Software Supply Chain Security at Runtime

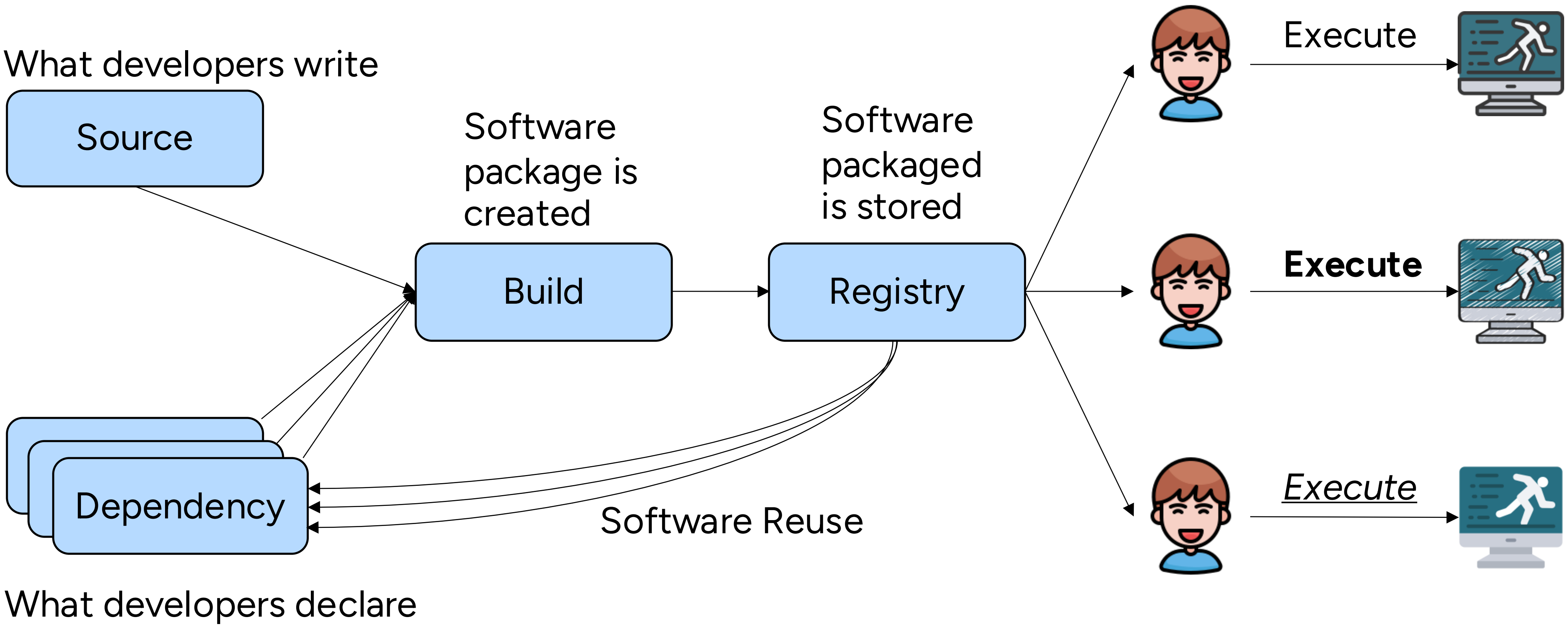
Whitepaper: SBOM.EXE: Countering Dynamic Code Injection based on Software Bill of Materials in Java

Aman Sharma, Martin Wittlinger, Benoit Baudry, Martin Monperrus

Outline

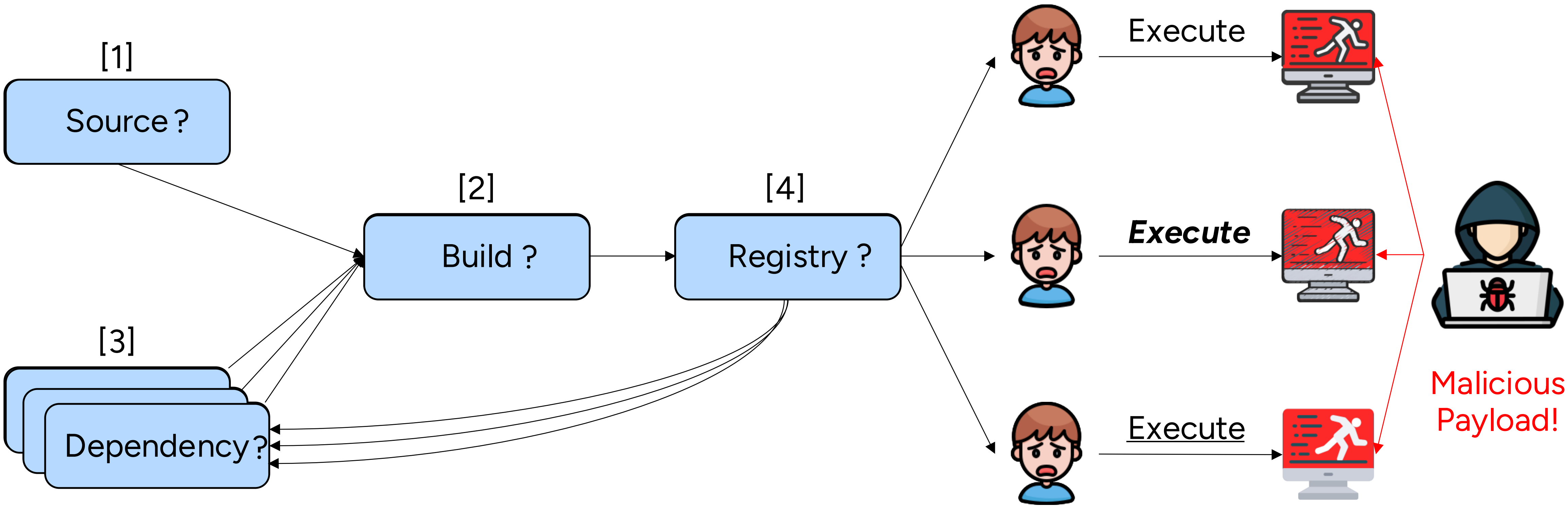
- Background about Software Supply Chain and Java
- Demo of log4shell exploit
- Related Work
- Novel Tool: SBOM.EXE: Countering Dynamic Code Injection based on Software Bill of Materials in Java
- Demo of log4shell mitigation
- Evaluation
- Future Work
- Conclusion

What is Software Supply Chain?



