

## 1. Introduction

---

A JVM memory shell (or JVM memory-based backdoor) is a type of malicious trojan that resides entirely in the JVM's memory. When code execution vulnerabilities exist—such as ScriptEngine expression injection, deserialization flaws, etc.—an attacker can leverage reflection to modify certain properties within Java web frameworks and replace them with their own malicious classes, thereby implanting the backdoor directly into the JVM memory space. When the attacker accesses a specific route or supplies specially crafted parameters, these inputs are captured by the malicious method and used to execute arbitrary commands.

The injection process of a JVM memory shell leaves no files on disk, and its network behavior is extremely stealthy. From a defender's perspective, traffic generated by a JVM memory shell appears as a legitimate HTTP request. In fact, it can even be hidden within normal business routes: an attacker can craft an HTTP request that mimics regular business traffic and embed malicious payloads in specific parameters. Once the compromised method (replaced by the attacker) processes these parameters, it executes malicious actions (e.g., command execution) inside the JVM and returns the results via the HTTP response. Additionally, JVM memory shells elegantly bypass situations where strict firewall policies only allow inbound/outbound traffic on ports 80 and 443, rendering traditional reverse shells ineffective.

Over the past eight years, as offensive and defensive techniques around JVM memory shells have evolved, security researchers have gradually uncovered numerous types of memory shells, including:

- Tomcat Filter Memory Shell: Dynamically registers a malicious Filter component via a code execution vulnerability.
- Tomcat Listener Memory Shell: Dynamically registers a malicious Listener component via a code execution vulnerability.
- Spring Controller Memory Shell: Dynamically injects a malicious Controller via a code execution vulnerability.

Currently, the discovery of new JVM memory shell variants relies primarily on manual auditing of Java web framework source code. In the last two years, no significant new memory shell types targeting mainstream frameworks have emerged.

On the defense side, two main approaches are used to combat memory shells:

- Detection of existing memory shells (e.g., java-memshell-scanner)
- Runtime injection interception (e.g., RASP – Runtime Application Self-Protection)

Both methods fundamentally rely on checking whether loaded JVM bytecode classes match known memory shell signatures. This puts defenders in a reactive and passive position. To address this, my collaborator yulate and I decided to systematically extract the core characteristics of memory shells and combine SAST (Static Application Security Testing), Java Agent attach, and AI agents to automate the discovery of new memory shell variants—enabling proactive defense in JVM-based attack scenarios.

We designed a system that:

1. Uses SAST to analyze source code and identify potential classes/methods that could be weaponized as memory shells.
2. Employs Java Agent attach technology to hook these candidate classes/methods at runtime.
3. Applies dynamic JVM memory introspection to determine the optimal reflection path for exploiting each candidate.
4. Feeds the results to an AI agent, which then automatically generates PoC code, closing the loop of the entire discovery pipeline.

Using this system, we expanded the number of known Spring memory shell variants from just 2 to 9 in a very short time—an efficiency far surpassing years of manual research by human analysts.

## 2. Introduction and Evolution of JVM Memory Shell Technology

---

As a fileless attack technique, JVM memory shells are profoundly reshaping the web security landscape. With traditional webshells increasingly constrained by widespread defensive measures—such as file monitoring, Endpoint Detection and Response (EDR), and network traffic inspection—attackers are gradually shifting toward memory shell techniques to achieve more stealthy and persistent control. Memory shells dynamically inject malicious code directly into memory (e.g., Servlets, Filters, Listeners, or by leveraging Java Agents to modify class bytecode at runtime) without writing any files to disk, thereby bypassing conventional detection mechanisms.

In recent years, memory shell technology has rapidly evolved from proof-of-concept demonstrations to mature, real-world deployment. In 2017, Tomcat-based Filter and Servlet memory shells were systematically introduced. After 2020, the availability of offensive tooling (e.g., Behinder, Godzilla) and technical innovations—such as Spring framework-based and WebSocket protocol-based memory shells—further accelerated their adoption. Today, memory shells are deeply integrated with modern web frameworks (e.g., Spring MVC) and cloud-native environments, featuring multi-protocol support and distributed adaptability, which significantly enhances their stealth and persistence.

At the same time, attackers continue to adopt anti-detection techniques—such as blocking Java Agent injection or using ShutdownHooks to resurrect the shell after application restarts—further increasing the threat level of memory shells. Although security vendors have begun researching detection methods based on Runtime Application Self-Protection (RASP), deep learning, and other advanced approaches, the high adversarial nature of memory shells continues to make them a major challenge in cyber offense-defense dynamics.

In summary, JVM memory shells have become an unavoidable threat in web security. Their technical complexity, stealthiness, and persistence pose significant challenges for defenders.

### 2.2 Technical Principles and Evolutionary History of JVM Memory Shells

As a fileless attack technique, JVM memory shells have rapidly evolved from proof-of-concept to mature application since their inception in 2014, profoundly altering the web security offense-defense landscape. Below is a summary of key milestones in their technological evolution and offensive-defensive confrontation:

1. **Technology Germination Stage (2014-2016):**

- **2014:** The concept of a "Java Timer backdoor" was first proposed by Yuan Zhang, enabling fileless backdoor execution through scheduled tasks. However, the technology was still at the proof-of-concept stage with simple implementation.
- **Characteristics:** Attackers discovered that even after deleting JSP files, malicious code could continue running in memory through the Java Timer mechanism, initially revealing the possibility of fileless attacks.

#### 1. Initial Development Stage (2017-2019):

- **2017:** N1nty published "Notes on Debugging Tomcat Source Code - Invisible Shell," systematically proposing the implementation of memory shells based on Tomcat, including Filter-based, Servlet-based, and Listener-based memory shells.
- **Technological Breakthroughs:**
- Attackers began utilizing the dynamic registration feature of the Servlet 3.0 API to create malicious components in memory via reflection.
- The security community started focusing on the detection and defense against memory shells, such as techniques based on the Java Instrumentation mechanism.

