



Software Supply Chain Security at Runtime

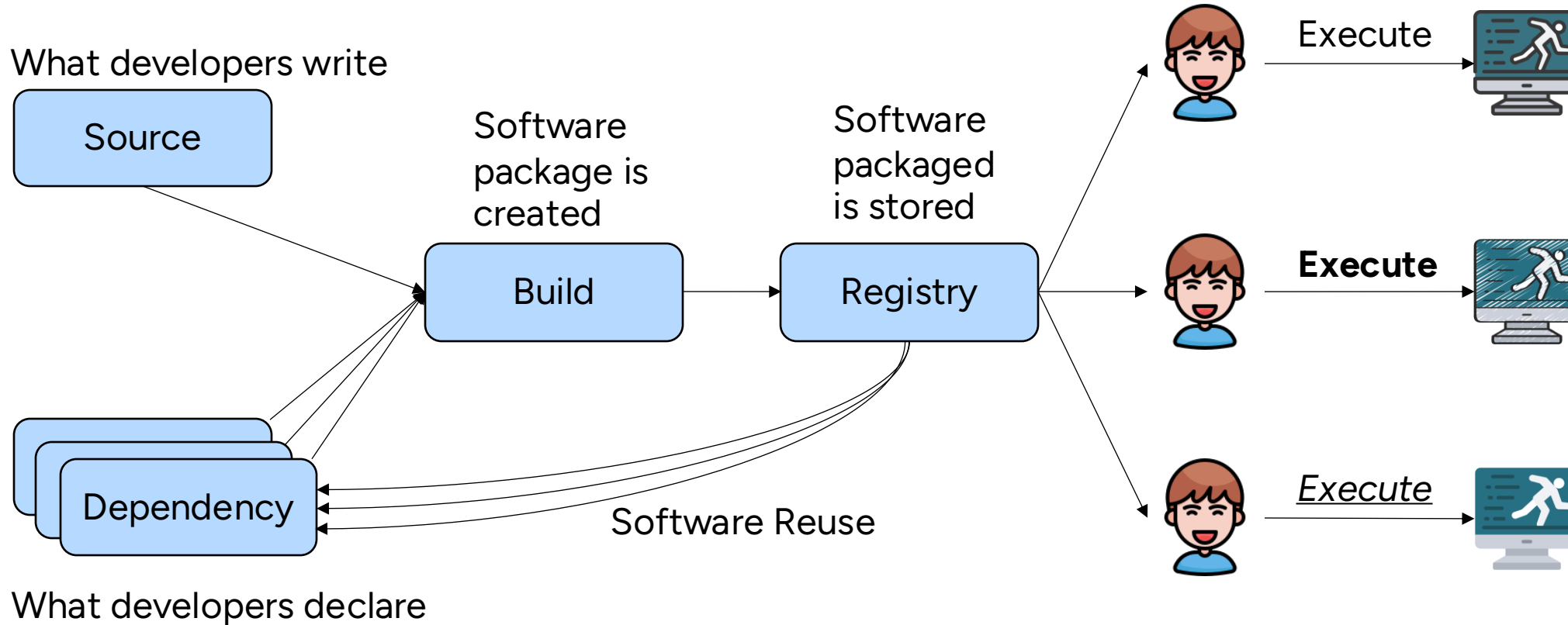
Whitepaper: SBOM.EXE: Countering Dynamic Code Injection based on
Software Bill of Materials in Java

