

Investigation of Java Language Security

Past and present

Olof Magnusson

August 10, 2025



Contents

1	Introduction	4
2	Background	5
3	Java language security	6
3.1	Stack inspection and access control	6
4	Attacks	10
4.1	Type confusion	10
4.2	Buffer overflow	11
5	Defenses	12
5.1	Static and dynamic code analysis	12
5.2	Fuzzers	12
5.3	Software-based reference monitors	14
5.4	Proof-carrying code	15
5.5	Static certification	16
6	Experiment	17
6.1	Implementation	17
6.2	Depedency-check module	19
7	Discussion	23
7.1	Java and access-control	23
7.2	Language-based defenses	23
8	Conclusion	25

Abstract

This project investigates the Java language with a focus on stack inspection to study attacks and protection mechanisms in the course TDA602 at Chalmers University. We demonstrate the fundamental architectural functionality by describing permission models, security policies, and access control which serve as the standard mechanism for searching the runtime call stack. We continue to study the current landscape by analysing state-of-the-art attacks and defenses. The findings show that type confusion is still a problem in Java by looking into the recent CVEs, and it is suggested that more applied research is needed in this area. We also demonstrated the most common protection mechanisms and how they can be applied to mitigate particular vulnerabilities in the protocol. The topic was discussed with security developers, which provided their insight and thoughts on the different mechanisms in performance and security. One clear observation was that language-based security is not widely discussed if we compare it to the other domains of computer security. We believe organizations focus their attention on traditional defense mechanisms like antivirus, firewalls, and intrusion detection systems. The problem is that some security vulnerabilities might not be detected because the mechanisms focus on signatures and prediction capability. However, reference monitors appear to be a lightweight approach to monitoring and injecting security checks in untrusted software. The experiment of this project shows that SonarQube could be a viable tool to scan for potential issues in production code. Cross-checking code by running maven dependency with CVEs gives a portion of assurance that the committed code does not include classified vulnerabilities from the community. Finally, the project has allowed for a deeper understanding of language-based security and how to argue more precisely about security properties.

1 Introduction

Java is a general-purpose programming language developed by Sun Microsystems that advocates security, memory management, portability, object-oriented principles, and fewer implementation dependencies, regardless of the underlying computing architecture. These features let developers Write Once, Run Anywhere (WORA), and have the ability to safeguard sensitive information without leaking information through loose typing or poor memory handling that has plagued programming languages like C and C++ for years.

The Java Security Architecture comprises several components; class loading, data typing, bytecode verification, and memory management, designed to elaborate a security model to assure integrity and confidentiality in a computer program. The Java Virtual Machine (JVM) [1] is an abstract instruction-set architecture that, under compilation, enforces security guarantees by performing runtime checks on the stack to prevent improper execution in the Java environment. JVMs are widely deployed in backend and frontend applications where code arrives from both trusted (local) and untrusted (remote) address spaces. To support this capability, the Java language needs a way of distinguishing code coming from these sources. Applications are subject to security policies that determine the restrictions of operations (read/write) and memory accesses to create objects.

Access control is the central component of the architecture with the responsibility for monitoring and delegating access to sensitive resources [2]. Stack inspection policies like the sandbox model enforce protection domains in order to control requests to resources and make security decisions by examining the runtime call stack. The security manager mediates access control and separates trusted and untrusted applications. The principle behind the implementation is to create a defense-in-depth mechanism to protect the java applets from malicious code [3]. The mechanism would approve all code from local sources but restrict all requests from untrusted sources. In Java 1.2, Security Manager was redesigned to enforce rigid policies on all code by applying the principle of least privilege. All code was considered untrusted by default and subjected to policies that limit access to resources. The java applet API was deprecated in Java 1.7 with the ambition to remove the security manager for future releases [3].

Thinking beyond the original sandbox model is a necessary step to investigating a more flexible architecture compared to the limited sandbox model. The former model made access-control decisions based on trusted and untrusted classes by monitoring requests to system resources. Java Access Control (JAC) architecture evolved this model by expressing more fine-grained security models [4]. The three mechanisms: permission checking, privileged execution, and permission contexts, embody the fundamental principles of the architecture. Confidentiality is obtained through enforcing custom mechanisms with aspectized access control that gives assurance that applications can use different security policies without conflicts, regardless of the underlying software architecture. Ulfar Erlingsson and Fred B. Schneider [5] had similar observations about improving the Java security architecture using custom enforcement mechanisms

like inline reference monitors. The mechanisms assure that the security policies can be tailored to applications that requires separate policies to protect and ensure reliable access to information.