#### 1. Rapid Development Period (2020-2022):

- **Tool Proliferation:**
- Tools like Behinder 3.0 Beta7 and Godzilla integrated memory shell capabilities, significantly lowering the barrier for attackers.
- Exploit tools (e.g., Shiro deserialization tools) also began supporting one-click injection of memory shells.
- **Technological Innovations:**
- Framework-based memory shells: Components such as Tomcat connector, Controllers and Interceptors in Spring framework became new injection points.
- Protocol extensions: Attackers utilized WebSocket, gRPC, and other protocols for more covert communication.
- Persistence techniques: Methods like using `Runtime.addShutdownHook()` were employed to achieve persistence across restarts.

#### 1. Period of Technological Proliferation and Slower Growth (2023 to present):

- From 2023 to now, JVM memory shells have become foundational knowledge that both attackers and defenders must master.
- Manual discovery of memory shells has begun to show fatigue; obvious exploitable mechanisms within Java middleware are becoming scarce, leading to a creative drought.

This evolution reflects the dynamic nature of offensive and defensive confrontation in Java web security within the domain of JVM memory shells. From early, simplistic Timer-based backdoors to today's multi-protocol, multi-framework-adapted variants, their stealthiness and persistence have posed severe challenges to traditional defense systems. At the same time, the scarcity of newly discovered memory shell techniques over the past three years also highlights the limitations of manual discovery approaches.

## 2.3 Current Bottlenecks in Memory Shell Discovery and Defense Techniques

Based on summaries of existing technical discussions regarding JVM memory shell discovery, the primary approach to uncovering memory shells still relies heavily on manual code auditing of Java web framework source code. Reviewing the history of memory shell discovery reveals that the earliest identified types, such as Filter-based and Listener-based memory shells, were actually common mechanisms inherent to the Tomcat framework itself. These mechanisms natively support preprocessing incoming traffic at the Java application layer, making their repurposing into malicious backdoors conceptually straightforward.

As similar components were gradually exhausted through research, security researchers appear to have fallen into a "fixed mindset": given the already extensive variety of known memory shell types, the effort required to analyze various web framework source codes for novel memory shells no longer yields proportional returns. Consequently, research into discovering new types of memory shells has significantly declined.

From a defensive perspective, memory shell protection primarily consists of two aspects:

1. **Runtime injection defense**
2. **Detection of resident memory shells**

Runtime injection defense is implemented in many RASP (Runtime Application Self-Protection) products: by tracing the call stack from an initial code execution vulnerability, if downstream operations exhibit behaviors such as invoking the `defineClass` method—indicating dynamic injection of Java bytecode—the system evaluates whether to block the activity based on characteristics of the injected bytecode (primarily the class type).

Meanwhile, detection of resident memory shells involves scanning JVM memory to check whether corresponding class files exist on disk for loaded objects. If no matching class file is found, it suggests the class was dynamically generated at runtime; such classes can then be dumped and analyzed to determine if they match known memory shell patterns. For a practical implementation, refer to: <https://github.com/c0ny1/java-memshell-scanner>.

It is evident that both injection prevention and post-infection detection of memory shells heavily depend on known signatures or characteristics of existing memory shells. In other words, attacks leveraging previously undisclosed, novel memory shell techniques are not covered by current defense mechanisms.

## 3. Technical Overview of an Automated Memory Shell Discovery Project

### 3.1 Overall Architecture Design Philosophy

Our team's work at the company primarily focuses on building RASP (Runtime Application Self-Protection) capabilities. We have developed our own offensive-and-defensive benchmark environment to evaluate RASP effectiveness, and our team maintains a strong enthusiasm for researching emerging attack techniques. Therefore, we are not satisfied with the current state of JVM memory shell offense-and-defense technologies.

Personally, I frequently attempt to discover previously unknown types of memory shells. However, this process has consumed so much time—mostly spent reading source code from Java EE frameworks such as Spring and Tomcat. When I successfully identified and implemented a new type of memory shell based on FlashMap through manual code auditing, I realized it was time to systematize my discovery methodology and automate the process to efficiently uncover potential memory shell vectors across projects at scale.

First, after analyzing malicious characteristics from multiple known JVM memory shells, we distilled common code-level patterns associated with such threats (see Section 3.2.2 for details). Leveraging these patterns, we applied Static Application Security Testing (SAST) to generate an initial candidate list. However, this list was large and noisy—most entries were irrelevant and could not serve as viable memory shell candidates. Nevertheless, it did include several memory shell types already discovered by security researchers, confirming that our initial approach was fundamentally sound.

The next challenge was how to further narrow down this candidate list and reduce SAST noise. We reasoned that since JVM memory shells are injected at runtime, we should address the problem during JVM execution. Thus, we adopted Java Agent-based dynamic instrumentation—inspired by Interactive Application Security Testing (IAST)—to hook all SAST-identified candidates at runtime. Whenever a hooked method was triggered, relevant data was recorded into a lightweight database.