Aman Sharma, Martin Wittlinger, Benoit Baudry, Martin Monperrus

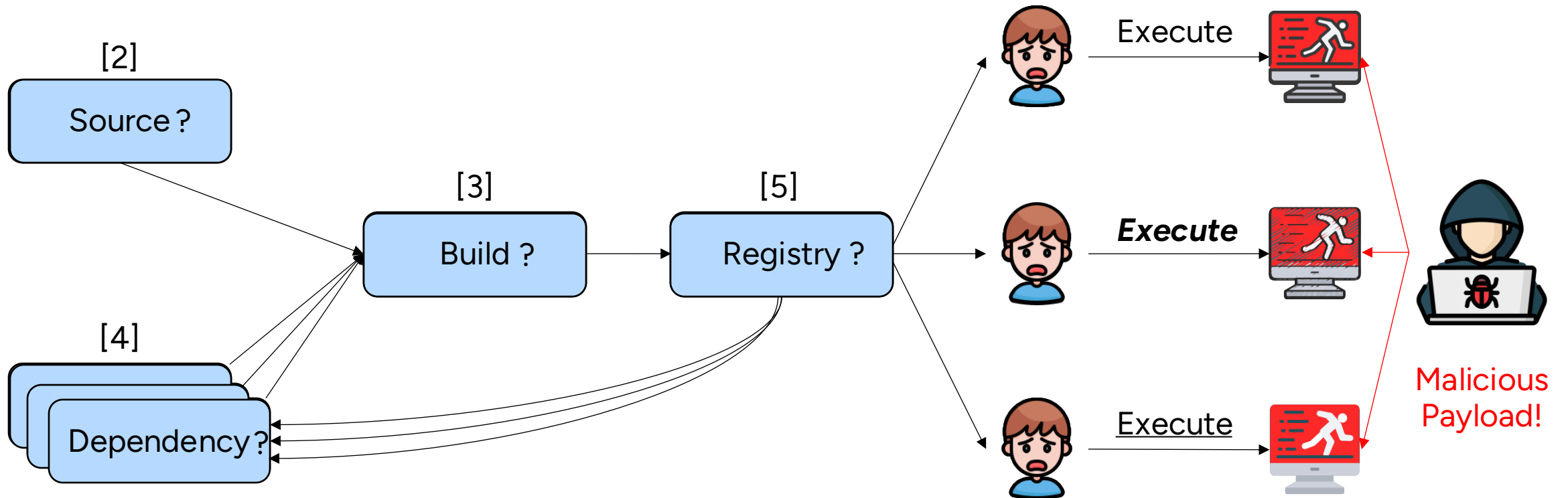
Outline

- 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?



What is Software Supply Chain Attack?



[2] Q. Wu et al. "On the Feasibility of Stealthily Introducing Vulnerabilities in Open-Source Software via Hypocrite Commits", 2021

[3] S. Peisert et al. "Perspectives on the solarwinds incident," IEEE Security Privacy, 2021

[4] 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

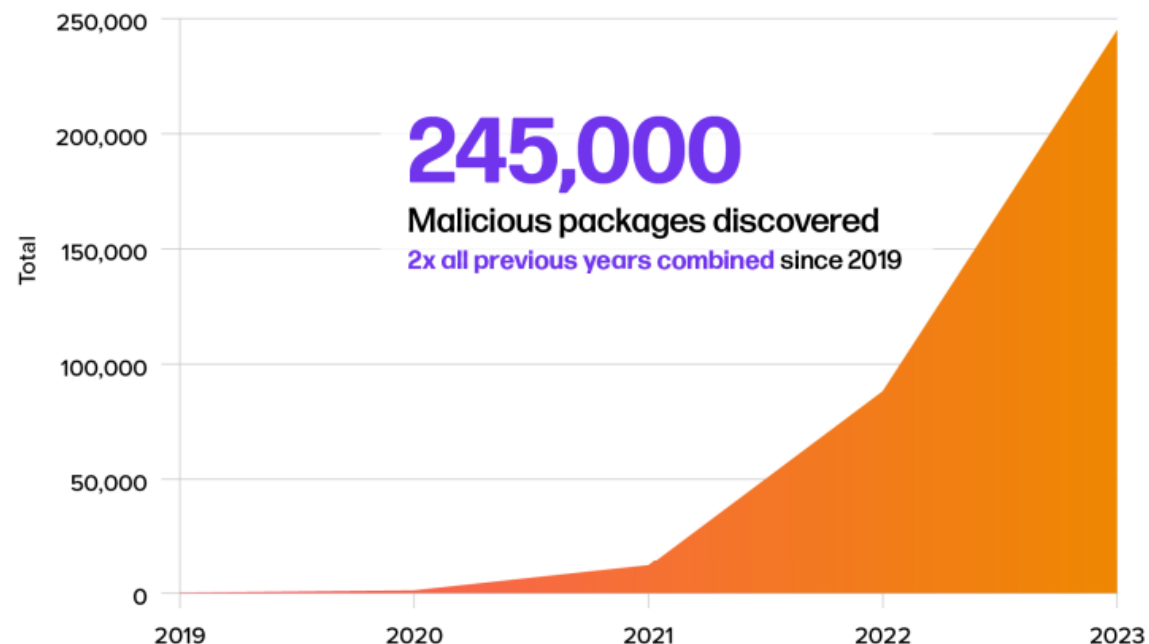
[5] J. Cappos et al. "A look in the mirror: attacks on package managers," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008

How prevalent Software Supply Chain attacks are?

2023 saw **twice** as many software supply chain attacks as 2019-2022 combined [2].

