Finding Java Deserialization Gadgets with CodeQL

Automating Security Analysis for Gadget Chain Discovery

isomo[1]

2025-11-22

[1] github/jiahaoxiang2000

Background

What is Java Deserialization?

Java Serialization converts objects to byte streams for storage or transmission

What is Java Deserialization?

Java Serialization converts objects to byte streams for storage or transmission

```java
ObjectOutputStream out = new ObjectOutputStream(fileOut);
out.writeObject(myObject);  // Serialize
```

What is Java Deserialization?

Java Serialization converts objects to byte streams for storage or transmission

```java
ObjectOutputStream out = new ObjectOutputStream(fileOut);
out.writeObject(myObject);  // Serialize
```

Java Deserialization reconstructs objects from byte streams

```java
ObjectInputStream in = new ObjectInputStream(fileIn);
MyClass obj = (MyClass) in.readObject();  // Deserialize
```

What is Java Deserialization?

Java Serialization converts objects to byte streams for storage or transmission

```
ObjectOutputStream out = new ObjectOutputStream(fileOut);
out.writeObject(myObject);  // Serialize
```

Java Deserialization reconstructs objects from byte streams

```
ObjectInputStream in = new ObjectInputStream(fileIn);
MyClass obj = (MyClass) in.readObject();  // Deserialize
```

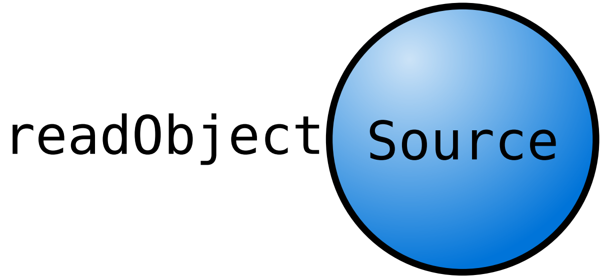
Security Issue: Untrusted data can trigger arbitrary code execution

The Gadget Chain Concept

A **gadget chain** is a sequence of method calls that leads from a safe entry point to a dangerous operation

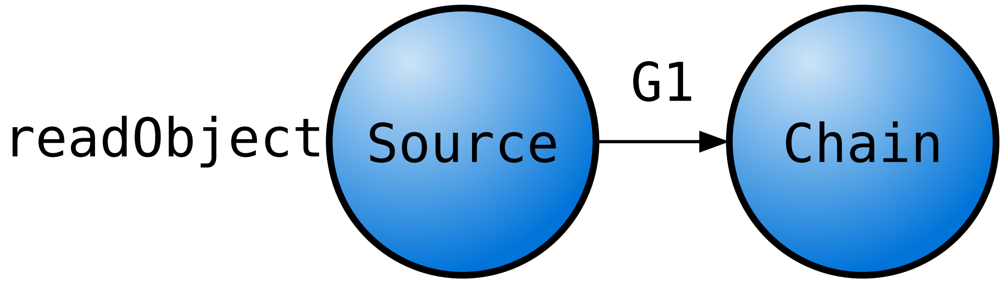
The Gadget Chain Concept

A **gadget chain** is a sequence of method calls that leads from a safe entry point to a dangerous operation