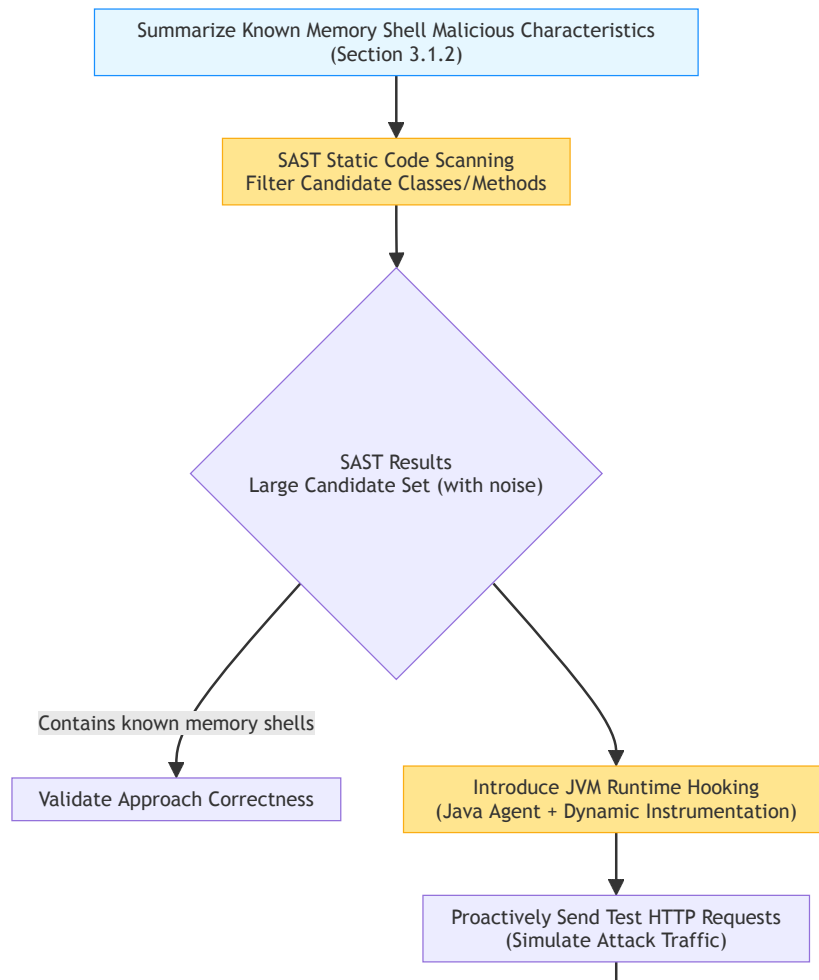
Similar to active IAST, after instrumenting all targets, we proactively sent a few test HTTP requests to the running web application. The entries recorded in the database afterward represented highly suspicious, potentially exploitable memory shell candidates.

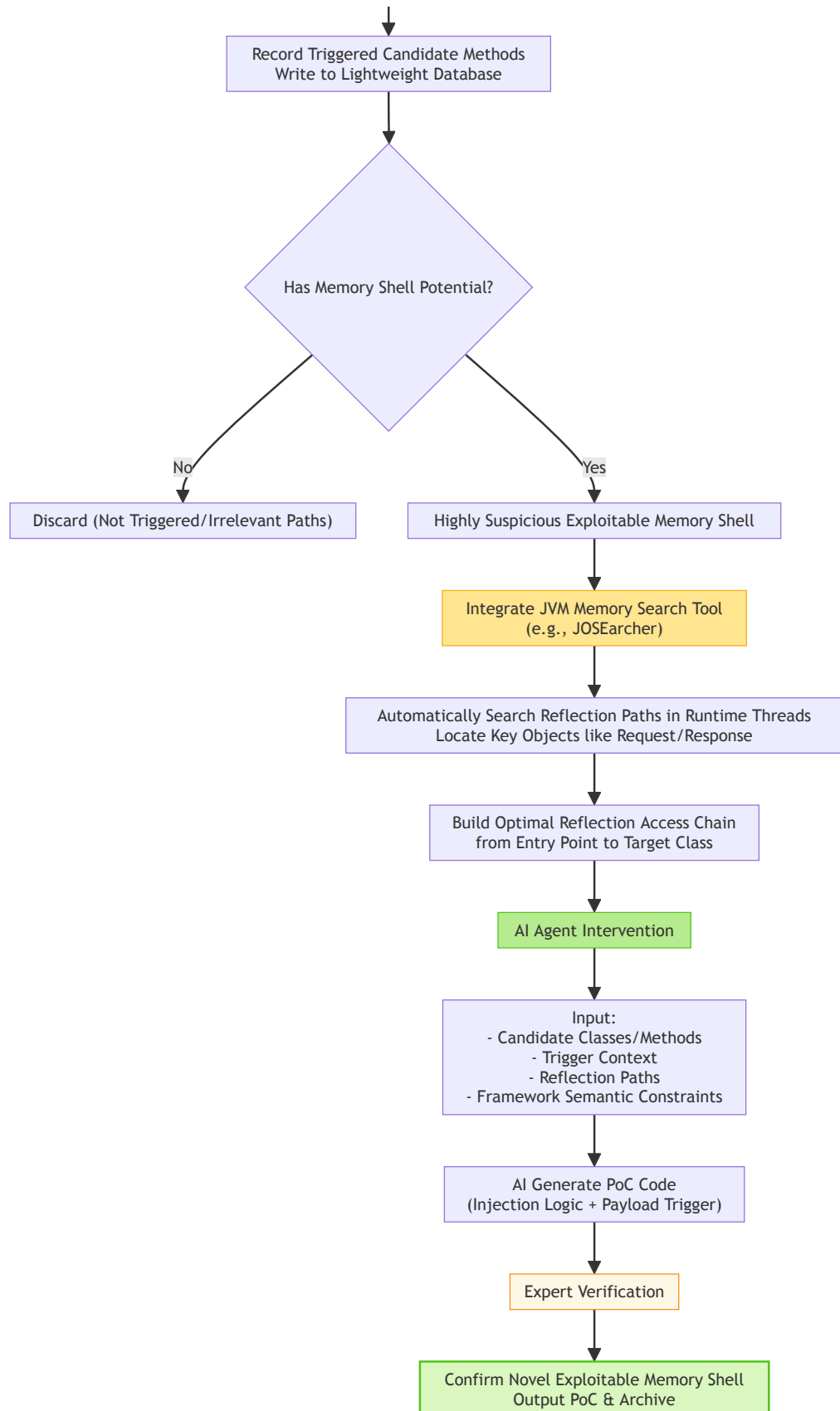
At this point, readers might still have a critical question: existing JVM memory shells typically require their malicious methods to accept a `Request` object as a parameter, enabling them to recognize and process attacker-controlled requests. However, the combined SAST+IAST results described above do not inherently satisfy this condition—so why claim they are "highly likely" exploitable?