FIGURE 17

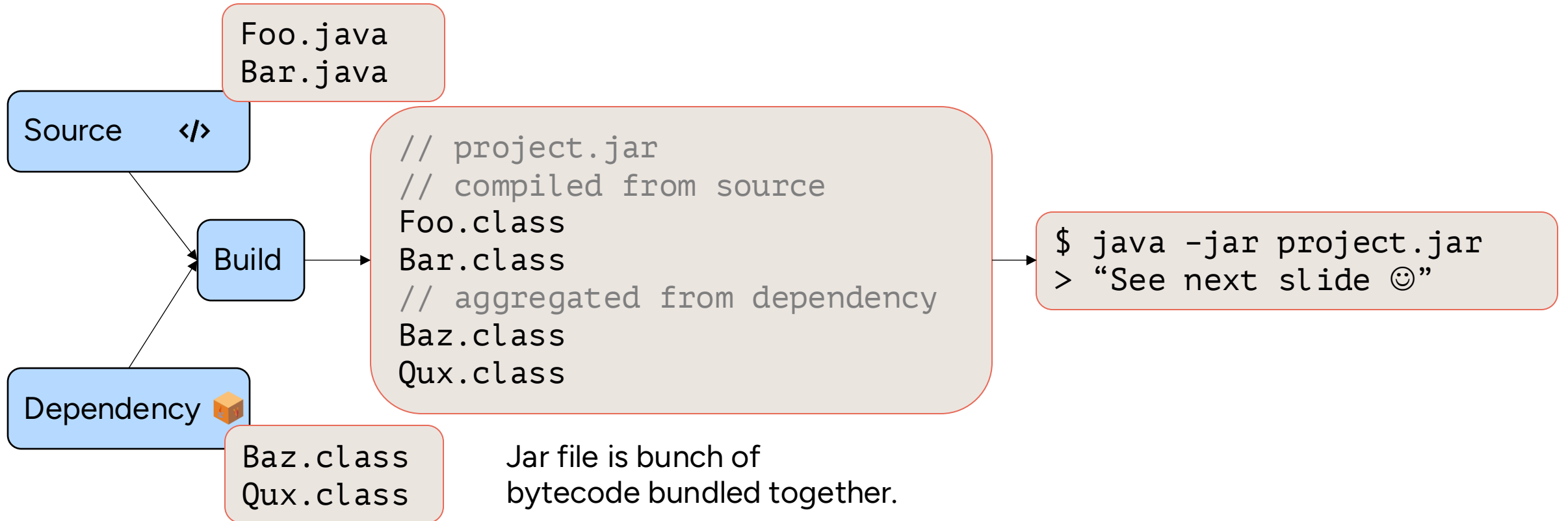
NEXT GENERATION SOFTWARE SUPPLY CHAIN ATTACKS (2019-2023)



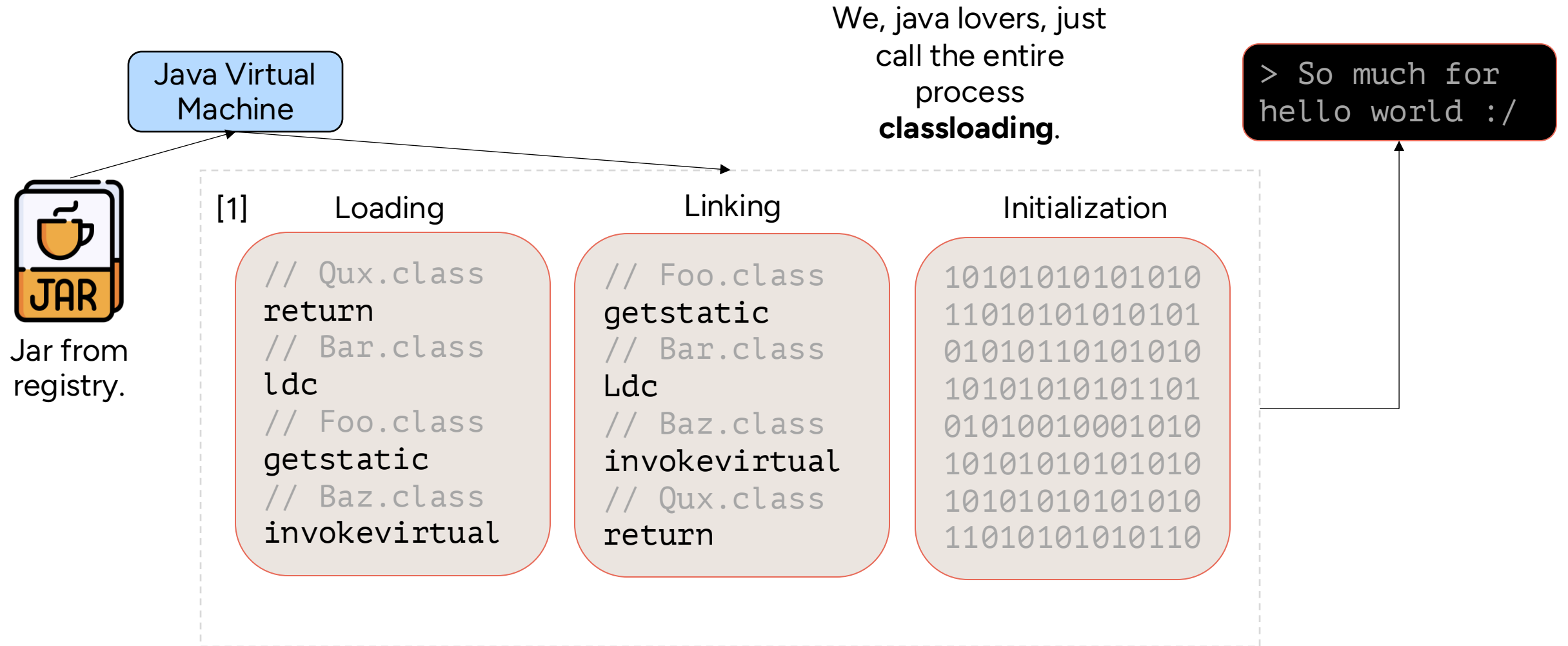
mathjs-min
GitGot XZ Polyfill.io
log4shell
Dependabot peacenotwar
Fake

