black hat USA 2023

AUGUST 9-10, 2023

BRIEFINGS

Reflections on Trust in the Software Supply Chain

Speaker:
Jeremy Long

black hat USA 2023

Jeremy Long
@ctxt/@ctxt.bsky.social

20+ years in security
Founder of OWASP Dependency-Check
Currently Principal Security Engineer @ ServiceNow

The Software Supply Chain is Massive

- "It has been estimated that Free and Open Source Software (FOSS) constitutes 70-90% of any given piece of modern software solutions."[1]
- **CI/CD Infrastructure and build management tools** are also modern software and are part of the supply chain
- **Third Party Services** used in the CI/CD are also modern software and are part of the supply chain

Targeting the Supply Chain

ENDPOINT SECURITY

MOVEit: Testing the Limits of Supply Chain Security

The need for cyber resilienc
measures are no longer end

By Torsten George
July 12, 2023

BREACH EXPLAINED

supply chain breach
step by step

le to
ive
s
se,

Double Supply Chain Compromise

17 Comments

Share

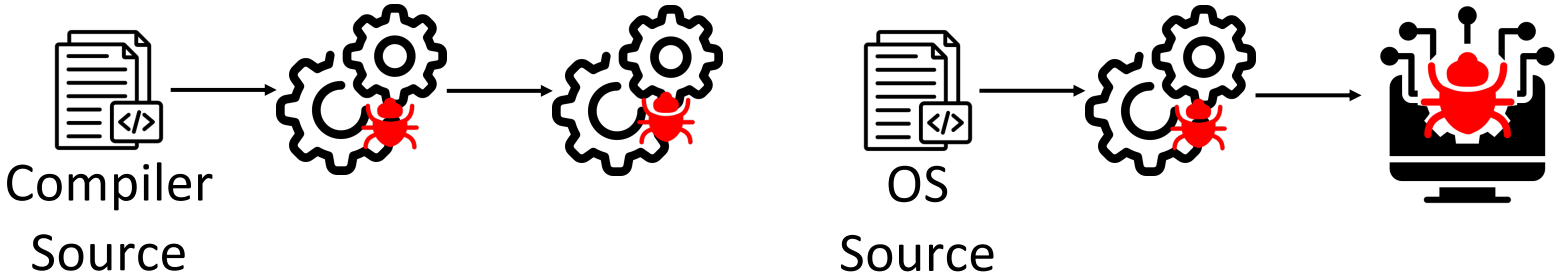
SolarWinds attack explained: And why it was so hard to detect

News Analysis
Dec 15, 2020 • 10 mins

Trend No. 3: Digital supply chain risk

Gartner predicts that by 2025, 45% of organizations worldwide will have experienced attacks on their software supply chains, a three-fold increase from 2021.