Twenty years of Escaping the Java Sandbox have shown security vulnerabilities, and performance issues with the sandbox model [6]. Furthermore, the security manager has not been supported for years and has been scheduled to be deprecated in Java 18. This provides an opportunity for analysing and discussing alternative approaches. The project was developed during the course TDA602 at Chalmers University to survey access control and, more concretely, investigate the stack inspection mechanism used in Java. The paper is structured as follows: Section 2 gives an overview and work in access control and application security. Section 3 will introduce Java Language Security with Java stack inspection and access control. Section 4 and 5 will discuss attacks and possible defenses of Java Security. Section 6 will demonstrate the experiment. Finally, section 7 will sum up the findings and discuss future directions together with section 8.

2 Background

Over the years, access-control mechanisms have served as powerful techniques to enforce control by granting or denying processes from reading, and writing to files in computer systems [7, 8]. There have been earlier studies on access-control mechanisms that allows the ability to specify security properties based on calculus to protect objects from information manipulation. Code access security relies on algorithms that analyse execution traces in code to mediate control with read/write operations, class interactions, method invocations, and other essential constructs that might transmit data. Confidential information needs to be released to correct processes, and policies need to enforce soundness without compromising performance in the communication [9].

A significant contribution in object-oriented communications was developed using confinement rules to protect data from unauthorized modifications [10]. Lampson observed that the integrity of sensitive processes could be preserved using a controlled environment to distinguish between trusted and possibly other unwanted programs with technical safeguards that control access to system resources. However, information release is a necessity in many areas of computing systems [11, 12], and rules must be defined so that integrity can be assured through safe declassification using security policies [13].

Biba observed techniques for sharing information between resources in complex information systems. The ability to test properties in a program using integrity models gives a portion of assurance that privileged information is protected from improper modifications [14]. The integrity aspect focuses on controlling the flow of components in a program from being modified by inappropriate sources. If there are insufficient checks during runtime, we risk releasing too much information to an attacker that could use that to its advantage.

Stack inspection policies have essential capabilities in programming lan-

guages by creating a sandbox environment that distinguishes application code through enforcement of access control when there is code on the stack frame trying to access sensitive resources [2]. The inspection policies divide a security model into two classes – trusted and untrusted. Trusted types belong to local sources on the system and are allowed to request secret (confidential) information from the operating system without heavily enforced restrictions. Untrusted classes can read public information on computer systems but are restricted from accessing information of higher security classification. When requests arrive from untrusted sources to information classified as confidential, the system must adequately validate the code before being given access. This abstraction enforces soundness in computer systems, where each process gets assigned to the correct security level.

Attackers are in today’s environments more creative and are shifting the attack pattern to target the behavior of the applications. Defenders must show similar creativity to protect systems against malicious code by investigating new mitigation techniques. It is, therefore, of interest to explore the security architecture in Java and discuss the attacks and defenses.

3 Java language security

In this section, we will provide information related to Java Security to understand the essential background of stack inspection by describing permission models, security policies, and access-control methods with relevant examples. Finally, section 4 will expand upon this type of architecture by describing attacks and language-based defense strategies.

3.1 Stack inspection and access control

Java stack inspection algorithm facilitates access-control logic and embodies four primitives that make the underlying architecture more robust. By examining the runtime stack, the security manager can determine decisions by either granting or discarding requests from local and remote address spaces. Although commonly, vendors use different terminology and name conventions, the algorithms fundamentally boil down into four operations: `enablePrivilege()`, `disablePrivilege()`, `checkPrivilege()` and `revertPrivilege()`. We borrow ideas and concepts from previous studies on stack inspection [2, 3, 15] and discuss heavily based on three components:

- **Permission model:** The permission tag is one property in the Java language used to enforce stricter policies to access an object. Consider the code statements:

```
FileInputStream fileInput
= new FileInputStream("home/chalmers/cse");
FilePermission TDA602
= new FilePermission("home/chalmers/cse", "read");
AccessController.checkPermission(TDA602);
```

FilePermission class consists of a pathname and a set of operations. The operations are the computations that can be performed on an object – that is, read, write, execute and delete. The current permissions of an object can be visualised by implicit call to `TDA602.getAction()` for a canonical string representation. The security manager mediates access control and delegates responsibility to `AccessController` to prevent improper access to objects. Any code that attempts to access a sensitive system resource must be granted by `AccessController` based on the security policy in effect. The `AccessController` will return data quietly upon success or throw `AccessControlException` if insufficient permission is set to access a object.