This concern was resolved thanks to advances made around 2021: tools like the JVM object searcher (e.g., <https://github.com/c0ny1/java-object-searcher>) emerged. Around the same time, security researchers, while solving Java code execution output (i.e., "echo") challenges, discovered a universal technique using JVM memory introspection to reflectively retrieve the Tomcat `Request` object from thread contexts. This breakthrough directly addressed our earlier limitation.

With these two steps—SAST filtering followed by runtime validation via dynamic instrumentation—we identified a refined set of high-confidence JVM memory shell candidates. The next phase involved PoC development and verification. We embedded the JVM object searcher directly into our Java Agent, enabling it to automatically discover the optimal reflective path within JVM threads to locate and replace target classes. Furthermore, we integrated AI-assisted coding to generate PoC code, which is then verified by security experts.

The final output of this pipeline is a set of verified, exploitable novel Java memory shells.





Our end-to-end automated workflow demo: <https://github.com/L1gh73r/java-memory-shells-Auto-Search>

## 3.2 Part 1: SAST Static Analysis Engine

### 3.2.1 Overall Design Approach of the SAST Engine

In the overall workflow, the SAST (Static Application Security Testing) engine plays a preliminary role: "filtering suspicious candidate points from a vast code space." Its design goal is not to directly provide a binary judgment of "whether or not it is a memory shell," but rather to produce a high-recall, structured candidate set that serves as input for subsequent JVM runtime hooking and AI-assisted PoC generation.

Specifically, the system first uniformly decompiles the specified target JAR files to construct a comprehensive source-code view encompassing both application logic and third-party dependencies. It then builds a Java semantic database using CodeQL in *source mode* (i.e., building the CodeQL database solely from source code), which uniformly models classes, methods, fields, inheritance hierarchies, and call relationships.

Based on this semantic database, we authored multiple custom CodeQL queries (e.g., `prescan1.q1`, `prescan2.q1`) that perform static filtering of potentially weaponizable classes and methods along several key dimensions:

- Extensibility (e.g., whether a class can be subclassed),
- Field ownership relationships (e.g., long-lived objects holding references to candidate classes),
- Method call graphs, and
- Presence of request-related parameter types.

Unlike traditional scanning approaches that rely solely on string matching or AST (Abstract Syntax Tree) pattern recognition, our design leverages CodeQL’s semantic abstraction capabilities to naturally express higher-order patterns—such as “an extensible type that is held by a long-lived component and invoked along a request-processing path.” This approach maintains interpretability while effectively enabling candidate convergence at scale across complex Java Web applications.

3.2.2 Criteria for Filtering Potential Memory Shell Classes

After summarizing typical JVM memory shell implementations (e.g., Filter-, Listener-, and Controller-based types) in Chapter 2, we did not attempt to match specific known samples. Instead, we abstracted a set of structural criteria strongly correlated with “being exploitable as a memory shell” and formally encoded them using CodeQL.

At the **type level**, our candidate filtering conditions include:

1. The type must be `public`, and neither `final` nor `abstract`.
2. It must not be an anonymous inner class.
3. Types whose names contain noise-inducing terms like `Exception` or `Object` are excluded.
4. The package scope is restricted to Spring-related namespaces such as `org.springframework%`, `org.spring%`, and `org.springframework.boot%`.
5. Standard library packages (e.g., `java.*`, `javax.*`, `sun.*`) are explicitly excluded.

This layer of filtering captures the characteristic that effective memory shells typically reside at the boundary between application logic and framework infrastructure—exhibiting extensibility and reusability.

At the **call and data structure level**, we further applied predicates such as `hasFieldAndUsesIt` to require that candidate types:

1. Are persistently held as non-`final` fields in other classes, where the static type of the field is assignment-compatible with the candidate class (either directly or via inheritance);
2. Have those fields initialized via constructors or typical factory/container methods such as `getBean`, `getInstance`, or `create`;
3. Are actually invoked through method calls on those fields.

These constraints collectively model potential injection points that are “persistently referenced by containers or business components and schedulable at runtime.”

Building upon this foundation, an enhanced query (`prescan2.q1`) adds an additional requirement: the parameter types of the invoked methods must have simple names or fully qualified names containing keywords like `request` or `response`. This focuses the analysis on logic that directly interacts with HTTP requests and responses—aligning with the behavioral pattern of memory shells, which typically “receive malicious requests, parse commands, and return output.”

Through this multi-dimensional, signature-free filtering pipeline, we can automatically identify a set of “potential memory shell classes” in real-world projects that are highly suspicious both structurally and semantically—without relying on any known attack signatures.

3.2.3 Output Results Overview

We abstract the results of SAST into four aspects:

- Classes that can serve as Java memory shells — `fieldTypeName`
- Malicious methods within Java memory shell classes — `calledMethod`
- Classes that instantiate Java memory shell classes as fields — `ownerClass`
- Field names used when Java memory shells are instantiated within other classes — `fieldName`

After filtering with CodeQL, we obtained a set of candidate targets that meet the criteria, as shown in the table below:

| ownerClass                                                               | fieldName            | fieldTypeName                                                                  | calledMethod  |
|--------------------------------------------------------------------------|----------------------|--------------------------------------------------------------------------------|---------------|
| org.springframework.cache.interceptor.CacheAspectSupport                 | keyGenerator         | org.springframework.util.function.SingletonSupplier                            | obtain        |
| org.springframework.jmx.export.MBeanExporter                             | namingStrategy       | org.springframework.boot.autoconfigure.jmx.ParentAwareNamingStrategy           | getObjectNam  |
| org.springframework.jmx.export.MBeanExporter                             | assembler            | org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler    | getMBeanInfo  |
| org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaConfiguration | poolMetadataProvider | org.springframework.boot.jdbc.metadata.CompositeDataSourcePoolMetadataProvider | getDataSource |
| org.springframework.transaction.jta.JtaTransactionManager                | jndiTemplate         | org.springframework.jndi.JndiTemplate                                          | getEnvironmer |
| ...                                                                      | ...                  | ...                                                                            | ...           |

## 3.3 Part 2: Using Java Instrumentation Technology to Identify Instrumentation Points Triggered at Runtime

### 3.3.1 Purpose of Using JVM Runtime Hooking Technique

Relying solely on SAST results can identify a large number of classes and methods that appear suspicious at the code-structure level, but these candidates often include substantial noise—components never actually triggered in real-world business workflows. To further narrow down from "suspicious" to "highly exploitable," we must answer a critical question:

**Which candidate points are actually invoked during real application execution?**

To address this, we introduce JVM runtime hooking after static analysis, leveraging a Java Agent to perform systematic dynamic observation of these candidates.