Aman Sharma | amansha@kth.se

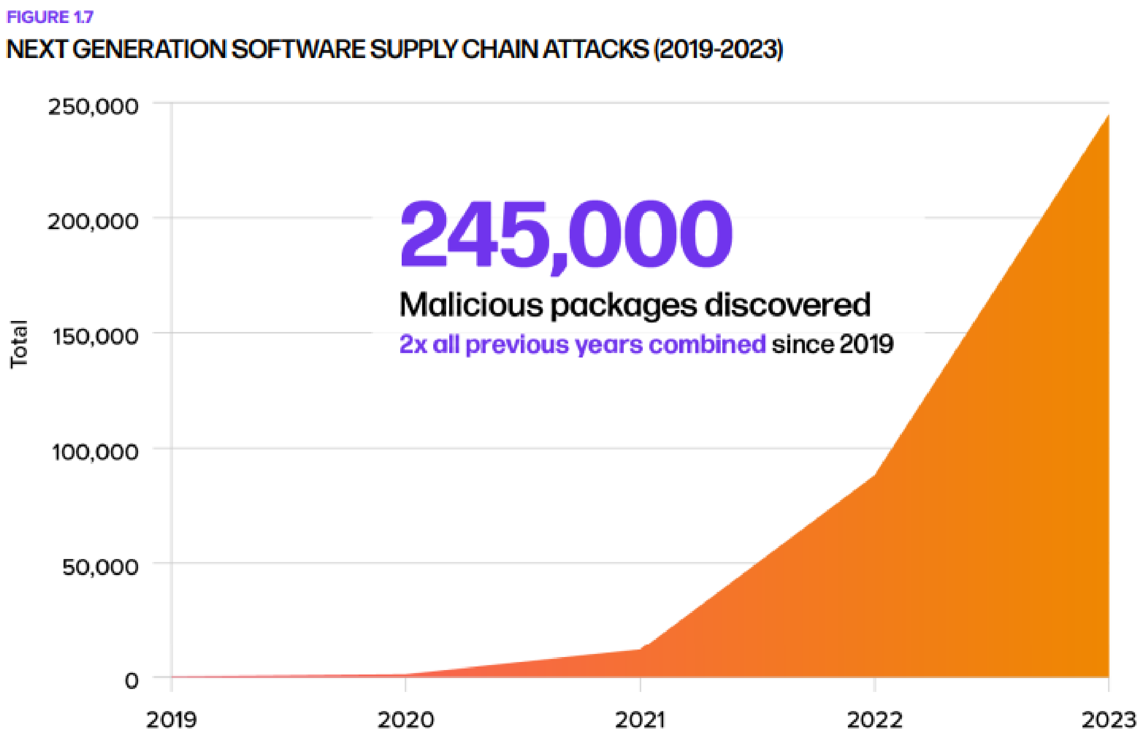
What is Software Supply Chain Attack?



[1] Q. Wu et al. "On the Feasibility of Stealthily Introducing Vulnerabilities in Open-Source Software via Hypocrite Commits", 2021
[2] S. Peisert et al. "Perspectives on the solarwinds incident," IEEE Security Privacy, 2021
[3] P. Ladisa et al. Towards the Detection of Malicious Java Packages. In Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, 2022
[4] J. Cappos et al. "A look in the mirror: attacks on package managers," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008

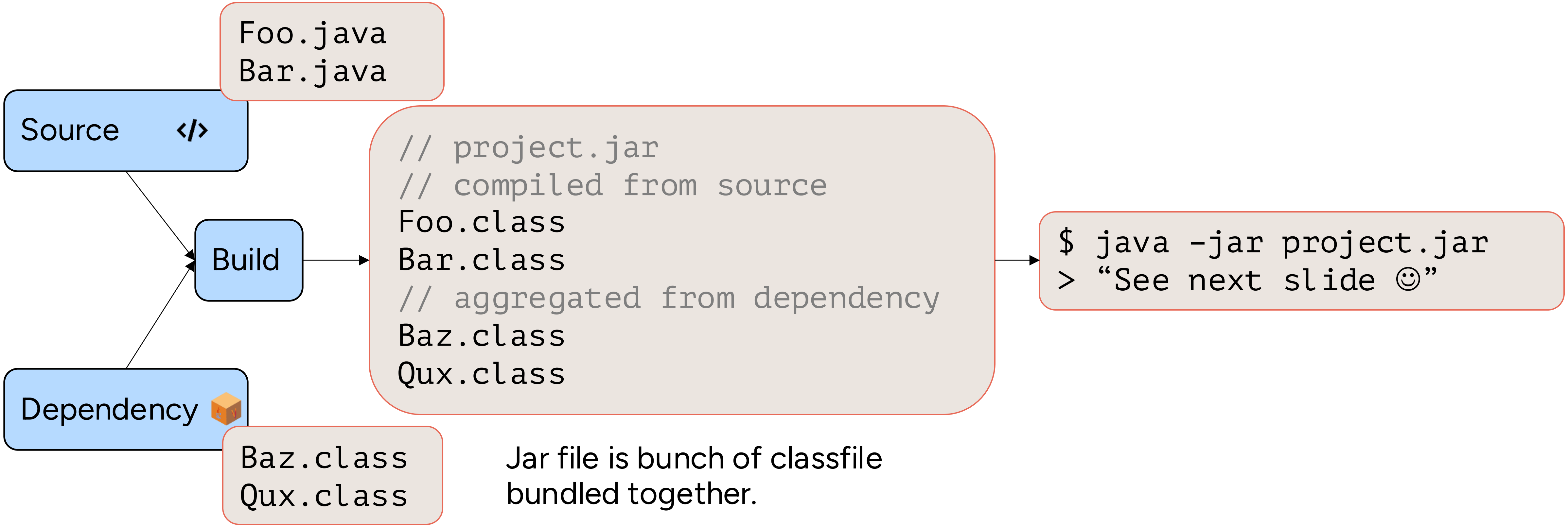
How prevalent Software Supply Chain attacks are?