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

Intricacies of Java: Build

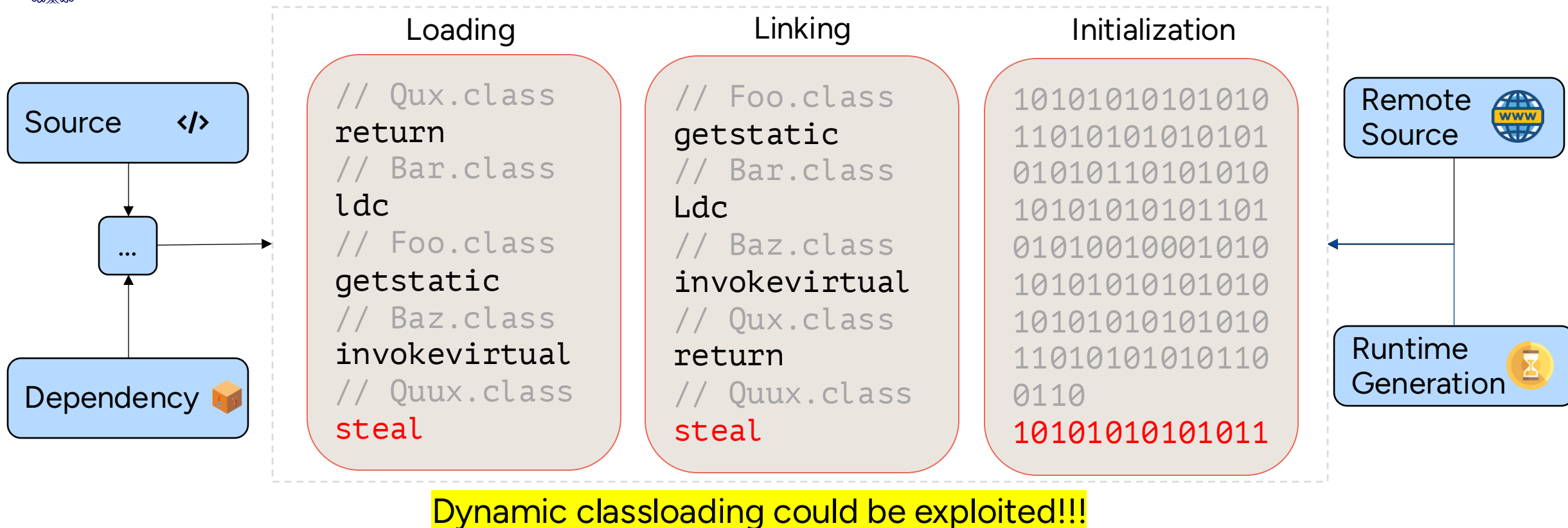


Intricacies of Java: Runtime



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

How is Software Supply Chain Attack relevant?



- Code can be downloaded at runtime.
- Code can be generated at runtime. [5]