Specifically, during the pre-analysis phase, the target Spring application is launched with parameters such as `-javaagent:DynamicTracer-1.0-SNAPSHOT.jar` and `-Dctfagent.csv.path=/app/codeql-output/Sast_result.csv`, which inject the SAST-generated CSV configuration into DynamicTracer.

As classes are loaded or retransformed, the Agent instruments the specified methods. Without altering business logic semantics, it captures concise summaries of method arguments, call stacks, thread context, and optionally object reference relationships for every invocation. This data is then written as structured JSON logs into lightweight storage for subsequent inspection.

### 3.3.2 Java Agent Design Approach

The Java Agent in this system uses `com.dynamictracer.Agent` as its entry point and complies with the Java Instrumentation specification, supporting both static attachment (`premain`) and dynamic attachment (`agentmain`) modes.

In JVM startup scenarios, `premain` first appends its own JAR to the Bootstrap ClassLoader's search path via `addJarToBootstrap`, ensuring that the internal hooking logic and JOSEarcher components remain visible even in complex environments involving multiple classloader isolation or containerized deployments. It then calls `setGlobalInstrumentation` to cache a global `Instrumentation` reference and registers a single `DefaultTransformer` instance as a `ClassFileTransformer` to centrally handle all subsequent class loading and retransformation events.

When dynamically attached to a running JVM, `agentmain` performs the same initialization steps and additionally invokes `retransformClasses` on critical web components—such as `javax.servlet.http.HttpServlet` and `org.apache.catalina.core.ApplicationFilterChain`—to instrument existing classes without requiring a restart.

Internally, `DefaultTransformer` adopts a modular rule-engine design:

- On one hand, it reads the CSV file specified by the `ctfagent.csv.path` system property via `loadCsvHookRules`. Each line is parsed by `CsvParser` into a `MethodHookEntry`, which is then passed to `DynamicMethodHookRule` for precise matching and instrumentation of target classes and methods.
- On the other hand, it initializes `BreakpointRule` through `loadBreakpointRules`, loading breakpoint templates (e.g., `JOSEARCHER_SEARCH`, `PARAMETER_DUMP`) from a JSON configuration file (e.g., `breakpoint-config.json`). Custom logic is inserted before or after matched methods to precisely trigger memory searches, parameter dumps, or stack traces.

`DynamicMethodHookRule` is implemented with strong emphasis on compatibility and stability:

- Namespace Safeguards:** A class-name prefix blacklist excludes sensitive namespaces—such as JDK core classes, Sun internal implementations, Spring property binding utilities, and the Agent itself—to prevent dangerous modifications to JVM infrastructure or framework internals.
  - Method-Level Exclusions:** Native, abstract, final methods, basic `Object` methods, and constructors are skipped. A `ThreadLocal<Boolean>` guard prevents recursive reentrancy of hook logic.
  - Instrumentation Logic:** For each matched method, monitoring code is injected at the method entry point to:
    - Capture argument information (distinguishing primitives from reference types),
    - Record the top few layers of the call stack,
    - When reference tracking is enabled, use `ObjectReferenceTracker` to trace in-heap object references starting from the `this` object.
1. All collected data is asynchronously logged via `HighPerformanceSQLiteLogger.logHookWithReferences`.

Additionally, the rule detects cases where the method is actually declared in a superclass. In such cases, the superclass is added to the `parentClassesToHook` set, and the transformer ensures these parent classes are also instrumented during subsequent class loading—naturally enabling unified instrumentation across inheritance hierarchies.

### 3.3.3 Output Results Overview

The output from the runtime hooking phase primarily consists of two types of data:

- High-granularity observation logs of candidate method invocations
- In-heap object search results triggered by breakpoints

To better align these raw logs with the context of memory shell analysis, a pre-analysis script invokes `filter_logs.py` after the agent finishes execution. This script uses the first occurrence of an `ApplicationReadyEvent` log entry as a temporal boundary and retains only the hook records generated after the application has fully started and HTTP requests have been sent.