Reflections on Trusting Trust



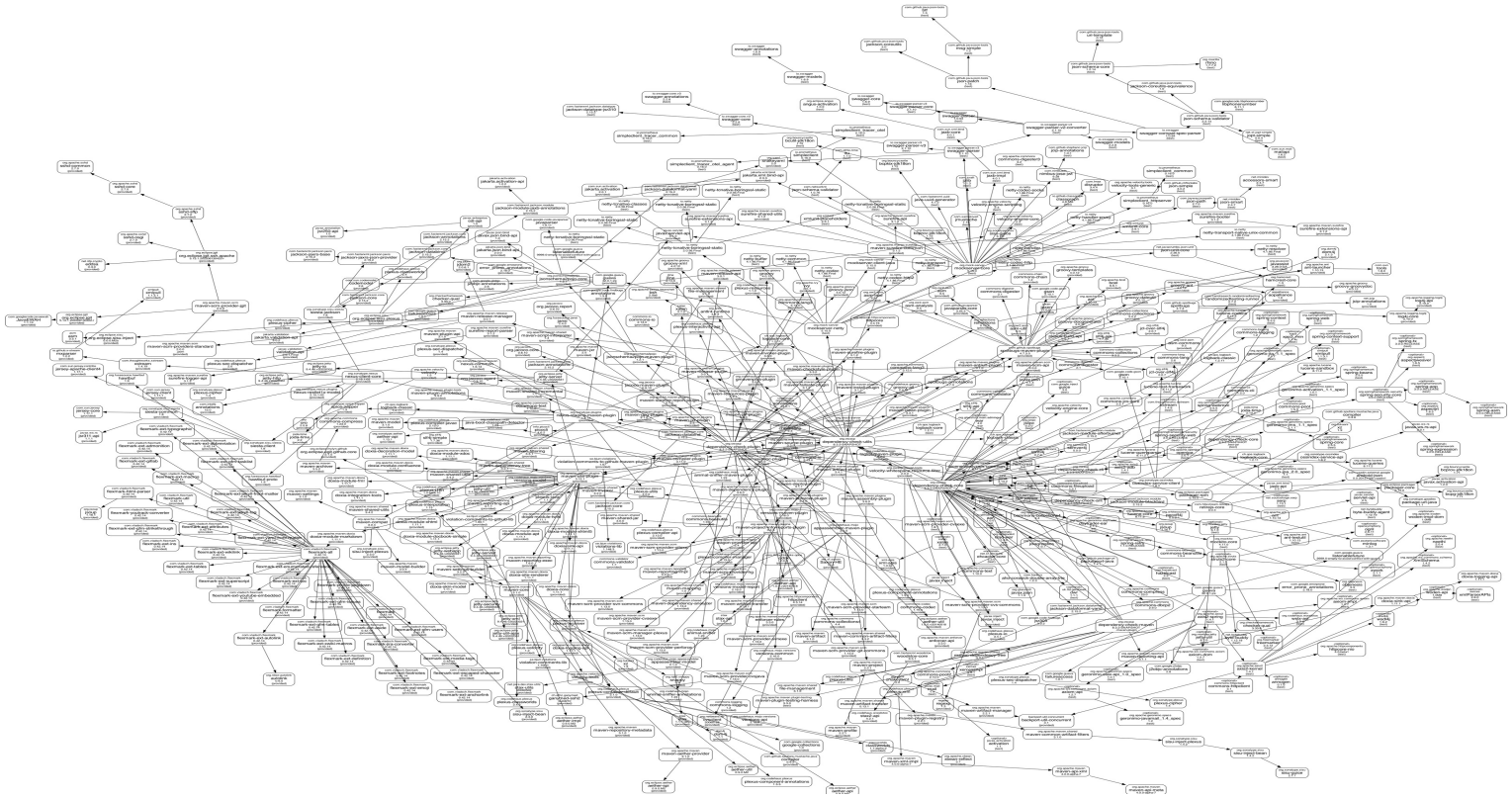
"The moral is obvious. You can't trust code that you did not totally create yourself."

-- Ken Thompson

Executive Order on Improving the Nation's Cybersecurity: Section 4 Enhancing Software Supply Chain Security

- Provenance of software code and components
- Software Bill of Materials (SBOM)
- Software Composition Analysis (SCA)

What is a dependency?



Industry Frameworks

- Supply-chain Levels for Software Artifacts, or SLSA ("salsa")
- Software Component Verification Standard (SCVS)