2023 saw **twice** as many software supply chain attacks as 2019-2022 combined [5].



[5] Sonatype. *9th Annual State of the Software Supply Chain.* October (2023)

Intricacies of Java: Build



Jar file is bunch of classfile bundled together.

Intricacies of Java: Runtime

Java Virtual Machine

We, java lovers, just call the entire process **classloading**.

```
> So much for hello world :/
```

Jar from registry.

[6]

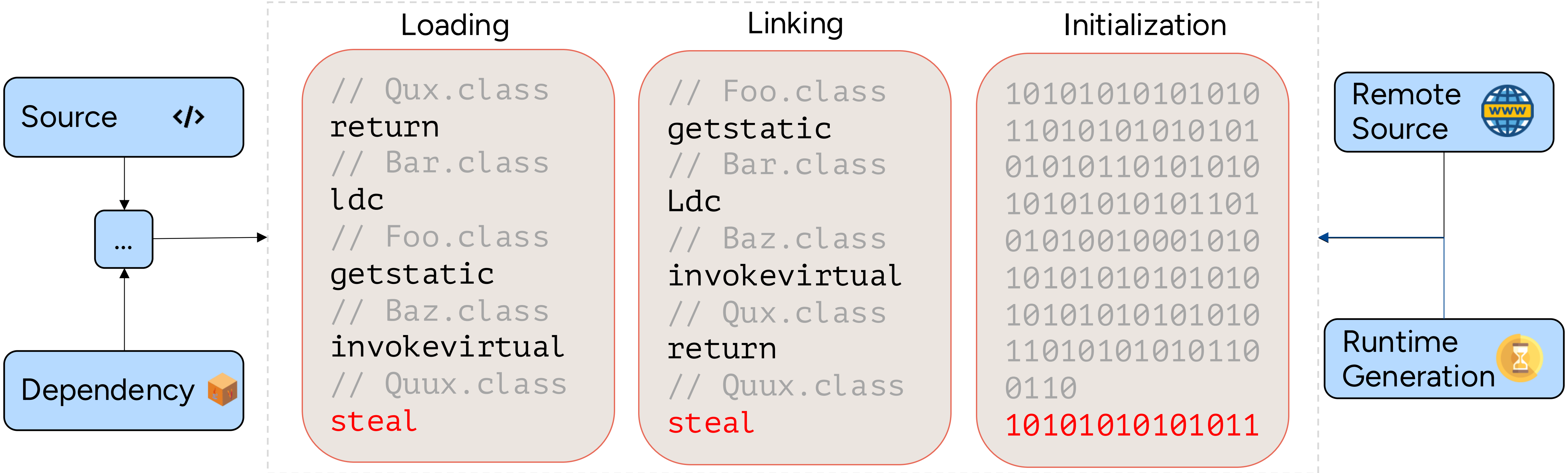
Loading	Linking	Initialization

// Qux.class
return
// Bar.class
ldc
// Foo.class
getstatic
// Baz.class
invokevirtual
	// Foo.class
	getstatic
	// Bar.class
	Ldc
	// Baz.class
	invokevirtual
	// Qux.class
	return
		10101010101010
		11010101010101
		01010110101010
		10101010101101
		01010010001010
		10101010101010
		10101010101010
		11010101010110

Gets binary representation	Combines into runtime state	Ready for execution!

[6] Oracle, Chapter 5. Loading, Linking, and Initializing (oracle.com), 2023, https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-5.html

How is Software Supply Chain Attack relevant?



Dynamic classloading could be exploited!!!

- Code can be downloaded at runtime.
- Code can be generated at runtime. [7]

[7] Oracle, ClassLoader (Java SE 21 & JDK 21) (oracle.com), 2023, https://docs.oracle.com/en%2Fjava%2Fjavase%2F21%2Fdocs%2Fapi%2F%2F/java.base/java/lang/ClassLoader.html#builtinLoaders

Demo: Exploitation

CVE-2021-44228 (Log4Shell)