This filtered result is then loaded by the workflow module via an API for display in the web console and for decision-making by the AI agent—for example, to identify which candidate classes are frequently triggered under test traffic, and to analyze typical parameter patterns and call stack structures.

```

{
  "ID": 25940,
  "TIMESTAMP": 1764925724763,
  "DATE_TIME": "2025-12-05 09:08:44.763",
  "CLASS_NAME": "org.springframework.context.event.SimpleApplicationEventMulticaster",
  "METHOD_NAME": "multicastEvent",
  "METHOD_SIGNATURE": "
(Lorg/springframework/context/ApplicationEvent;Lorg/springframework/core/ResolvableType;)V",
  "OWNER_CLASS": "org.springframework.boot.context.config.DelegatingApplicationListener",
  "FIELD_NAME": "multicaster",
  "PARAMETERS": "Parameter 1:
org.springframework.boot.availability.AvailabilityChangeEvent[source=org.springframework.boot.web.servlet.context.A
nnotationConfigServletWebServerApplicationContext@724af044, started on Fri Dec 05 09:08:42 UTC 2025], Parameter 2:
null, ",
  "STACK_TRACE":
"java.lang.Thread.getStackTrace(Thread.java:1564)\norg.springframework.context.event.SimpleApplicationEventMulticas
ter.multicastEvent(SimpleApplicationEventMulticaster.java)\norg.springframework.context.support.AbstractApplication
Context.publishEvent(AbstractApplicationContext.java:421)\norg.springframework.context.support.AbstractApplicationC
ontext.publishEvent(AbstractApplicationContext.java:378)\norg.springframework.boot.availability.AvailabilityChangeE
vent.publish(AvailabilityChangeEvent.java:92)\norg.springframework.boot.availability.AvailabilityChangeEvent.publis
h(AvailabilityChangeEvent.java:78)\norg.springframework.boot.context.event.EventPublishingRunListener.ready(EventPu
blishingRunListener.java:115)\norg.springframework.boot.SpringApplicationRunListeners.lambda$ready$6(SpringApplicat
ionRunListeners.java:82)\njava.util.ArrayList.forEach(ArrayList.java:1259)\norg.springframework.boot.SpringApplicat
ionRunListeners.doWithListeners(SpringApplicationRunListeners.java:120)\n",
  "REFERENCE_INFO": "",
  "THREAD_NAME": "main",
  "THREAD_ID": 1,
  "CREATED_AT": "2025-12-05 09:08:44.763"
}
...

```

### 3.4 Part 3: Brief Overview of the JVM Runtime Memory Search Module

After completing SAST-based candidate filtering and JVM runtime hooking, the system still faces a critical question: even if a candidate method is triggered under the tested HTTP request traffic, does it truly possess sufficient memory access capabilities to "evolve" into an exploitable memory shell?

To address this, we integrated a JVM in-memory object search tool (such as Java-Object-Searcher / JOSEarcher) into the Java Agent and tightly coupled it with the breakpoint mechanism. This allows on-demand exploration of the heap object topology at specific execution moments.

The core objective of this module is to automatically discover, from the current thread and its calling context, the optimal reflection chains leading to HTTP request/response objects, core framework components, and instances of candidate classes—thereby providing directly usable access paths for subsequent PoC generation.

#### 3.4.1 Dynamic Analysis of JVM Memory Structure

Dynamic memory structure analysis is built upon JOSEarcher searches triggered by breakpoints. By setting breakpoints of type `JOSEARCHER_SEARCH` at critical event dispatch points—such as `SimpleApplicationEventMulticaster.multicastEvent`—the system can perform breadth-first search (BFS) over the object graph reachable from the current thread during key time windows, such as after application startup completion, during event broadcasting, or while handling HTTP requests.

The search configuration includes a set of "keywords" (e.g., `Request`, `Session`) and "blacklisted types" (e.g., `java.io.File`). Keywords help locate objects related to HTTP request processing or high-risk resource access, while blacklisted types are used to filter out noisy or potentially dangerous system resources during traversal.

The concrete execution flow is as follows: when a breakpoint is hit, a `BreakpointRule` automatically constructs a `SearchRequestByBFS` instance. Using the current thread as the root node, it performs a bounded depth-first or breadth-first traversal of the in-heap object graph. Paths leading to objects matching the keyword criteria are serialized into JSON report files and saved under the `jos_result` directory.

These reports not only record the types and field structures of matched objects but also preserve the full access path from the thread to each target object—including intermediate collections, Maps, framework contexts, etc.—providing high-quality "memory-side evidence" for subsequent reflection chain inference and PoC construction.

Compared to approaches relying solely on static call graphs, this runtime-based memory structure analysis more accurately reflects the actual behavior of the application under specific configurations and incoming requests, effectively mitigating the uncertainty introduced by framework-level reflection, dynamic proxies, and other mechanisms that challenge static analysis.

#### 3.4.2 Automatic Discovery of Java Reflection Chains

After obtaining the memory search reports, the system needs to shift its perspective from "in-heap object locations" back to "programmable invocation paths," i.e., automatically deducing a reflective chain from an attacker-controlled entry point (such as a method in a Controller/Filter) to the target object or class. Our program abstracts this step as the "automatic detection of reflective chains": given thread stacks, object graphs, and candidate class information, it searches for the shortest or least costly access path to support subsequent Java memory shell injection or exploitation.

In terms of implementation, the system first combines the call stack information from hook logs with JOSEarcher outputs to locate currently executing candidate methods along with associated key nodes such as Request/Response objects and context objects. Following this, based on field references between objects, containment relationships within collections, and framework conventions (e.g., typical chains from ThreadLocal to Request in Servlet containers), it constructs a "reflectively accessible graph." On this graph, using bounded breadth-first search (BFS) or heuristic search algorithms, it can automatically find a path from the current execution point, traversing several layers of `getField()`, `get()`, `getAttribute()` and other reflective operations, to reach the target object. This path is then transformed into a semi-structured description for direct consumption by the AI Agent.

The output of the automatic detection module for reflective chains can be understood as a procedural blueprint answering the question: "Given a candidate class and method, how do we acquire its instance and alter its behavior purely in-memory or through reflection?" This provides a structured approach for understanding and manipulating instances at runtime for further exploitation or analysis, ultimately supporting the generation of PoC code for JVM memory shells.