Provenance

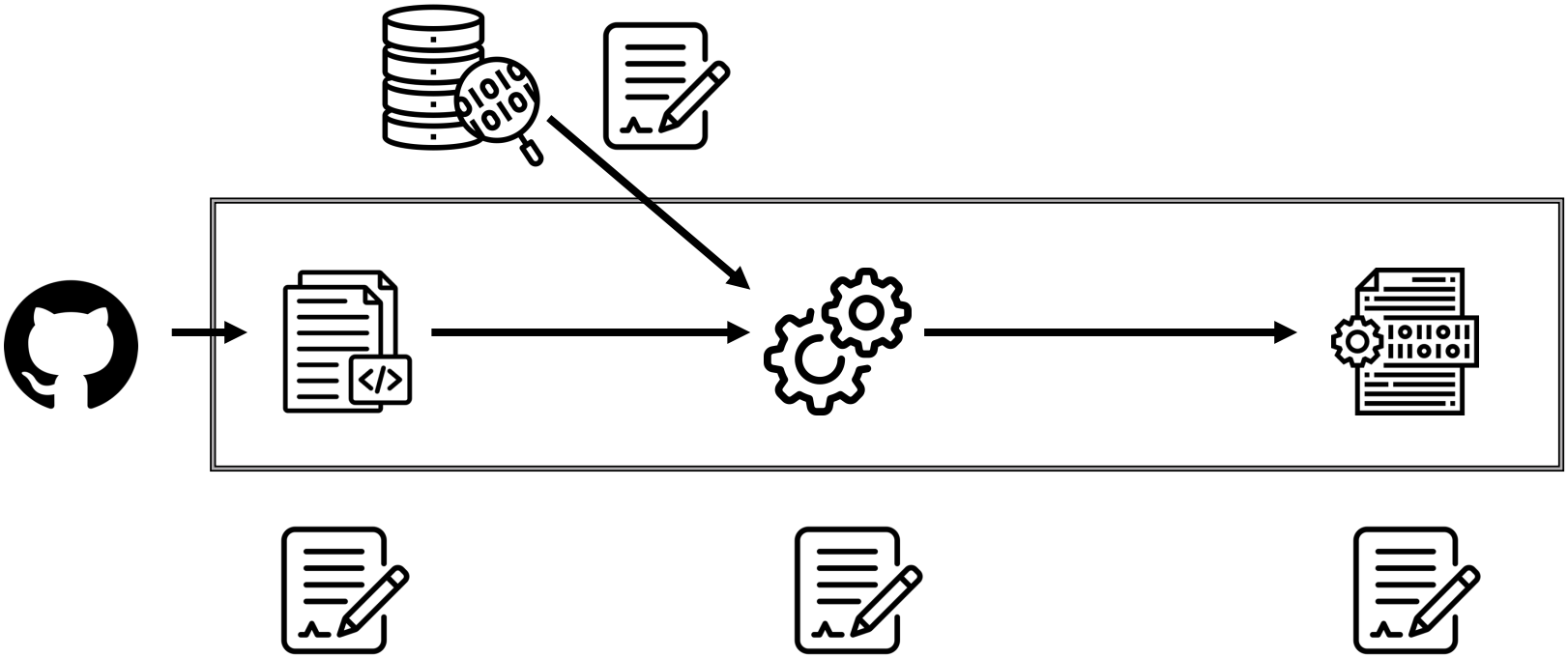
- SLSA Definition:

  Attestation (metadata) describing how the outputs were produced, including identification of the platform and external parameters.

- SCVS Definition:

  The chain of custody and origin of a software component. Provenance incorporates the point of origin through distribution as well as derivatives in the case of software that has been modified.

Provenance



SLSA v1.0 - Threats

Use a compromised runtime dependency

- **Threat**: The adversary injects malicious code into software required to run the artifact.
- **Mitigation**: N/A - This threat is out of scope of SLSA v1.0.

Use a compromised build dependency

- **Threat**: The adversary injects malicious code into software required to build the artifact.
- **Mitigation**: N/A - This threat is out of scope of SLSA v1.0, though the build provenance may list build dependencies on a best-effort basis for forensic analysis.

Software Composition Analysis (SCA)

- Analyze dependencies for known vulnerabilities
- Runtime dependencies are analyzed
- Build plugins and test dependencies?
- SCA tools that work at the repository level
- OWASP Dependency-Check
  - ✓ Maven Plugins
  - ✓ Gradle Plugins

Software Bill of Materials (SBOM)

- CycloneDX and SPDX
- Describes the runtime dependencies
- CycloneDX v1.5 introduced Manufacturing Bill Of Materials (MBOM)

