:

```
$ # TCP listener on attacker host captures RMI handshake:
CONNECTION CAPTURED from ('172.17.0.4', 53378)
Data (7 bytes): b'JRMI\x00\x02K'
Hex: 4a524d4900024b
*** RMI HANDSHAKE DETECTED - JNDI INJECTION CONFIRMED ***
```

This chain appears to have several novel features. First, the use of the `AttributeComparator` class as a `toString` trigger. Historically the class `BadAttributeValueExpException` has been used for this purpose but it no longer works in JDK 18+. The `InjectableJMSContext` as a path to a JNDI lookup also appears to be new.

WildFlyDataSource-JNDI: A Not So Novel Chain

`WildFlyDataSource.readObject()` calls `new InitialContext().lookup(jndiName)` directly, where `jndiName` comes straight from the deserialized stream, a pretty straight forward and short path to JNDI lookup.

```
WildFlyDataSource.readObject()
    new InitialContext().lookup(jndiName)      [JNDI / SSRF]
```

After some research, we discovered that although the LLM believed this chain to be novel, it had previously been documented in a blogpost by Synactiv (https://www.synacktiv.com/publications/finding-gadgets-like-its-2022).

CC4-BeanComparator Bridge (several chains)

The four remaining chains all use the same trick, bridging the `commons-collections4` `TransformingComparator` to `commons-beanutils`'s `BeanComparator`. These chains are minor variations on existing known gadget chains. Ysoserial already includes a gadget that works for `commons-beanutils`, these new gadget chains just give an alternate path to using a similar gadget in cases where the `PriorityQueue` class might be blocked for deserialization. This is not likely to be a common scenario. RCE of these gadgets is blocked in modern JDK's because the necessary paths rely on `TemplatesImpl` (... unless you have a shaded copy!).

The problem: ysoserial's CC2 and CC4 chains need `InvokerTransformer` to be `Serializable`. Starting with CC4 version 4.5.0, the Apache maintainers made `InvokerTransformer` non-serializable. CC2 and CC4 are dead on modern CC4 libraries. But `TransformingComparator` is still serializable and has no safety check.

Two entry points were found, each reaching two sinks:

TreeBag reaches `TemplatesImpl` (RCE) and `JdbcRowSetImpl` (JNDI):

```
TreeBag.readObject()
    TreeMap.put()
     TransformingComparator.compare()
       ConstantTransformer    BeanComparator    sink
```

DualTreeBidiMap reaches the same two sinks:

```
DualTreeBidiMap.readObject()
    createBidiMap()    putAll()    TreeMap.put()
     TransformingComparator.compare()
       ConstantTransformer    BeanComparator    sink
```

In total, four chains were discovered: two RCE, two JNDI injection.

| Property | ysoserial CC2/CC4 | CC4-TreeBag / CC4-DualTreeBidiMap | | | Needs InvokerTransformer serializable | Yes | No | | | Works on CC4 4.5.0+ | No | **Yes** | | | Entry point | PriorityQueue | TreeBag / DualTreeBidiMap | | | Cross-library | No (CC4 only) | **Yes** (CC4 + commons-beanutils) | |

Compared to ysoserial's gadget CB1, CB1 uses `PriorityQueue` directly with `BeanComparator`. This chain wraps `BeanComparator` inside `TransformingComparator` via `ConstantTransformer`, using `TreeBag` as the entry point. The entry point difference matters when deserialization filters block `PriorityQueue` but allow `TreeBag`.

Broader Impact: The Shaded TemplatesImpl Ecosystem

After confirming the CB1-Shaded-RCE chain on WildFly, we wanted to explore if other application servers ship a shaded `TemplatesImpl` outside the `java.xml` platform module.

As described, we confirmed exploitation on WildFly 39.0.1, where the JSTL JAR is auto-loaded for all WAR deployments. WildFly ships with the `Commons-BeanUtils` library which provides the trigger needed to take advantage of the shaded `TemplatesImpl`. Applications do need to declare `Commons-BeanUtils` as a dependency but it ships with the application server.

JBoss EAP (Jakarta EE 10+), GlassFish 7+, Payara 6+ all include a shaded `TemplatesImpl` artifact but none of them include `Commons-BeanUtils`. Some limited exploration was attempted to identify other viable triggers that are shipped with these application servers but none were found during the short duration of this project. While assessing these other application servers, two new shaded Xalan namespaces were discovered `com.oracle.wls.shaded.org.apache.xalan` (Jetty), and `openejb.shade.org.apache.xalan` (TomEE). Applications deployed to these application servers that ship their own libraries that could contain a `TemplatesImpl` trigger class could enable exploitability of the shaded `TemplatesImpl` that the application server ships. Some example libraries that contain such triggers include CommonsCollections 3 and 4, some versions of Jackson, ROME, and Hibernate.

IBM WebSphere is immune, both Liberty and traditional. Neither WebSphere server includes a shaded `TemplatesImpl`. Traditional WebSphere runs on IBM J9, which replaces Oracle's `TemplatesImpl` entirely with `com.ibm.xtq.xslt.jaxp.TemplatesImpl`. That class is not `Serializable`, has no `_bytecodes` field, and loads translet classes by name rather than from raw byte arrays.

We also noted that the standalone `xalan:xalan` artifact on Maven Central has over 112,000 dependents (version 2.7.2 alone). Any application that pulls it in, directly or transitively, has a `TemplatesImpl` outside JPMS. Shaded copies also exist in Apache Drill, Oracle WebLogic, docx4j, and OSGi bundles from Karaf, ServiceMix, and SpringSource.

Wrapping Up

Our goal was to assess whether LLMs could effectively hunt Java deserialization gadgets in modern application server stacks with all of the mitigations that have piled on over the years. We believe this initial proof-of-concept was a success.

One thing that worked surprisingly well was the "test, debug, reason" loop that had the LLM automatically try and debug payloads that were not working. This was a very effective strategy and generalized to using agents in general - whenever possible, building in feedback loops is a great strategy. Another surprising and accidental result was watching the LLM agents make changes to the core tooling when it was not working as effectively as it could; this exposes a new way of thinking about source code as dynamic and malleable while a `Runbook` defines the actual process and goals.

The Claude Code plans and `Runbook` can be found on our GitHub repository at https://github.com/atredispartners/llmchainhunter.