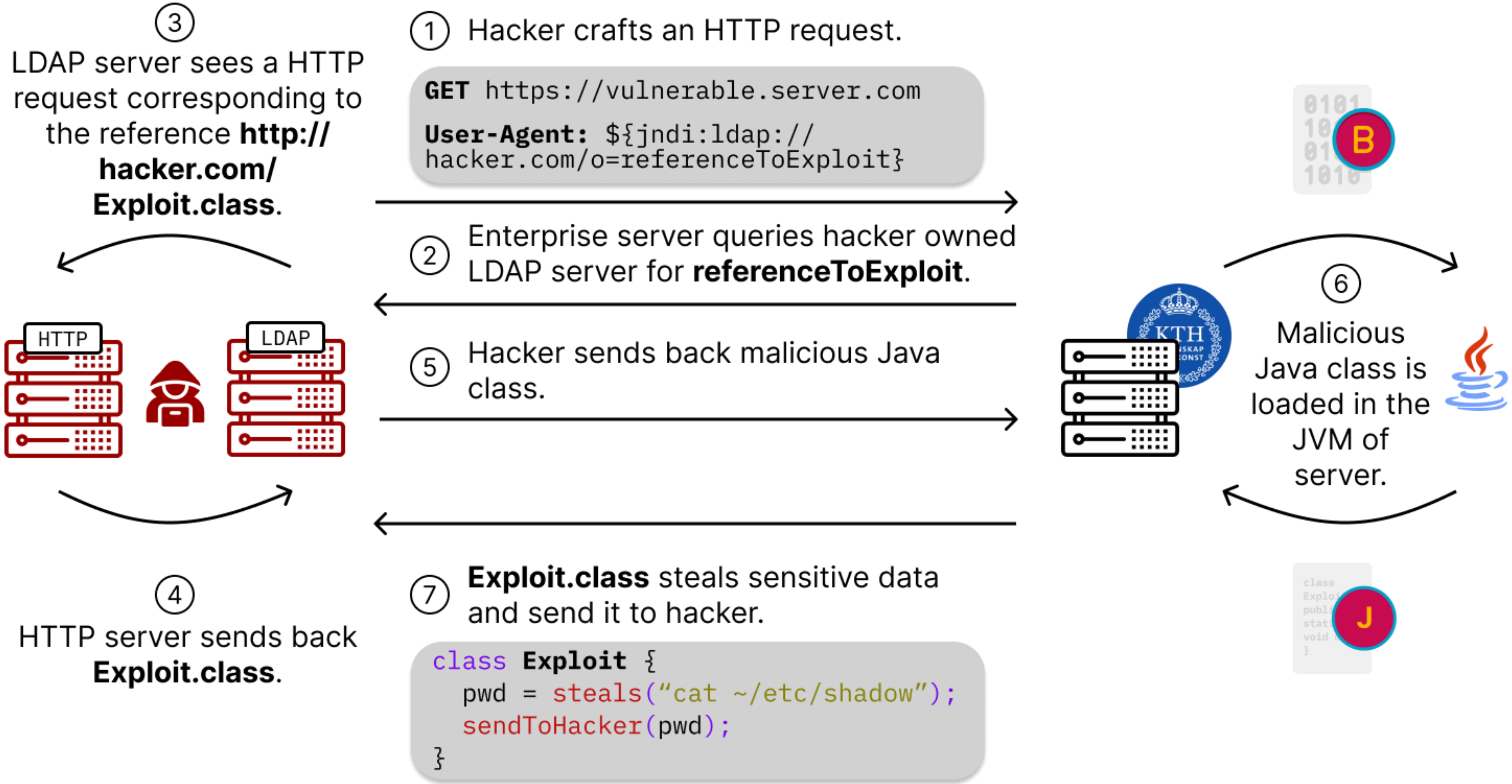
Source: https://github.com/chains-project/exploits-for-sbom.exe/tree/main/rq2/log4shell-2021-44228

Demo Steps (for replication later) for exploit

1. Make sure Java 17 (or earlier) is on `PATH`.
2. Inspect code in src/main/java and run `./normal-usage.sh`. This should log “this is an error”.
3. Now startup the LDAP server by going to root of the project and run `java -jar target/RogueJndi-1.1.jar --command "gedit /etc/passwd"`.
    1. This will inject the command argument in the bytecode that will be hosted on LDAP server.
4. Next, go back to the same directory where “normal-usage” was run. Run `./malicious-usage.sh`. This will execute the malicious bytecode.

Aman Sharma | amansha@kth.se

Log4Shell – a software supply chain attack at runtime



Why is Log4Shell a Software Supply Chain attack?

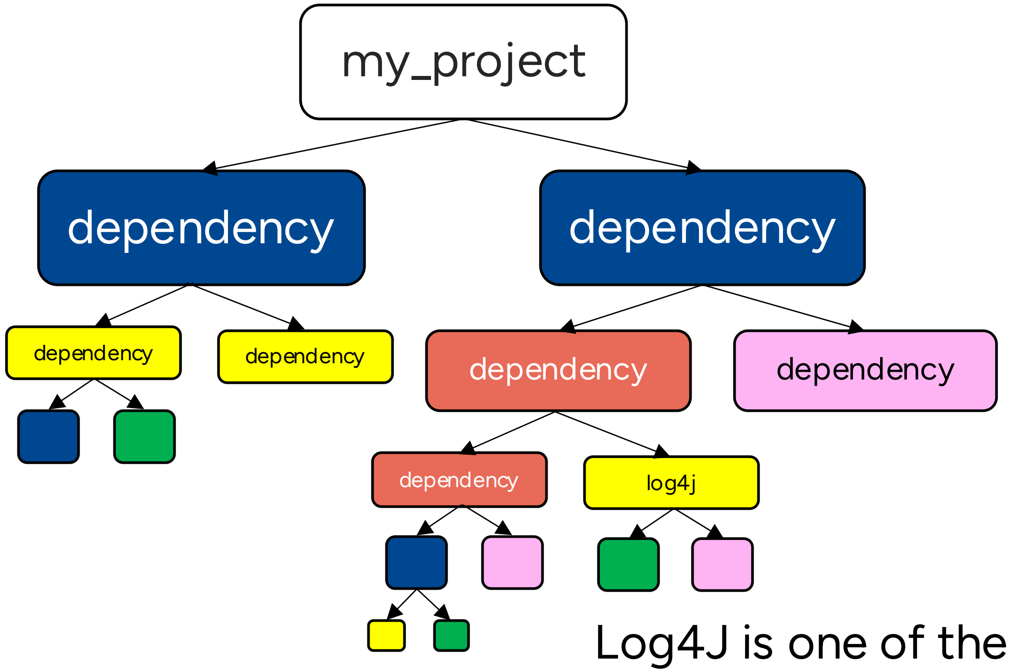
Only clue: it happened after execution with the malicious payload.