Modern Supply Chain Attacks

Crash Override
@crashappsec

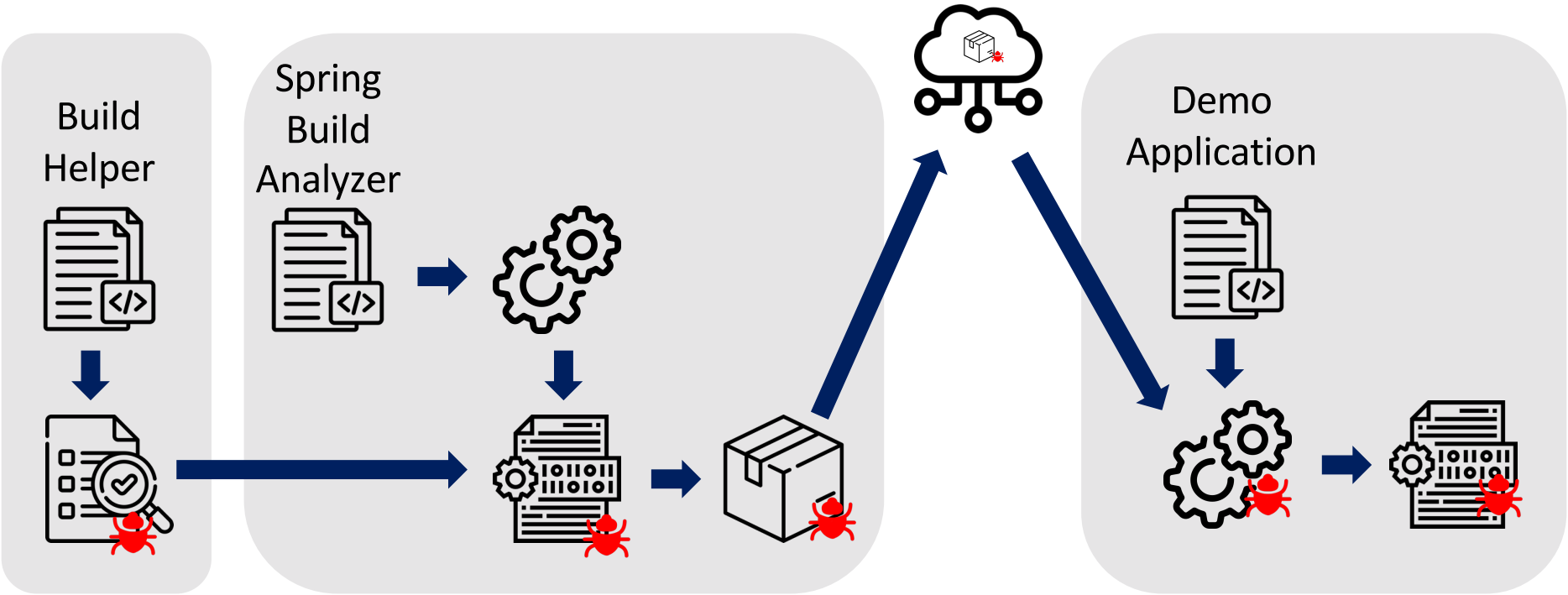
Worse than a bad logging library would be a backdoored library in a popular ide that then backdoored everything built with it. Vulnerable open source with a supply chain attack vector. Now there is a nightmare.