### 3.4.3 Output Results Overview

```
{ "tag": "bp_search_1", "field_name": "global", "class": "org.apache.coyote.RequestGroupInfo", "depth": 7, "chain": "TargetObject = {java.lang.Thread} \n ---> group = {java.lang.ThreadGroup} \n ---> threads = {class [Ljava.lang.Thread;}\n \n ---> [15] = {java.lang.Thread} \n ---> target = {org.apache.tomcat.util.net.NioEndpoint$Poller} \n ---> this$0 = {org.apache.tomcat.util.net.NioEndpoint} \n ---> handler = {org.apache.coyote.AbstractProtocol$ConnectionHandler} \n ---> global = {org.apache.coyote.RequestGroupInfo}" }
{ "tag": "bp_search_1", "field_name": "global", "class": "org.apache.coyote.RequestGroupInfo", "depth": 7, "chain": "TargetObject = {java.lang.Thread} \n ---> group = {java.lang.ThreadGroup} \n ---> threads = {class [Ljava.lang.Thread;}\n \n ---> [15] = {java.lang.Thread} \n ---> target = {org.apache.tomcat.util.net.NioEndpoint$Poller} \n ---> this$0 = {org.apache.tomcat.util.net.NioEndpoint} \n ---> handler = {org.apache.coyote.AbstractProtocol$ConnectionHandler} \n ---> global = {org.apache.coyote.RequestGroupInfo}" }
{ "tag": "bp_search_1", "field_name": "global", "class": "org.apache.coyote.RequestGroupInfo", "depth": 7, "chain": "TargetObject = {java.lang.Thread} \n ---> group = {java.lang.ThreadGroup} \n ---> threads = {class [Ljava.lang.Thread;}\n \n ---> [15] = {java.lang.Thread} \n ---> target = {org.apache.tomcat.util.net.NioEndpoint$Poller} \n ---> this$0 = {org.apache.tomcat.util.net.NioEndpoint} \n ---> handler = {org.apache.coyote.AbstractProtocol$ConnectionHandler} \n ---> global = {org.apache.coyote.RequestGroupInfo}" }
...
```

## 3.5 Part 4: AI Agent Generates PoC

After completing the pipeline of "static candidate filtering + dynamic trigger observation + memory structure and reflective chain inference," the system has already acquired most of the critical information required to construct new JVM memory shells:

- The location of candidate classes/methods
- Their role within the request processing pipeline
- Available in-memory access paths

The goal of the AI-powered PoC generation module is to automate the final step—from "attack concept" to "executable exploit code." Based on the aforementioned multi-modal analysis results, it automatically generates PoC code for each high-confidence candidate, which is then verified by security experts and archived.

### 3.5.1 AI Agent Workflow

The AI Agent's workflow can be broadly divided into four phases:

1. **Data Aggregation**
2. **Strategy Planning**
3. **Exploit Script Generation**
4. **Result Attribution**

First, the Agent extracts key information from multiple sources—including SAST results (CSV), runtime hook logs (`analysis_result.json`), memory search reports (JOSEarcher outputs), and reflective chain descriptions—to construct a unified "candidate memory shell sample description." This description includes semantic labels such as class/method signatures, trigger paths, request parameter patterns, and in-heap object access paths.

Next, leveraging a precompiled knowledge base of JVM memory shell techniques and offensive/defensive experience, the Agent formulates an appropriate exploitation strategy for each candidate sample. Examples include: "injecting a memory shell by dynamically registering a Controller within a Spring event listener" or "weaving malicious logic into a specific Filter chain."

Once the strategy is determined, the Agent uses the sample description and a strategy template as contextual input to prompt a large language model to generate the corresponding Proof-of-Concept (PoC) code and exploitation steps. This includes concrete reflective call chains, class loading/replacement mechanisms, and the required payload trigger parameters.

Finally, the generated PoC code is submitted to security experts for manual verification. Experts evaluate each PoC's feasibility, test its functionality, and provide feedback. These expert evaluations form a feedback loop of actionable experience data, enabling continuous optimization of future sample selection and strategy planning.

### 3.5.2 How AI Generates PoC Code

In the concrete PoC generation phase, the AI Agent must not only integrate the results from prior memory analysis and reflective chain inference but also adhere to the semantic constraints imposed by the target framework and middleware. For instance, for a candidate identified as a Spring Controller-type memory shell, PoC generation must account for the ApplicationContext lifecycle, RequestMapping parsing rules, and security configurations (e.g., CSRF protection, authentication/authorization mechanisms). For Listener/Filter-type candidates, it must respect the component registration process and filter chain execution order defined by the Servlet container.

In our engineering implementation, the program provides the AI Agent with decompiled source code alongside reflective path information, instructing it to generate PoC code capable of reproducing the exploitation scenario in the given runtime environment. This includes:

- Implementation of the malicious class
- The injection procedure
- Construction of the triggering HTTP request

The generated PoC can either be a standalone Java program or a patch-style extension built upon an existing exploit chain (e.g., appended to a deserialization entry point).

Because earlier modules have significantly narrowed the candidate space and supplied rich, structured context, the AI Agent no longer needs to "blindly write" exploit code in a vast search space. Instead, it performs constrained, conditional code generation under well-defined boundaries—dramatically improving both the success rate and interpretability of the resulting PoC code.