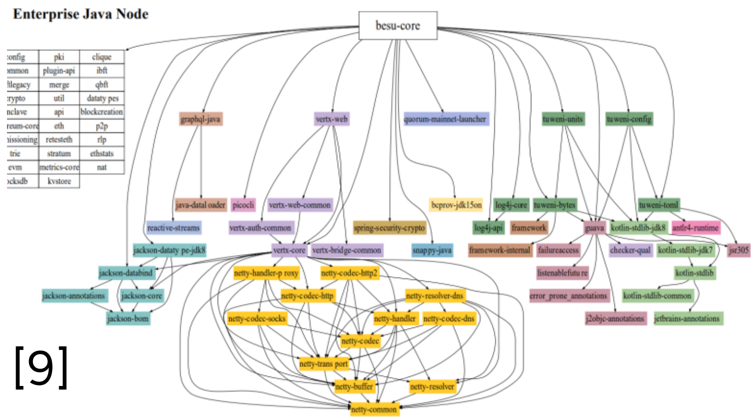
Source?

Build?

Registry?

Dependency?





[9]

Log4J is one of the most popular logging libraries. [8]

[8] B. Chen et al. “Studying the use of Java logging utilities in the wild”. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE ‘20)
[9] C. Soto-Valero et al. “The Multibillion Dollar Software Supply Chain of Ethereum”, Computer, 2022

Problem: Java can trigger <u>download</u> or <u>generation</u> of unknown code.

Related Work: Permission Managers

Overview: Define access permissions for the application at varying granularities.

[10] P. C. Amusuo et al. “Preventing Supply Chain Vulnerabilities in Java with a Fine-Grained Permission Manager”, arXiv, 2023

- Map network, filesystem, and process permissions to dependency.

[11] N. Vasilakis et al. “Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction”, In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021

- Map read, write, execute permissions to each field and method.

[12] Y. Xu et al. “SWAT4J: Generating System Call Allowlist for Java Container Attack Surface Reduction”, IEEE International Conference on Software Analysis, Evolution and Reengineering, 2024

- Restrict invocation of system calls at runtime for different containers.

Shortcomings: 1) Setting permissions are susceptible to privilege escalation. 2) Requires modification to runtime, eg, JVM, node.js.

Related Work: Compartmentalization

Overview: Different parts of an application are executed in different protection domains

Hardware level compartmentalization

[13] J. Jiang et al. “Uranus: Simple, Efficient SGX Programming and its Applications”, Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, 2020

- Proposes API to run code on Intel Software Guard Extensions which is also called an enclave in CPU.

Software level compartmentalization

[14] C. Song et al. “Exploiting and Protecting Dynamic Code Generation”, Network and Distributed System Security Symposium, 2015

- Enables to run trusted and untrusted parts in different processes.

Shortcomings: 1) Requires manual work to split code. 2) Context switch overhead could be high.

Related Work: Integrity Measurement

Overview: Measuring application in terms of its side effects, memory, or any kind of execution behavior.

Manual verification of integrity

[15] H. Ba et al. "RIM4J: An Architecture for Language-Supported Runtime Measurement against Malicious Bytecode in Cloud Computing", Information Technology and Its Applications, 2021

- Support for user to query the application for measurement and then subsequent verification.

Automated verification of integrity

[16] X. Wang et al. "RSDS: Getting System Call Whitelist for Container Through Dynamic and Static Analysis", IEEE International Conference on Cloud Computing, 2020

- Keeps a track of the allowlist of system calls that can be invoked.

Shortcomings: 1) Manual work of verifying integrity.

SBOM.exe falls under this category. It is the first automated tool for Java.

Related Work: Some more work on security

- Control Flow Integrity: checking that the application executes according to the control flow graph as intended.
  - [17] N. Burow et al. “Control-Flow Integrity: Precision, Security, and Performance”, ACM Computing Surveys (CSUR), 2017

- Oblivious Hashing: a technique where the side effects of executed code are verified.
  - [18] M. Ahmadvand, et al. “Practical Integrity Protection with Oblivious Hashing,” in Proceedings of the 34th Annual Computer Security Applications Conference, ser. ACSAC ’18, 2018

- Deserialization Attacks: attacker sends a serialized object to the application and expects the application to deserialize it.
  - [19] I. Sayar et al. “An In-depth Study of Java Deserialization Remote-Code Execution Exploits and Vulnerabilities,” ACM Transactions on Software Engineering and Methodology, 2023

Expectations from new approach

Minimal or no-modification to the runtime itself.

Should be fully automated – automatic detection and then proactive mitigation.

Minimal overhead.

Aman Sharma | amansha@kth.se

Solution: Create an allowlist of classes and only load those classes

Aman Sharma | amansha@kth.se

Step 1: Indexing

Problem: but, what to allowlist?

Solution: let's go for built-in classes, source code, dependencies, and finally all the dynamic code.

Step 1: Indexing

Problem: how to index built-in classes?

Solution: let's scan all classes using classgraph [20].

Problem: what about source code and dependencies?

Solution: finally, Software Bill of Materials, has one (now implemented) use case.

Problem: and code from remote source and runtime generated code?

Solution: if we execute the code, we can capture them. Let's just run tests.

```
java.util.List
org.apache.log4j.Log
Jdk.proxy1.$Proxy10
```

→ Checksum computation →

```
abf4834
8349dce
facaded
```

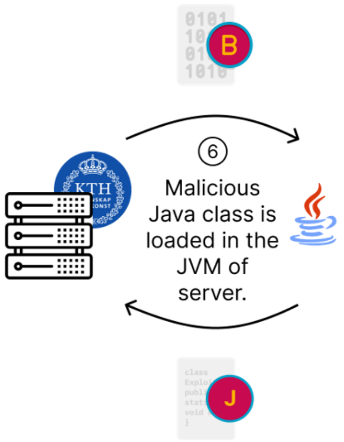
→

```
// allowlist.bomi
A hash table of class name
and checksums.
```