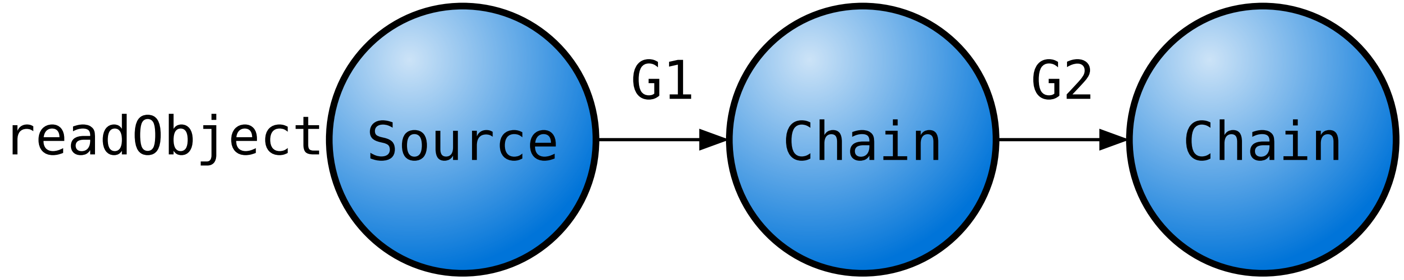
The Gadget Chain Concept

A **gadget chain** is a sequence of method calls that leads from a safe entry point to a dangerous operation



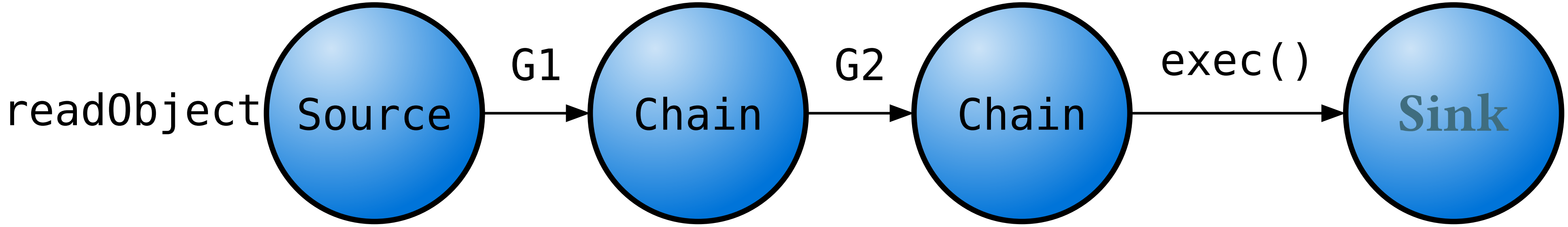
The Gadget Chain Concept

A **gadget chain** is a sequence of method calls that leads from a safe entry point to a dangerous operation



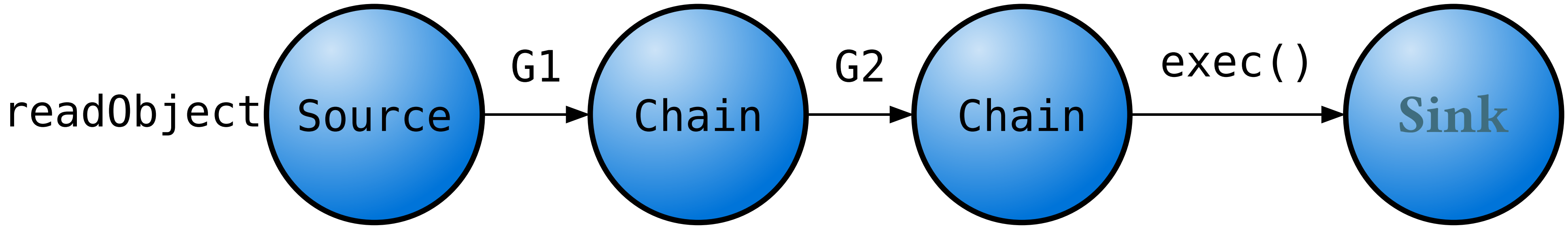
The Gadget Chain Concept

A **gadget chain** is a sequence of method calls that leads from a safe entry point to a dangerous operation



The Gadget Chain Concept

A **gadget chain** is a sequence of method calls that leads from a safe entry point to a dangerous operation



- Leverages existing classes on the classpath
- No need to inject new code - just arranges existing functionality
- Property-Oriented Programming (POP)

Why This Matters

Critical Security Impact:

- CVSS scores often 9.0+ (Critical)
- Remote Code Execution (RCE) without authentication

Why This Matters

Critical Security Impact:

- CVSS scores often 9.0+ (Critical)
- Remote Code Execution (RCE) without authentication