3:12 PM · Dec 11, 2021

Malicious Dependencies

https://github.com/jeremylong/malicious-dependencies

Demo Explanation



Injecting Malicious Code @ Build Time

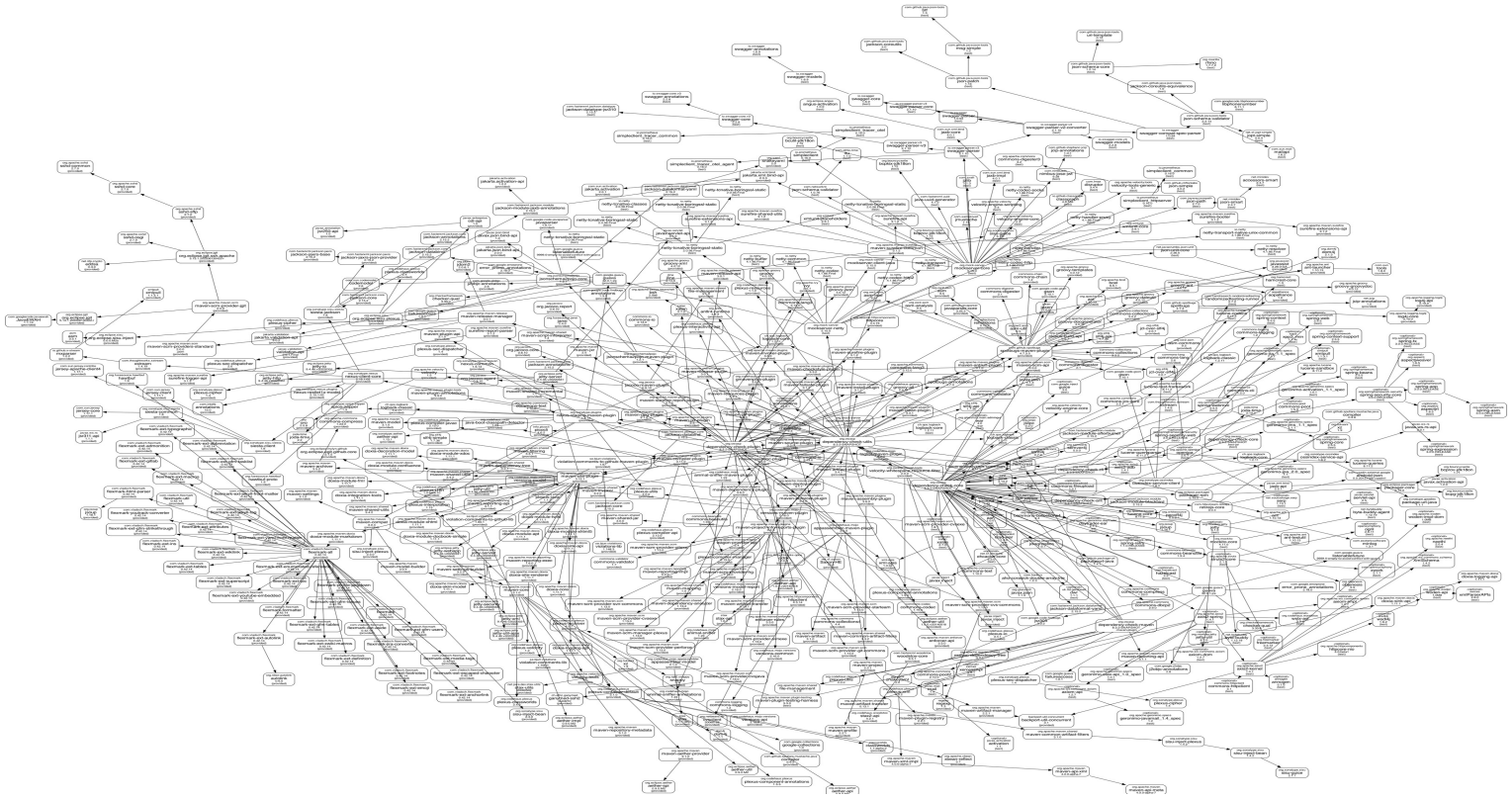
- Not limited to Java
- Build Plugins: Maven, Gradle, Poetry, etc.
- Testing Frameworks: JUnit, NUnit, Mocking Frameworks
- Gradle/Maven Wrapper