Part 1 done ✅

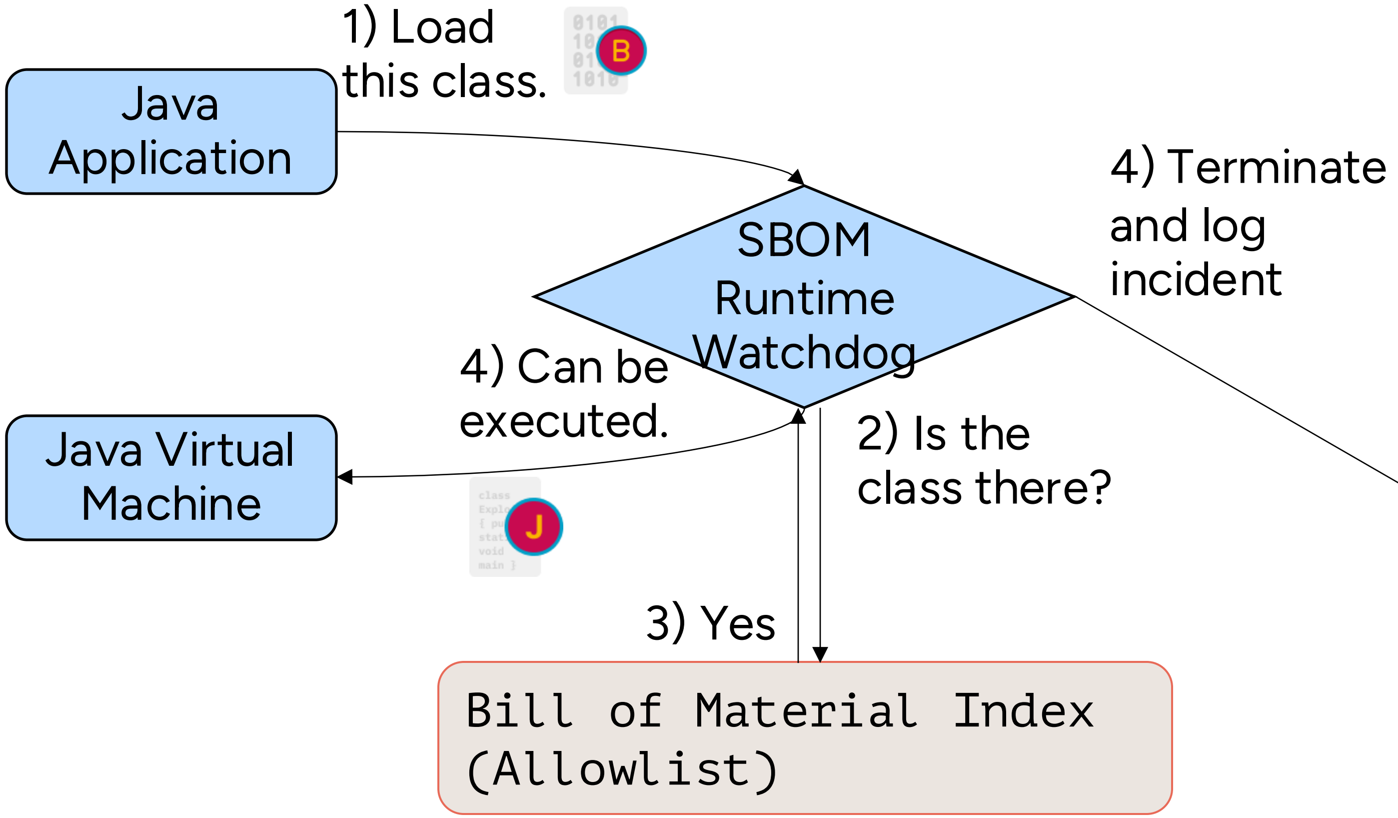
[20] L. Hutchison, Classgraph, GitHub.com, 2024

Step 2: Enforcement

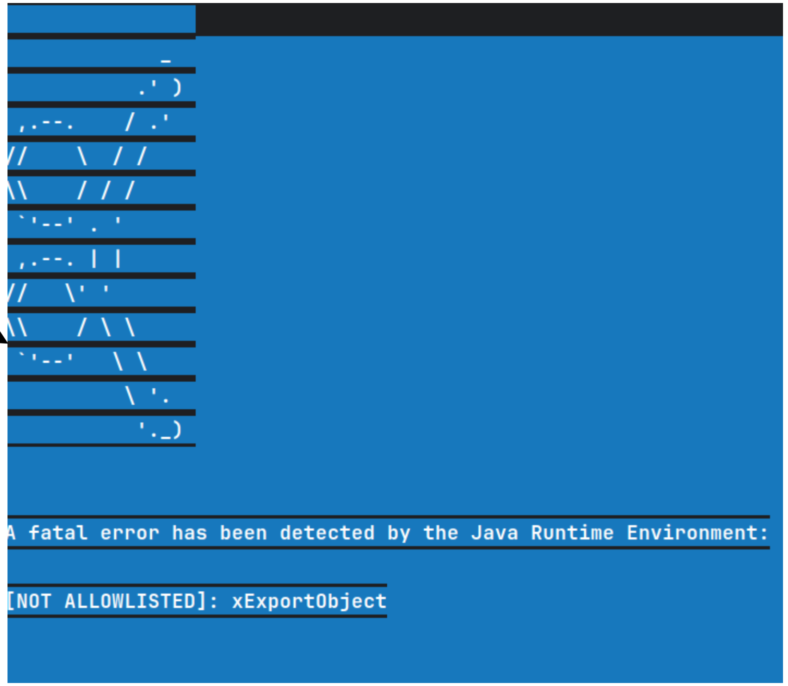


Problem: Java class is simply loaded without any integrity.

Solution: We intercept loading and then verify it.