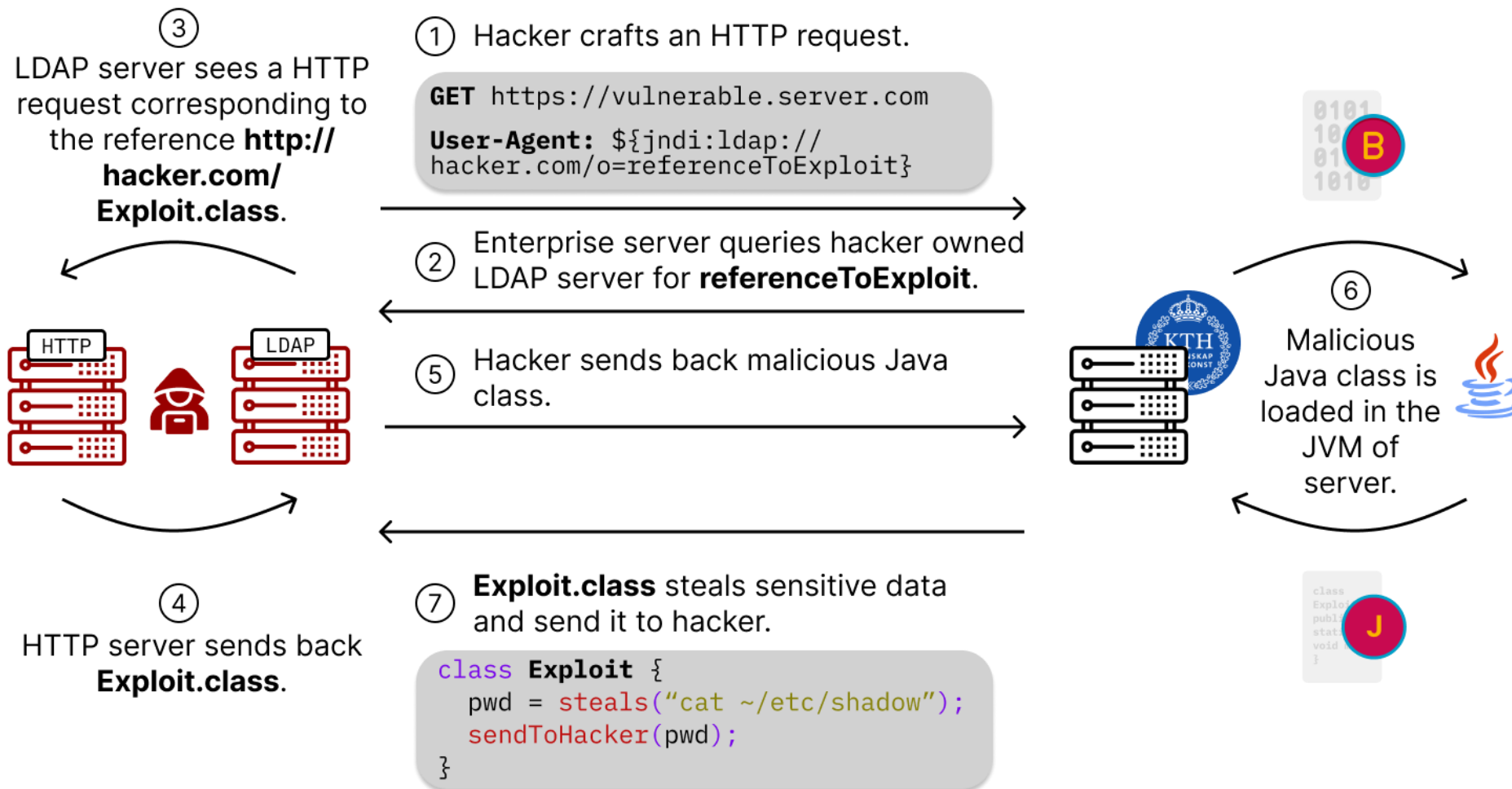
[5] Oracle, [ClassLoader \(Java SE 21 & JDK 21\) \(oracle.com\)](https://docs.oracle.com/en%2Fjava%2Fjavase%2F21%2Fdocs%2Fapi%2F%2F/java.base/java/lang/ClassLoader.html#builtinLoaders), 2023,
<https://docs.oracle.com/en%2Fjava%2Fjavase%2F21%2Fdocs%2Fapi%2F%2F/java.base/java/lang/ClassLoader.html#builtinLoaders>

Demo: Exploitation

CVE-2021-44228 (Log4Shell)

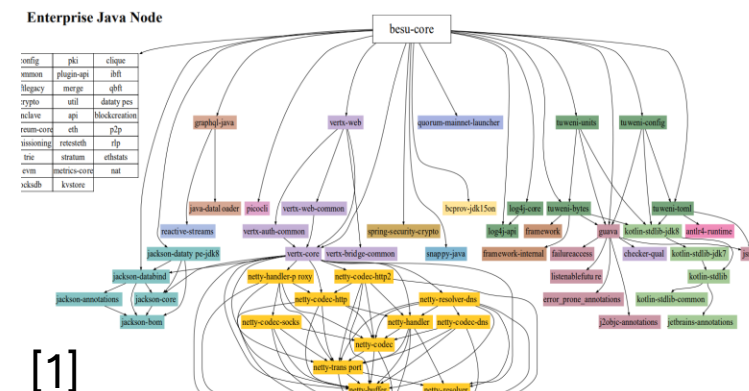
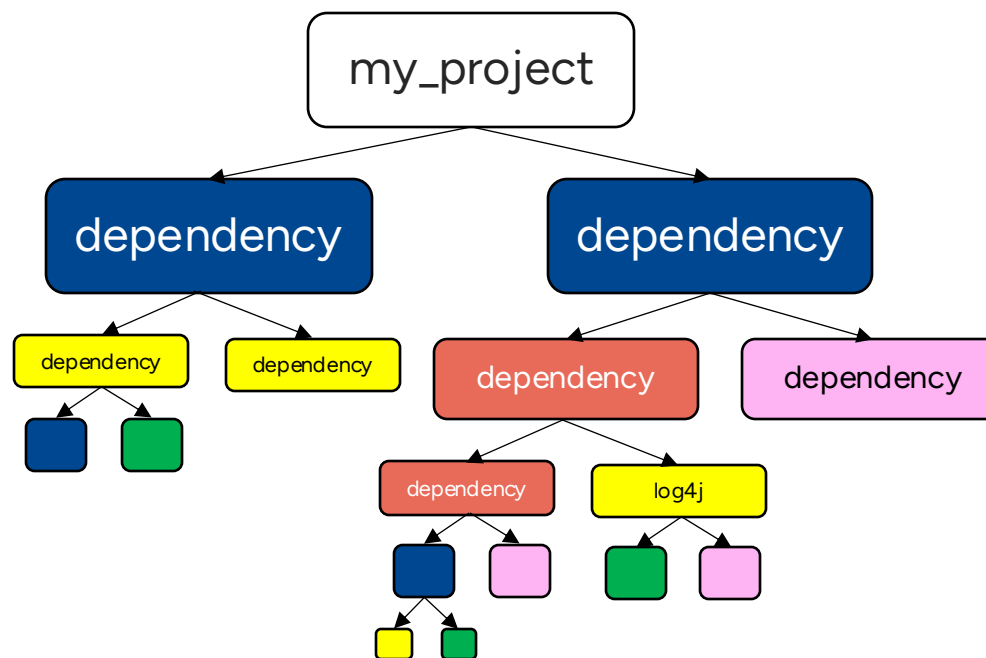
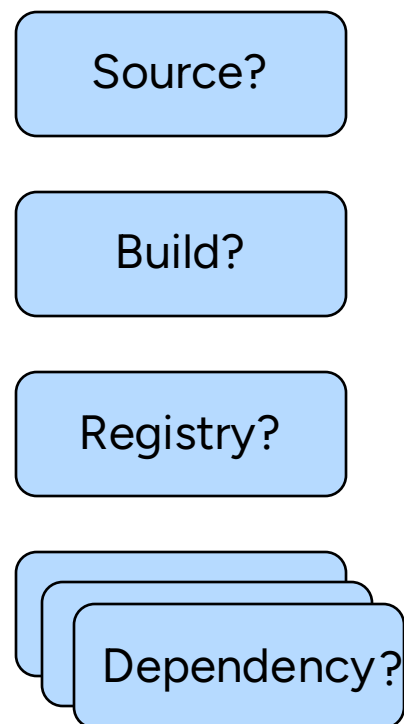
Source: <https://github.com/chains-project/exploits-for-sbom.exe/tree/main/rq2/log4shell-2021-44228>