Wide Attack Surface:

- Java RMI (Remote Method Invocation)
- JMX (Java Management Extensions)
- Message queues (JMS, Spring AMQP)
- Web frameworks (Spring, Struts)

Famous Vulnerabilities

Apache Commons Collections

- CVE-2015-7450, CVE-2015-7501
- InvokerTransformer gadget
- CVSS: 9.8 Critical

Famous Vulnerabilities

Apache Commons Collections

- CVE-2015-7450, CVE-2015-7501
- InvokerTransformer gadget
- CVSS: 9.8 Critical

Spring Framework

- CVE-2016-1000027 - HttpInvoker
- CVE-2023-34040 - Spring-Kafka
- Multiple gadget chains discovered

The Challenge

Manual Analysis is Hard:

The Challenge

Manual Analysis is Hard:

- Large codebases with thousands of classes
- Reflection-based method invocations

The Challenge

Manual Analysis is Hard:

- Large codebases with thousands of classes
- Reflection-based method invocations

Traditional Tools:

The Challenge

Manual Analysis is Hard:

- Large codebases with thousands of classes
- Reflection-based method invocations

Traditional Tools:

- `ysoserial` - Payload generator (requires known gadgets)
- Manual code review (time-consuming, error-prone)
- Dynamic testing (limited coverage)

The Challenge

Manual Analysis is Hard:

- Large codebases with thousands of classes
- Reflection-based method invocations

Traditional Tools:

- `ysoserial` - Payload generator (requires known gadgets)
- Manual code review (time-consuming, error-prone)
- Dynamic testing (limited coverage)

Solution: CodeQL - Automated semantic code analysis

CodeQL Introduction

What is CodeQL?

A semantic code analysis engine by GitHub

What is CodeQL?

A semantic code analysis engine by GitHub

- Treats code as data - creates queryable database
- Uses declarative query language (similar to SQL/Datalog)
- Performs deep semantic analysis, not just pattern matching

What is CodeQL?

A semantic code analysis engine by GitHub

- Treats code as data - creates queryable database
- Uses declarative query language (similar to SQL/Datalog)
- Performs deep semantic analysis, not just pattern matching

```
from MethodAccess call
where call.getMethod().hasName("readObject")
select call, "Potential deserialization"
```

What is CodeQL?

A semantic code analysis engine by GitHub

- Treats code as data - creates queryable database
- Uses declarative query language (similar to SQL/Datalog)
- Performs deep semantic analysis, not just pattern matching

```
from MethodAccess call
where call.getMethod().hasName("readObject")
select call, "Potential deserialization"
```

Think of it as: SQL for code, but with understanding of program semantics

Key Capabilities

1. Data Flow Analysis

- Track how data moves through the program
- Identify sources (input) and sinks (dangerous operations)

2. Taint Tracking

- Follow untrusted data from entry points to sensitive operations
- Understand data transformations

Key Capabilities

3. Control Flow Analysis

- Understand execution paths
- Identify reachable code

4. Cross-Project Analysis

- Analyze entire dependency trees
- Find vulnerabilities in third-party libraries

CodeQL Architecture

Three-Step Process:

CodeQL Architecture

Three-Step Process:

1. **Create Database** - Extract semantic information from source code

```
codeql database create myapp-db --language=java
```

CodeQL Architecture

Three-Step Process:

1. **Create Database** - Extract semantic information from source code

```
codeql database create myapp-db --language=java
```

2. **Write/Run Queries** - Query the database for patterns

```
codeql database analyze myapp-db query.ql
```

CodeQL Architecture

Three-Step Process:

1. **Create Database** - Extract semantic information from source code

```
codeql database create myapp-db --language=java
```

2. **Write/Run Queries** - Query the database for patterns

```
codeql database analyze myapp-db query.ql
```

3. **Analyze Results** - Review findings and validate

```
# Results in SARIF format for integration
```