SLSA v1.0 - Threats

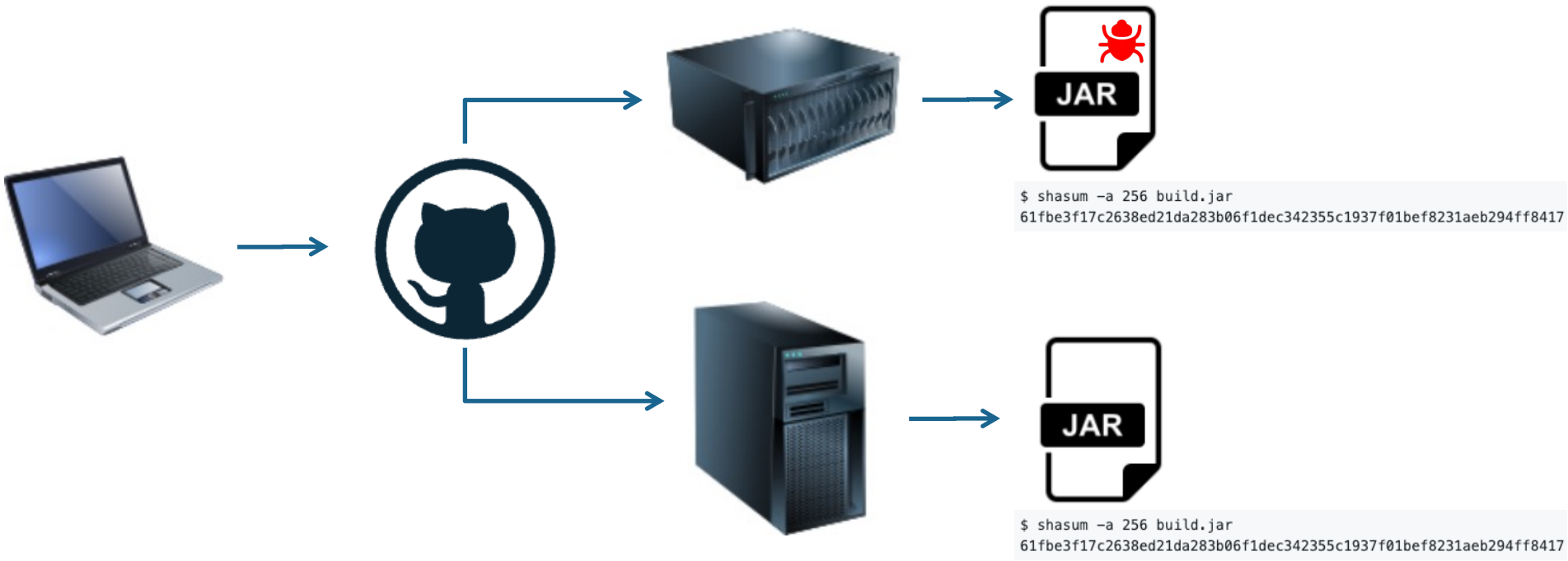
Use a compromised runtime dependency

- **Threat**: The adversary injects malicious code into software required to run the artifact.
- **Mitigation**: N/A - This threat is out of scope of SLSA v1.0. You may be able to mitigate this threat by pinning your build dependencies, preferably by digest rather than version number. Alternatively, you can apply SLSA recursively, but we have not yet standardized how to do so.