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.



[1] C. Soto-Valero et al. "The Multibillion Dollar Software Supply Chain of Ethereum", Computer, 2022

Problem: Java can trigger download or generation of unknown code.

Let's look at real world examples.

Related Work: Permission Managers

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

[6] 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.

[7] 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.

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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.
 - [13] 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.
 - [14] 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.
 - [15] 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.

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

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 [2].

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.
```

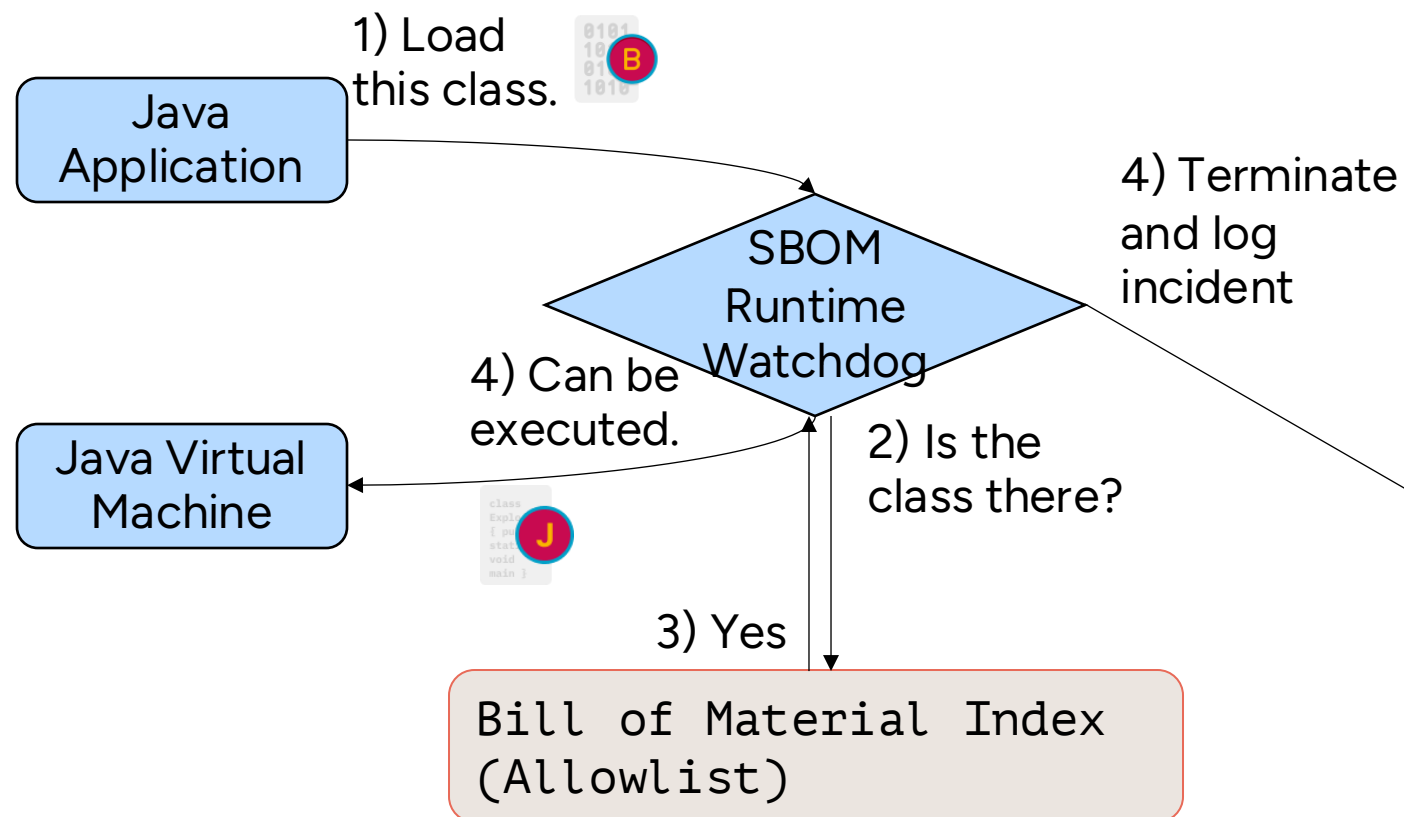
[2] L. Hutchison, Classgraph, GitHub.com, 2024

Part 1 done ✓

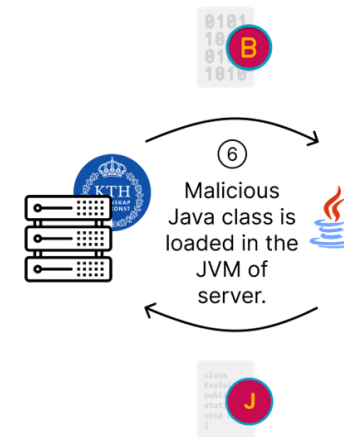
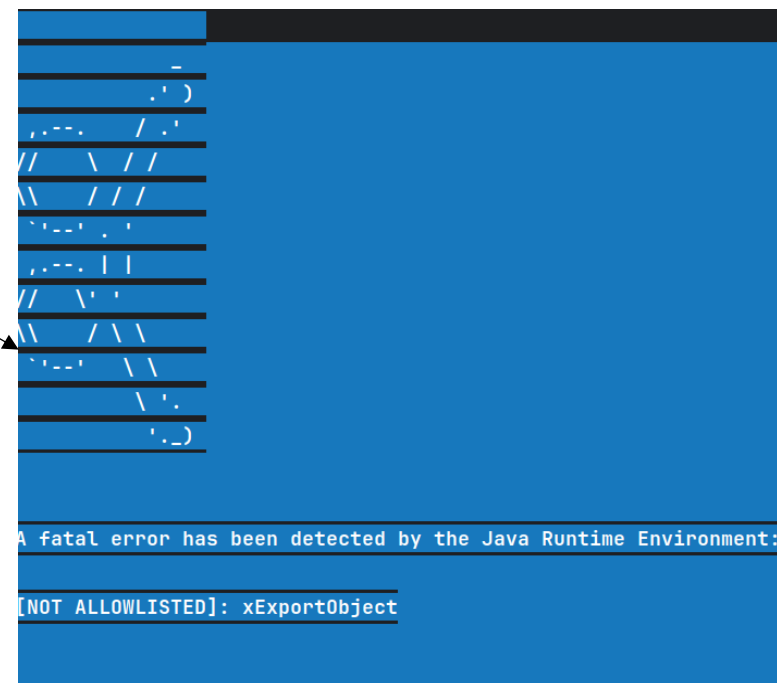
Step 2: Enforcement

Problem: Java class is simply loaded without any integrity.

Solution: We intercept loading and then verify it.



Part 2 done ✓



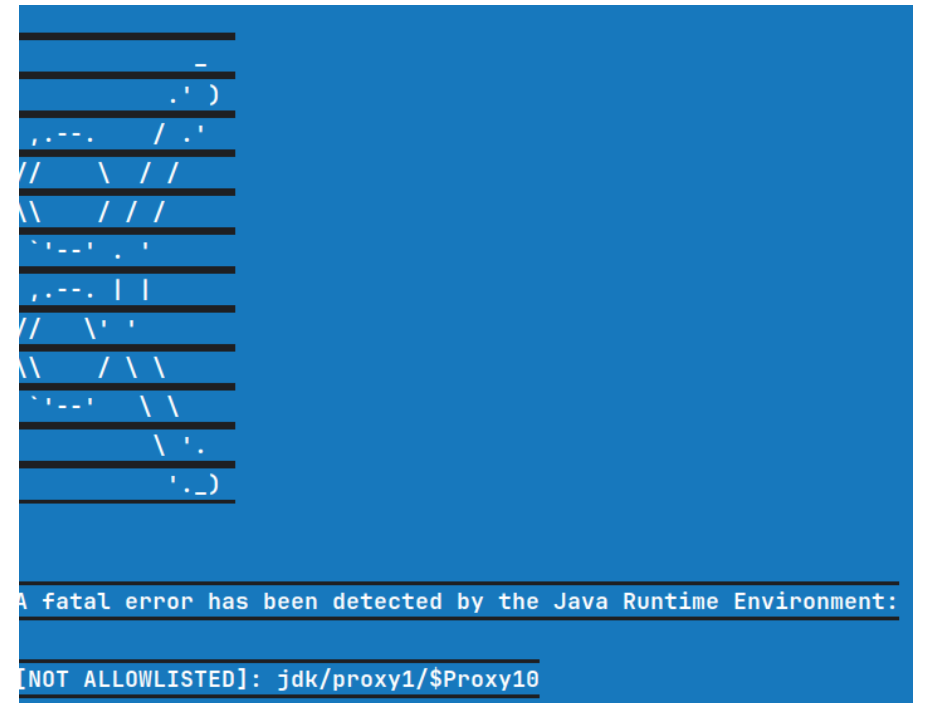
Test run

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.

A screenshot of a Java Runtime Environment error message. The background is blue. The text is white. The error message reads: "A fatal error has been detected by the Java Runtime Environment:". Below this, there is a line of text: "[NOT ALLOWLISTED]: jdk/proxy1/\$Proxy10".