### 3.5.3 Output Results Overview

```
{
  "class_name": "SimpleApplicationEventMulticaster",
  "code_length": 1485,
  "extend_class": "org.springframework.context.event.SimpleApplicationEventMulticaster",
  "feasibility": 88,
  "id": 1,
  "info": "Mount point: SimpleApplicationEventMulticaster.multicastEvent(ApplicationEvent, ResolvableType). The class is non-final, the method is a public instance method, non-final/non-static, with signature: (ApplicationEvent, @Nullable ResolvableType)void, which can be safely overridden in a subclass. The PoC only overrides this overloaded method, preserving @Override and @Nullable annotations. The method body first prints an ECHO, then calls super.multicastEvent(event, eventType) to maintain the original broadcast logic. The ECHO follows the standard format: SimpleApplicationEventMulticaster_multicastEvent_success. The class constructor requires no special handling. It is recommended to use Agent redefine or replace the original instance with a subclass during Bean initialization in actual exploitation. This PoC has no RCE capability; it is print-only, compatible with JDK 8 and the given Spring version, with high theoretical feasibility.",
  "memory_shell_code": "package com.pwn;\n\nimport java.util.concurrent.Executor;\n\nimport org.apache.commons.logging.Log;\nimport org.apache.commons.logging.LogFactory;\n\nimport org.springframework.beans.factory.BeanFactory;\nimport org.springframework.context.ApplicationEvent;\nimport org.springframework.context.ApplicationListener;\nimport org.springframework.context.PayloadApplicationEvent;\nimport org.springframework.context.event.SimpleApplicationEventMulticaster;\nimport org.springframework.core.ResolvableType;\nimport org.springframework.lang.Nullable;\nimport org.springframework.util.ErrorHandler;\n\n\n/**\n * Printing memory shell based on SimpleApplicationEventMulticaster.\n *\n * Only overrides multicastEvent(ApplicationEvent, ResolvableType) to add\n * a console echo and then delegates to the original logic via super.\n */\npublic class PwnSimpleApplicationEventMulticaster extends SimpleApplicationEventMulticaster {\n\n    /**\n     * Keep original constructors for potential BeanFactory-based wiring.\n     */\n    public PwnSimpleApplicationEventMulticaster() {\n        super();\n    }\n\n    public PwnSimpleApplicationEventMulticaster(BeanFactory beanFactory) {\n        super(beanFactory);\n    }\n\n    @Override\n    public void multicastEvent(final ApplicationEvent event, @Nullable ResolvableType eventType) {\n        // ECHO\n        System.out.println(\"SimpleApplicationEventMulticaster_multicastEvent_success\");\n        // Preserve original behavior\n        super.multicastEvent(event, eventType);\n    }\n}\n",
  "override_method": "multicastEvent",
  "package_name": "org.springframework.context.event",
```

```
"reflection_code": "org.springframework.web.context.request.ServletRequestAttributes attrs =
(org.springframework.web.context.request.ServletRequestAttributes)
org.springframework.web.context.request.RequestContextHolder.currentRequestAttributes();\njavax.servlet.http.HttpSe
rvletRequest request = attrs.getRequest();\norg.springframework.web.context.WebApplicationContext wac =
org.springframework.web.servlet.support.RequestContextUtils.findWebApplicationContext(request);\nif (wac == null)
{\n    System.out.println(\"[PwnSimpleApplicationEventMulticaster] WebApplicationContext is null, abort
injection\");\n} else {\n    org.springframework.context.support.AbstractApplicationContext context =
(org.springframework.context.support.AbstractApplicationContext) wac;\n    java.lang.reflect.Field field =
org.springframework.context.support.AbstractApplicationContext.class.getDeclaredField(\"applicationEventMulticaster
\");\n    field.setAccessible(true);\n    Object current = field.get(context);\n    if (current == null) {\n    System.out.println(\"[PwnSimpleApplicationEventMulticaster] current applicationEventMulticaster is null\");\n    }
else {\n    System.out.println(\"[PwnSimpleApplicationEventMulticaster] current type: \" +
current.getClass().getName());\n    if (current instanceof com.pwn.PwnSimpleApplicationEventMulticaster) {\n
    System.out.println(\"[PwnSimpleApplicationEventMulticaster] already_replaced\");\n    } else {\n
    com.pwn.PwnSimpleApplicationEventMulticaster newMulticaster;\n    if (current instanceof
org.springframework.context.event.AbstractApplicationEventMulticaster) {\n    // Attempt to copy
BeanFactory\n    org.springframework.context.event.AbstractApplicationEventMulticaster old =
(org.springframework.context.event.AbstractApplicationEventMulticaster) current;\n
    java.lang.reflect.Field bfField =
org.springframework.context.event.AbstractApplicationEventMulticaster.class.getDeclaredField(\"beanFactory\");\n
    bfField.setAccessible(true);\n    Object bf = bfField.get(old);\n    if (bf
instanceof org.springframework.beans.factory.BeanFactory) {\n    newMulticaster = new
com.pwn.PwnSimpleApplicationEventMulticaster((org.springframework.beans.factory.BeanFactory) bf);\n
} else {\n    newMulticaster = new com.pwn.PwnSimpleApplicationEventMulticaster();\n
    if (bf instanceof org.springframework.beans.factory.BeanFactory) {\n
    newMulticaster.setBeanFactory((org.springframework.beans.factory.BeanFactory) bf);\n    }\n
    } else {\n    newMulticaster = new
com.pwn.PwnSimpleApplicationEventMulticaster();\n    }\n    field.set(context, newMulticaster);\n
    System.out.println(\"[PwnSimpleApplicationEventMulticaster] replace_success: \" +
newMulticaster.getClass().getName());\n    }\n    }\n}
...

```

## 4.Introduction to Practical Results and Future Outlook on Memory Shell Technology

Using the aforementioned program, we discovered and verified some new types of Spring memory shells in a short period of time, and have already implemented defensive capabilities against them in our company's security products. The complete list of these novel memory shells is as follows (We have placed the testable PoC templates for the novel JVM memory shells into the project - <https://github.com/L1gh73r/java-memory-shells-Auto-Search/tree/main/pocs>):

| flashmap      | SpringMVC FlashMap Memshell      |
|---------------|----------------------------------|
| multicaster   | SpringMVC Multicaster Memshell   |
| multipart     | SpringMVC Multipart Memshell     |
| corsprocessor | SpringMVC CorsProcessor Memshell |
| viewresolver  | SpringMVC ViewResolver Memshell  |
| acceptheader  | SpringMVC AcceptHeader Memshell  |
| resource      | SpringMVC Resource Memshell      |

As Java offensive and defensive techniques continue to evolve and mature, an increasing number of Java code execution attack vectors have been uncovered, while various Java security defense mechanisms are also rapidly catching up—collectively creating a vast stage for memory shell offense and defense scenarios. We hope that memory shell technology will move beyond the bottleneck of manual research and exert deeper influence on both offensive and defensive fronts:

1. With our approach, any Java security researcher can reproduce our methodology to identify potential memory shell risks within their own Java web applications.
2. Given the rapid iteration of Java web frameworks, our methodology can be applied to every version of open-source components to quickly uncover latent memory shell vulnerabilities.
3. Beyond Spring and Tomcat, our approach can also be extended to assess potential memory shell risks in a wider range of commonly used Java components.