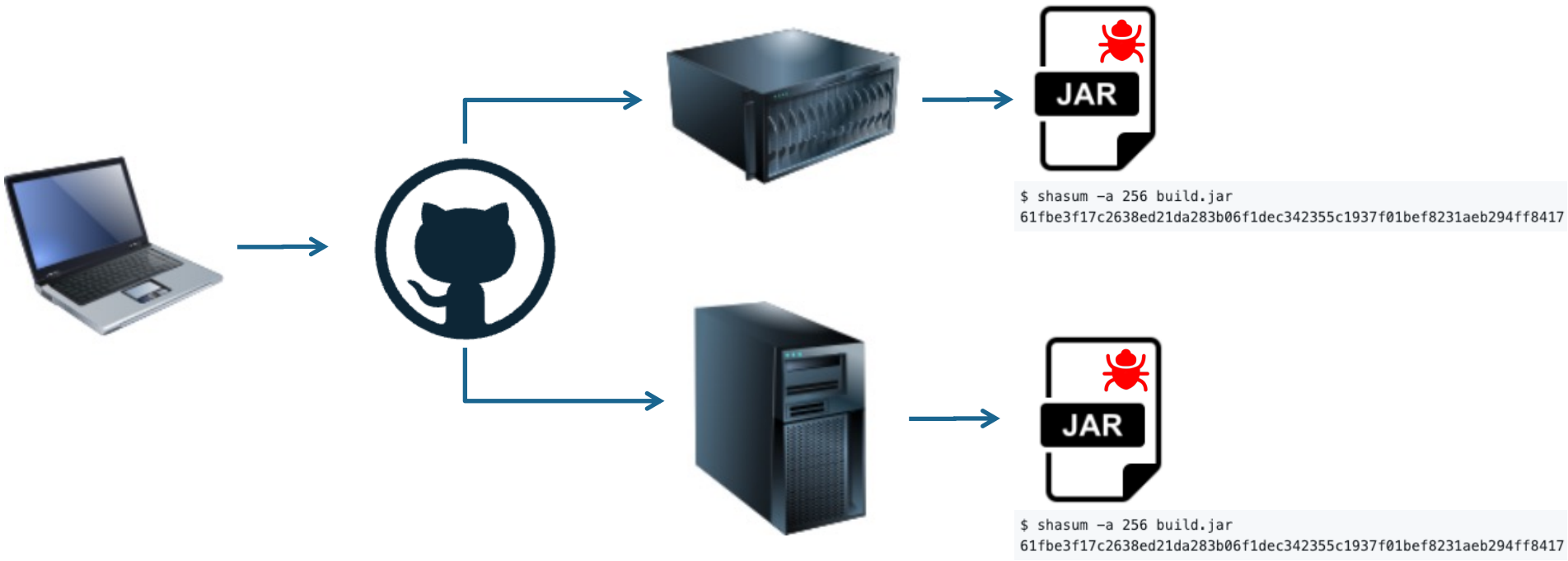
Apply SLSA Recursively



Reproducible Builds



Reproducibly Compromised Build



Vulnerable vs Malicious

binary-source validation

```
public class HtmlUtil {
  public String bold(String c) {
    return String.format("<b>%s</b>", c);
  }
}
```

compile

CLASS

binary-source validation: source model

```
public class HtmlUtil {
  public String bold(String c) {
    return String.format("<b>%s</b>", c);
  }
}
```

Class: HtmlUtil
+ Method: bold
  - args: String
  - constants: "%s”
  - called:
    - java.lang.String.format

Java Class Files

HtmlUtil.class

```
  0  CAFEBABE 00000037 00170A00 03000E08 000F0700  ....    7
 20  100A0011 00120700 13010006 3C696E69 743E0100              <init>
 40  03282956 01000443 6F646501 000F4C69 6E654E75   ()V    Code    LineNu
 60  6D626572 5461626C 65010004 626F6C64 01002628  mberTable    bold   &(
 80  4C6A6176 612F6C61 6E672F53 7472696E 673B294C  Ljava/lang/String;)L
100  6A617661 2F6C616E 672F5374 72696E67 3B01000A  java/lang/String;
120  536F7572 63654669 6C650100 0D48746D 6C557469  SourceFile    HtmlUti
140  6C2E6A61 76610C00 06000701 00093C62 3E25733C  l.java         <b>%s<
160  2F623E01 00106A61 76612F6C 616E672F 4F626A65  /b>    java/lang/Obje
180  63740700 140C0015 00160100 0848746D 6C557469  ct             HtmlUti
200  6C010010 6A617661 2F6C616E 672F5374 72696E67  l    java/lang/String
220  01000666 6F726D61 74010039 284C6A61 76612F6C     format   9(Ljava/l
```

binary-source validation: class model

Class: HtmlUtil
+ Method: bold
  - args: String
  - constants: "%s”
  - called:
    - java.lang.String.format

binary-source validation: Comparison

Class: HtmlUtil
+ Method: bold
  - args: String
  - constants: "%s”
  - called:
    - java.lang.String.format

Class: HtmlUtil
+ Method: bold
  - args: String
  - constants: "%s”
  - called:
    - java.lang.String.format

binary-source validation: Comparison

Class: HtmlUtil
+ Method: bold
  - args: String
  - constants: "%s”
  - called:
    - java.lang.String.format

Class: HtmlUtil
+ Method: bold
  - args: String
  - constants: "%s",
    "echo 'Never gonna give you up'"
  - called:
    - java.lang.String.format
    - java.lang.Runtime.getRuntime()
    - java.lang.Runtime.exec()

Binary Source Validation Challenges

- Compiler changes/optimization
- Code generators
- Model generation from a build artifact is technology specific
  - May limit the types of comparison that can be done

What can we do today?

- Reduce the number of dependencies
- Do not use code generators during the build
  - Generate code and check it into your source repo
  - Treat generated code as you do any other code
- Talk to your SAST and Supply Chain Vendors about build verification

Summary

- The trusting trust problem is real very real
- Any code running during the build can affect the build output - **reproducibly**
- Use OWASP Dependency-Check to scan plugins for maven and gradle builds
- Support open-source developers

Questions?