CodeQL for Deserialization

Built-in Deserialization Detection

CodeQL includes `java/unsafe-deserialization` query

```
/**
 * @name Unsafe deserialization
 * @description Deserializing user-controlled data may allow
 *              attackers to execute arbitrary code
 * @kind path-problem
 * @id java/unsafe-deserialization
 */
import java
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.security.UnsafeDeserializationQuery
```

Understanding Sources and Sinks

Source: Where untrusted data enters the system

```
predicate isSource(DataFlow::Node source) {
  source instanceof RemoteFlowSource
  // HTTP requests, socket input, etc.
}
```

Understanding Sources and Sinks

Sink: Dangerous operation that should not receive untrusted data

```
predicate isSink(DataFlow::Node sink) {
  exists(MethodAccess ma |
    ma.getMethod().hasName("readObject") and
    ma.getMethod().getDeclaringType()
      .hasQualifiedName("java.io", "ObjectInputStream") and
    sink.asExpr() = ma.getQualifier()
  )
}
```

Taint Tracking Configuration

```
import java
import semmle.code.java.dataflow.TaintTracking
module DeserializationConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource}
  predicate isSink(DataFlow::Node sink) {
    exists(MethodAccess ma |
      ma.getMethod().hasName("readObject") and
      sink.asExpr() = ma.getQualifier()
    )}}
```

Finding Gadget Chains

QLinspector - Advanced CodeQL queries by Synacktiv

GitHub: github.com/synacktiv/QLinspector

Finding Gadget Chains

QLinspector - Advanced CodeQL queries by Synacktiv

GitHub: github.com/synacktiv/QLinspector

Available Queries:

- `QLinspector.ql` - Main gadget chain finder
- `BeanFactoryGadgetFinder.ql` - JNDI injection chains
- `CommonsBeanutilsGadgetFinder.ql` - Alternative gadgets
- `ObjectFactoryFinder.ql` - BeanFactory alternatives

QLinspector Usage

Step 1: Create CodeQL Database

```
codeql database create target-app-db --language=java
```

Step 2: Run QLinspector Query

```
codeql database analyze target-app-db \
  --format=sarif-latest \
  --output=results.sarif \
  ./QLinspector/QLinspector.ql
```

Step 3: Review Results

Finding Runtime.exec Sinks

Track execution sinks reachable from deserialization:

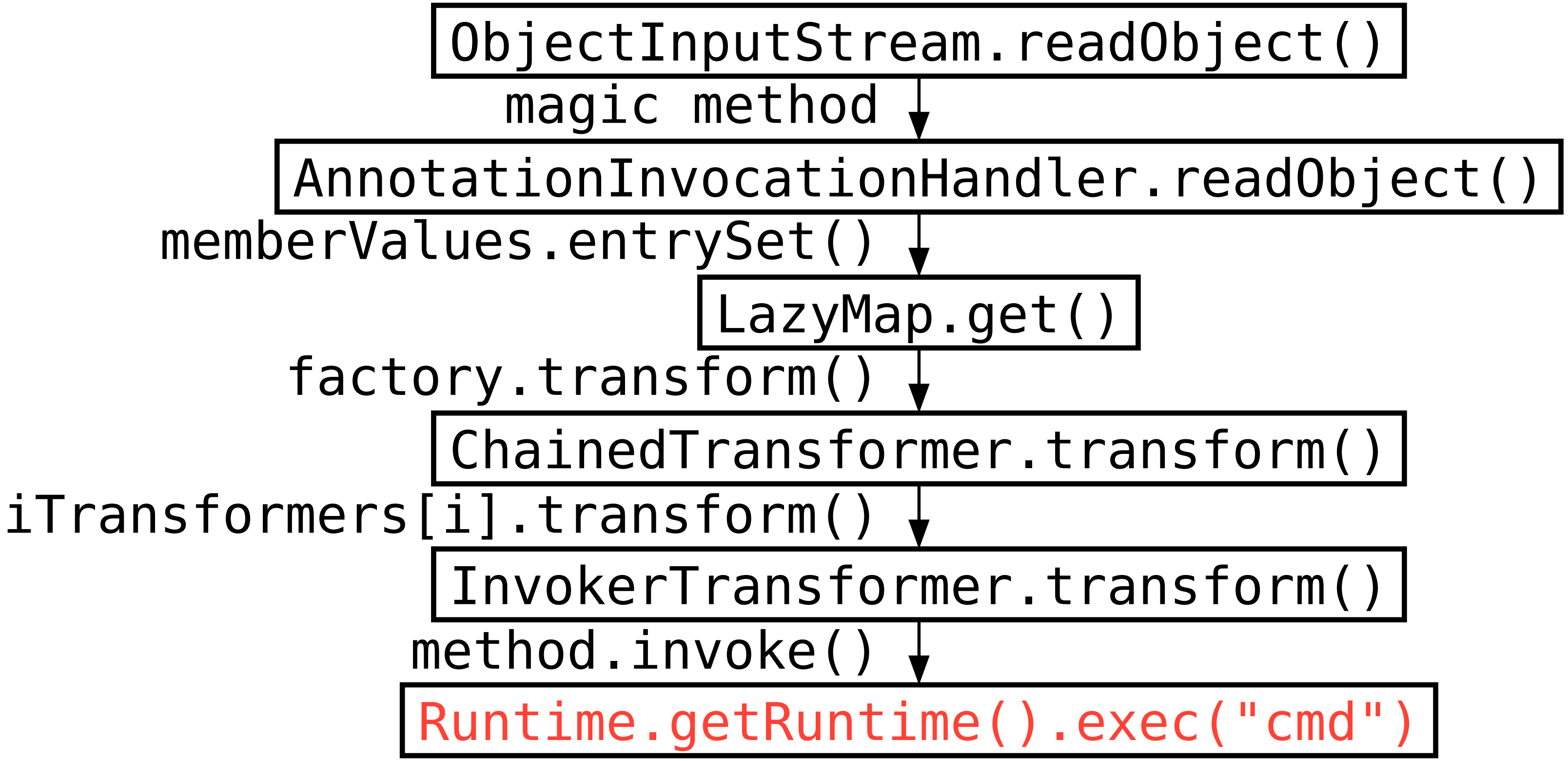
```
import java
class RuntimeExecCall extends MethodAccess {
  RuntimeExecCall() {
    this.getMethod().hasName("exec") and
    this.getMethod().getDeclaringType()
      .hasQualifiedName("java.lang", "Runtime")
  }
}
```

Finding Runtime.exec Sinks

```
from RuntimeExecCall exec
where exists(Method m |
  m.hasName("readObject") and
  exec.getEnclosingCallable().calls*(m)
)
select exec, "Potential gadget chain to Runtime.exec"
```

Real Example: CommonsCollections1

The CommonsCollections1 Gadget Chain



The Gadget Chain Explained

Step 1: Deserialize malicious `AnnotationInvocationHandler`

Step 2: `readObject()` iterates over `memberValues` (a `LazyMap`)

Step 3: `LazyMap.get()` calls `factory.transform()` on missing keys

Step 4: `ChainedTransformer` chains multiple transformations

Step 5: `InvokerTransformer` uses reflection to call methods

Step 6: Chain leads to `Runtime.getRuntime().exec()`

CodeQL Query for CommonsCollections1

```
import java
import semmle.code.java.dataflow.TaintTracking

class CommonsCollectionsGadget extends
TaintTracking::Configuration {
  CommonsCollectionsGadget() { this =
"CommonsCollectionsGadget" }

  override predicate isSource(DataFlow::Node source) {
    exists(Method m |
      m.hasName("readObject") and
```

```
      m.getDeclaringType().hasQualifiedName("java.io",

"ObjectInputStream") and
      source.asParameter() = m.getAParameter()
    )
  }
  override predicate isSink(DataFlow::Node sink) {
    exists(MethodAccess ma |
      ma.getMethod().hasName("exec") and
      ma.getMethod().getDeclaringType()
        .hasQualifiedName("java.lang", "Runtime") and
      sink.asExpr() = ma.getAnArgument()
    )
```

```
  }

  override predicate isAdditionalTaintStep(
    DataFlow::Node node1, DataFlow::Node node2
  ) {
    // Track through InvokerTransformer.transform()
    exists(MethodAccess ma |
      ma.getMethod().hasName("transform") and
      node1.asExpr() = ma.getQualifier() and
      node2.asExpr() = ma
    )
  }
}
```