Part 2 done ✅



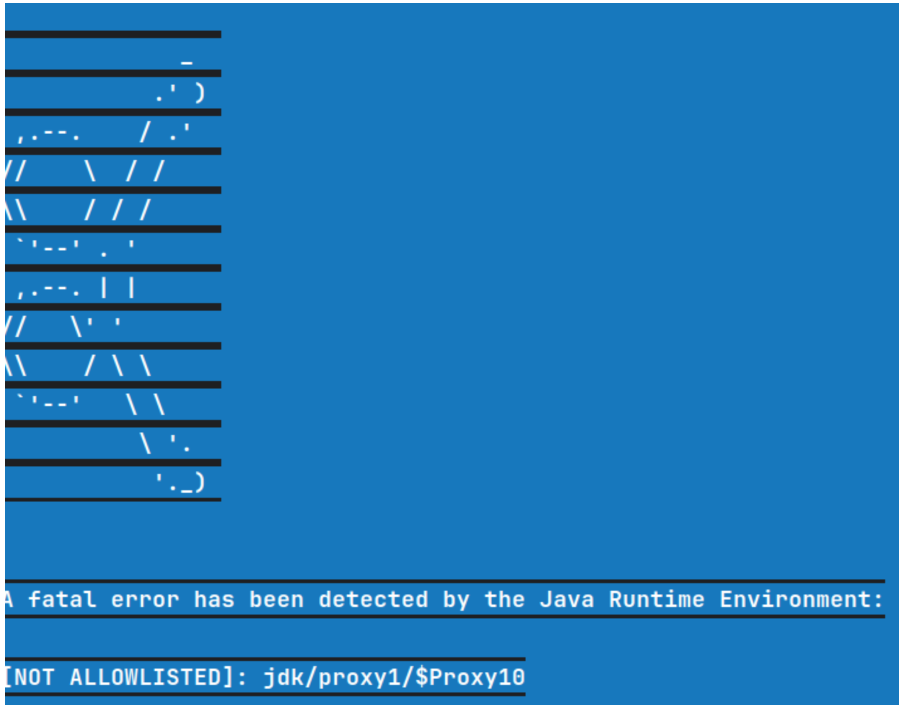
Test run

Okay, we seem to be done. Let's see what happened initially.

Problem 1: There seems to be false-positives. This class was in the allowlist.

Problem 2: There seems to be non-determinism in runtime generated code.

Solution: Let's ignore this non-deterministic features.



Bytecode Canonicalization

- Classnames could change across different executions.
- The type references change.
- The order of method is not fixed.

```diff
- public class $Proxy10 {
+ public class $Proxy7 {
-     private static $Proxy10.x;
+     private static $Proxy7.x;
-    m1 () {}
+    m3 () {}
-    m3 () {}
+    m1 () {}
}
```


Aman Sharma | amansha@kth.se

Novel Concepts Summarised

Problem: what to index?

Solution: 3 indexers for built-in classes source code, dependencies, and dynamic code.

Problem: how to load class with verification

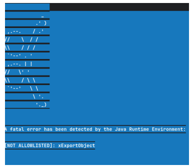
Solution: SBOM Runtime Watchdog is a novel tool to intercept Java classloading and verify integrity of each Java class.

Problem: non-determinism of Java bytecode.

Solution: Bytecode Canonincalization.

Problem: I miss blue screen of death in Linux (just kidding, no one likes kernel panic) ☹

Solution:



Demo: Mitigation

CVE-2021-44228 (Log4Shell)

Source: https://github.com/chains-project/exploits-for-sbom.exe/tree/main/rq2/log4shell-2021-44228

Aman Sharma | amansha@kth.se

Demo Steps (for replication later) for mitigation

1. To run with SBOM.exe protection, we follow two steps:
    1. Run `./generate-index.sh`. This outputs the `index.jsonl` which is the BOMI.
    2. Run `./sbom.exe.sh`. This would terminate the program just before the malicious class is initialized.

Aman Sharma | amansha@kth.se

Evaluation

RQ1: What is the scale of BOMI for all the study subjects?

- Methodology: Count classes in BOMI for all study subjects.

- Key findings:
  - BOMI-Environment contains around 25,000 classes, varying with JDK vendors and versions.
  - BOMI-SupplyChain strongly depends on number of classes in dependencies. It is also much more complex as dependencies come from so many diverse organizations, unlike JDK classes.
  - BOMI-Runtime constitutes few number of classes, but they are extremely important to detect in order to prevent attacks.

RQ1: What is the scale of BOMI for all the study subjects?

- Methodology: Count classes in BOMI for all study subjects.

<u>Takeaway: Around 25,000 for the selected study subjects.</u>

RQ2: To what extent can SBOM.EXE mitigate high-profile attacks in Java?

- Methodology: Replicate CVE-2021-44228 (Log4Shell), CVE-2021-42392 (H2 database console), and CVE-2022-33980 (Apache Commons Configuration) in a proof-of-concept and then run them with our BOMI and SBOM Runtime Watchdog attached.

- Key findings:
  - It successfully prevent execution of the malicious class loaded at runtime in all 3 CVEs. In more technical terms, it is prevented from being initialized.
  - **CVE-2021-44228 – mitigated**
  - **CVE-2021-42392 – mitigated**
  - **CVE-2022-33980 – mitiagted**

RQ2: To what extent can SBOM.EXE mitigate high-profile attacks in Java?

- Methodology: Replicate CVE-2021-44228 (Log4Shell), CVE-2021-42392 (H2 database console), and CVE-2022-33980 (Apache Commons Configuration) in a proof-of-concept and then run them with our BOMI and SBOM Runtime Watchdog attached.

Takeaway: Mitigated all 3 CVEs successfully.

RQ3: Is SBOM.EXE compatible with real-world applications?

- Methodology: Create BOMI for real-world projects – PDFBox, TTorrent, and GraphHopper – and then run these projects with SBOM Runtime Watchdog attached.