```
-  
' )  
,--- / '  
// \ //  
\\ // //  
'--- '  
,--- | |  
// \ '  
\\ / \ \  
'--- \ \  
      \ '  
'..)
```

A fatal error has been detected by the Java Runtime Environment:

[NOT ALLOWLISTED]: jdk/proxy1/\$Proxy10

Bytecode Canonicalization

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

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

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

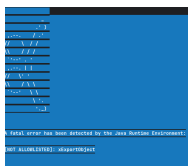
Solution: **SBOM Runtime Watchdog** is a novel tool to intercept Java classloading and verify integrity of each Java class.

Problem: non-determinism of Java bytecode.

Solution: **Bytecode Canonincalization**.

Problem: I miss blue screen of death in Linux ☹

Solution:



Demo: Mitigation

CVE-2021-44228 (Log4Shell)

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

Evaluation

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

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

Takeaway: Around 25,000 for the selected study subjects.

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:

Takeaway: Compatible with
real-world projects.

generated.

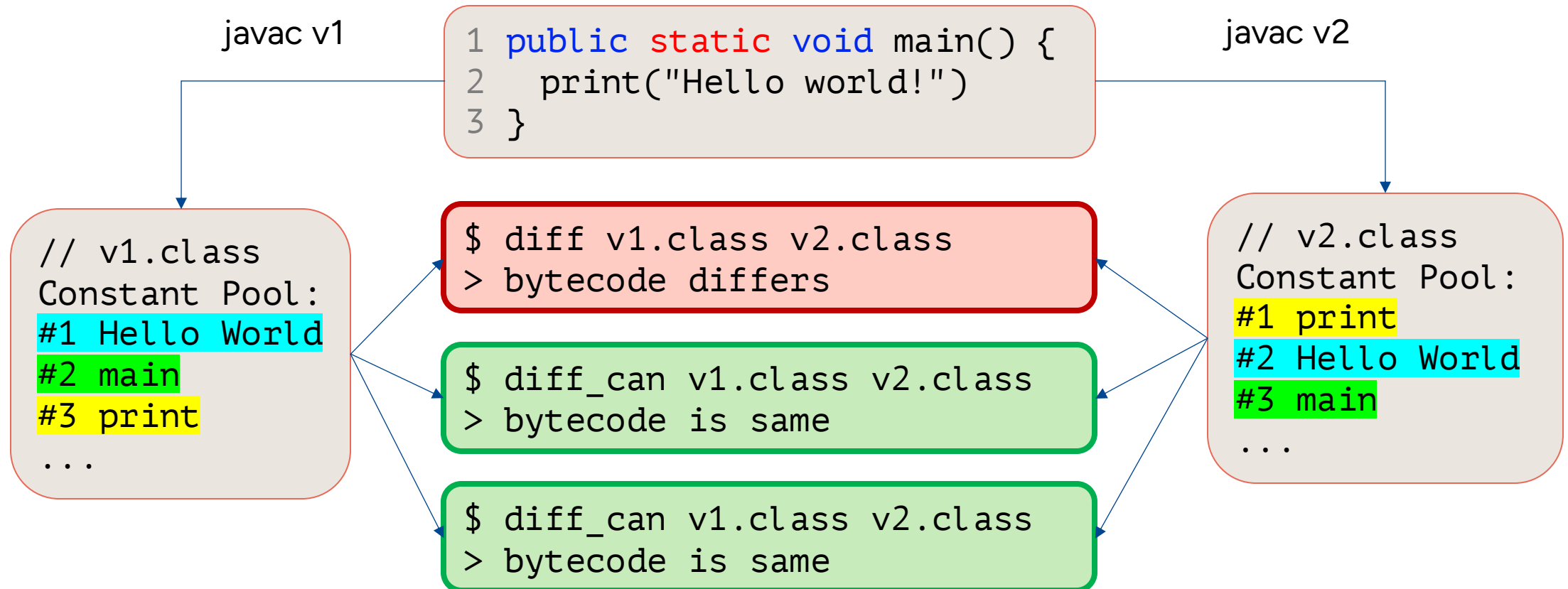
RQ4: What is the overhead of SBOM.EXE?

- 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).

**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

Key contribution: embeds
dependency information
in JVM bytecode.

+

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

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

=

We will be able
to tell what
dependency 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.



Personal
Webpage



<https://algomaster99.github.io>

Thank you!

Questions?

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](#)

Whitepaper



<https://arxiv.org/abs/2407.00246>