Practical Workflow

Complete Analysis Workflow

1. Reconnaissance

- Identify Java applications in scope
- Check dependencies (pom.xml, build.gradle)

Complete Analysis Workflow

1. Reconnaissance

- Identify Java applications in scope
- Check dependencies (pom.xml, build.gradle)

2. Database Creation

```
codeql database create app-db --language=java \
  --command="mvn clean compile"
```

Complete Analysis Workflow

1. Reconnaissance

- Identify Java applications in scope
- Check dependencies (pom.xml, build.gradle)

2. Database Creation

```
codeql database create app-db --language=java \
  --command="mvn clean compile"
```

3. Query Selection

- Run QLinspector for gadget discovery
- Custom queries for specific patterns

Complete Analysis Workflow

4. Analysis

```
codeql database analyze app-db \
  codeql/java-queries:Security \
```

Complete Analysis Workflow

4. Analysis

```
codeql database analyze app-db \
  codeql/java-queries:Security \
```

5. Validation

- Review identified paths
- Check if gadget chain is exploitable

Learning Resources

Official CodeQL Resources

Documentation & Guides:

- CodeQL Documentation - Comprehensive reference
- CodeQL for Java - Java-specific guide
- Data Flow Analysis - Taint tracking guide

Official CodeQL Resources

Documentation & Guides:

- CodeQL Documentation - Comprehensive reference
- CodeQL for Java - Java-specific guide
- Data Flow Analysis - Taint tracking guide

Learning Series:

- Zero to Hero Part 1 - Fundamentals
- Zero to Hero Part 2 - Getting started
- Zero to Hero Part 3 - Security research

Java Deserialization Resources

Essential Reading:

- Synacktiv: Finding Gadgets Part 1 & 2 - Deep dive into gadget discovery
- Synacktiv: Finding Gadgets 2022 - Modern techniques
- ysoserial - Essential payload generator tool

Java Deserialization Resources

Essential Reading:

- Synacktiv: Finding Gadgets Part 1 & 2 - Deep dive into gadget discovery
- Synacktiv: Finding Gadgets 2022 - Modern techniques
- ysoserial - Essential payload generator tool

Tutorials & Guides:

- PortSwigger Web Security Academy - Interactive learning
- Understanding Gadget Chains - Beginner-friendly

Advanced Resources

Research & Tools:

- QLinspector - CodeQL queries for gadget finding
- GitHub Security Lab Research - Real vulnerability findings
- Java Deserialization Cheat Sheet - Comprehensive catalog

Advanced Resources

Research & Tools:

- QLinspector - CodeQL queries for gadget finding
- GitHub Security Lab Research - Real vulnerability findings
- Java Deserialization Cheat Sheet - Comprehensive catalog

Community Resources:

- Awesome CodeQL - Curated resource list
- CodeQL Zero to Hero Exercises - Hands-on challenges

Conclusion

Key Takeaways

1. Deserialization is Critical

- CVSS scores typically 9.0+
- Wide attack surface in enterprise Java
- Affects many popular frameworks

2. CodeQL Enables Automation

- Scales to millions of lines of code
- Finds complex gadget chains automatically
- Low false positive rate with proper queries

Questions?

Thank you for your attention!

Resources:

- GitHub: github/codeql
- QLinspector: synacktiv/QLinspector
- ysoserial: frohoff/ysoserial

Happy Hunting! 🔍