- Key findings:
  - SBOM.exe is fully compatible with PDFBox and TTorrent.
    - PDFBox:
      - Additional test for PDFBox Debugger
    - TTorrent:
      - Additional test to simulate torrent downloading
      - Update java bytecode for it to be compatible with Java 21
  - SBOM.exe is partially compatible with GraphHopper as one of the class names were randomly generated.

RQ3: Is SBOM.EXE compatible with real-world applications?

- Methodology: Create BOMI for real-world projects – PDFBox, TTorrent, and GraphHopper – and then run these projects with SBOM Runtime Watchdog attached.

- Key findings:

Takeaway: Compatible with real-world projects.

generated.

RQ4: What is the overhead of SBOM.EXE?

- Methodology: Measure performance using two metrics (using JMH) for 3 real world projects.
  - End-to-end time with warmup: Measure of the sum of how long it takes for the JVM to warm up and then run the application.
  - Workload time excluding warmup: Measure of the runtime of the application when all classes are fully JIT compiled (or optimized).

- Key findings:
  - The overhead introduced by SBOM.EXE is negligible after warm-up.
  - SBOM.exe is suitable for production environments where warm-up is not a concern.

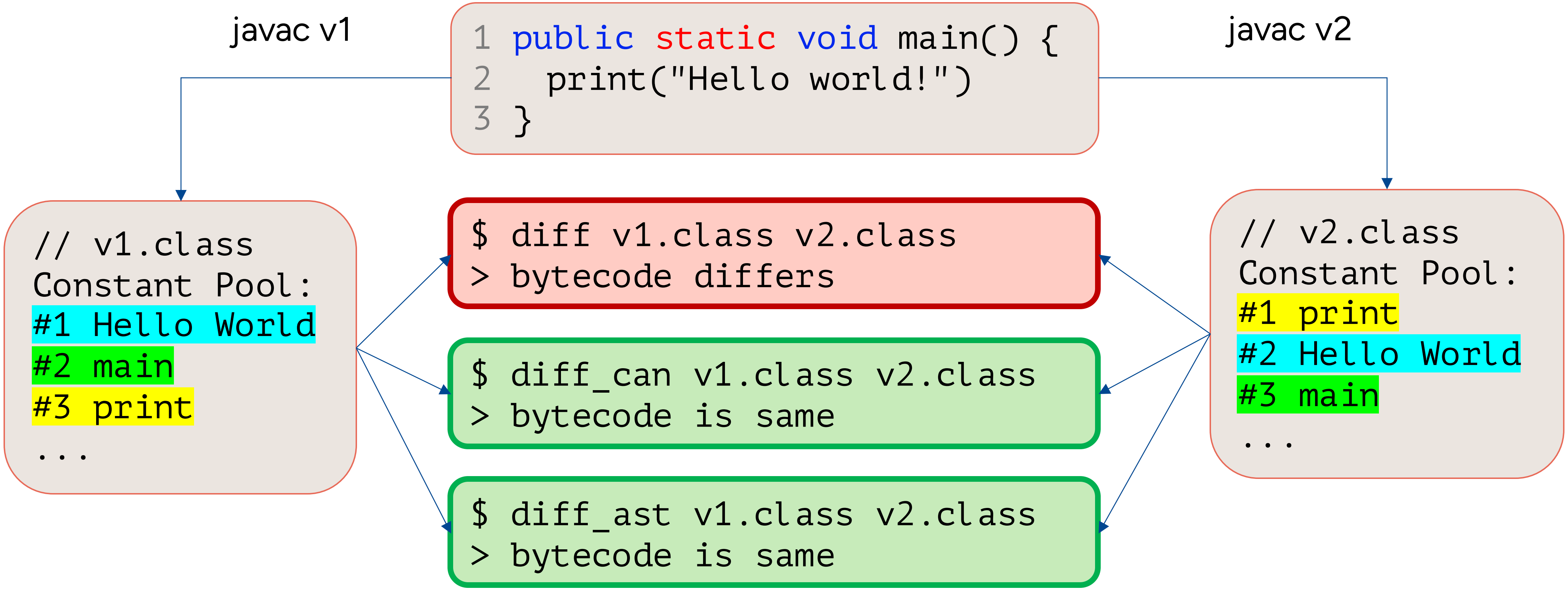
RQ4: What is the overhead of SBOM.EXE?

- Methodology: Measure performance using two metrics (using JMH) for 3 real world projects.
  - End-to-end time with warmup: Measure of the sum of how long it takes for the JVM to warm up and then run the application.
  - Workload time excluding warmup: Measure of the runtime of the application when all classes are fully JIT compiled (or optimized).

Takeaway: Negligible overhead.

Future Work: Diff bytecode for reproducibility

Problem: Diffing bytecode to know reproducibility is not optimum as current diff tools also incorporate non-deterministic features while diffing.



Future Work: Which dependency is running?

Problem: Knowing (and verifying) what dependency is executing is hard because that information is lost while building the application.

Daniel Williams, *“The Embedding and Retrieval of Software Supply Chain Information in Java Applications”*, KTH, 2024

+

A. Sharma et al., *“SBOM.EXE: Countering Dynamic Code Injection based on Software Bill of Materials in Java”*, arXiv, 2024

=

We will be able to tell what dependency is running in real-time!

Key contribution: embeds dependency information in JVM bytecode.

One of the contribution: can tell which bytecode is running in real-time.

Takeaways

SBOM.exe can mitigate three high-profile CVEs based on code generation and downloading.

SBOM.exe proposes a strong bytecode canonincalization algorithm which eliminates non-determinism in dynamic classes.

SBOM.exe can work well in production environment as shown by three real world projects.

Thank you!

Questions?

Personal Webpage

https://algomaster99.github.io

Whitepaper

https://arxiv.org/abs/2407.00246

Aman Sharma

amansha@kth.se

Project Link: https://github.com/chains-project/sbom.exe

Whitepaper: SBOM.EXE: Countering Dynamic Code Injection based on Software Bill of Materials in Java