The algorithm used by the `AccessController` to determine if access should be permitted or not is given below from the Java language specification. The current thread is traversed m callers in sequential order.

```

    for (int i = m; i > 0; i--) {

        if (caller i's domain does not have the
            permission)
            throw AccessControlException

        else if (caller i is marked as privileged) {
            if (a context was specified in the call to
                doPrivileged)
                context.checkPermission(permission)
            if (limited permissions were specified in
                the call to doPrivileged) {
                for (each limited permission) {
                    if (the limited permission implies
                        the requested permission)
                        return;
                }
            } else
                return;
        }
    }

    // Next, check the context inherited when the thread
    // was created.
    // Whenever a new thread is created, the
    // AccessControlContext at
    // that time is stored and associated with the new
    // thread, as the
    // "inherited" context.

    inheritedContext.checkPermission(permission);

```

- **Security Policy:** An object is created during runtime, and the security policy determines if a code should be permitted to access a sensitive object. The object is stored in a simple file, database, or another structure that caches authorized information. For any code attempting to access a sensitive object, the system needs to call `checkPrivilege(TDA602)` before allowing it to execute code on the current stack frame. The `AccessController` examines the runtime call stack. When code search is executed and a frame containing a privilege tag is detected on the current stack frame, the search reaches termination and the operation is dispatched. If the code initiates additional operations, the procedure `enablePrivilege(TDA602)` needs to be called. The local policy gets consulted and determines if the caller is permitted to use TDA602. Upon termination, the stack frame annotation needs to either be concealed or removed. The procedure `disablePrivilege(TDA602)` hides previous enabled privilege and `revertPrivilege(TDA602)` removes the stack annotations from the current stack frame. The method invocation `refresh(TDA602)` enables the object to be refreshed by calling the source of the policy data. To illustrate how it works, assume that we have a policy with the origin **www.chalmers.se** that is signed with **TDA** that has both read and write permission in **/home/chalmers/cse**. A request from a class signed by **TDA** arrives in the local system, and the Java runtime system tracks permissions. Access-control decisions are then made based on the runtime call stack.

```
grant codebase "http://chalmers.se",
    signedBy "CSE",
    principal javax.security.auth.x509.X509Principal "cn
=Dough" {
    permission java.io.FilePermission "/home/chalmers/
cse", "read, write";
};

//Fetching the java-security policy
    System.getProperty("java.security.policy"));

    //Permission checked, seems OK!
System.out.println("Writing to file " + file + " OK");

    // Else
catch (java.security.AccessControlException e) {
    System.out.println("Writing to file " + file + "
ACCESS DENIED");
}

catch (Exception e) {
    System.out.println("Writing to file " + file + "
EXCEPTION");
}
```


- **Access Control Enforcement:** The Java runtime keeps track of the sequence of Java calls using the security manager that handles control and access based on the security policy in effect. Any code that requests to execute operations on an object must first be marked as privileged by invoking `enablePrivilege(TDA602)`. When the code is loaded into Java runtime, the class loader associates the information with the following i) the environment code was loaded from, ii) the entity that signed the code, and iii) default permissions to make a decision. We extend the previous code by adding additional statements. Control checking algorithm `doPrivileged` assures that the application calls the function when there is code on the stack frame trying to access files on the local system. In addition, sensitive objects can be slow checked by `GuardedObject g = new GuardedObject(fileInput, TDA602);`.

```
FileInputStream fileInput
= new FileInputStream("home/chalmers/cse/lbs.tex");

FilePermission TDA602
= new FilePermission("home/chalmers/cse/lbs.tex", "write");
AccessController.checkPermission(TDA602);

try {
    Access.Controller.doPrivileged(TDA602)
    // protection domain
    write(sensitive_file)
}
catch(AccessControlException ace) {
    //If the callee function do not have permission }
```

`ProtectionDomain` class groups objects together in order to communicate and exchange information safely. The domain usually falls into two categories; system domain and application domain. The main purpose of the system domain is to protect against inappropriate access on the system level and only allow known devices to access resources. The application domain focuses more on the application level, for example, when an application tries to invoke a method that calls a sensitive process in the system. The protection domain is decomposed of three parameters `codesource`, `permissions`, `classLoader` and `principals` which are the array of principals associated with the domain. The code below creates a new `ProtectionDomain` based on the previous constructs and protects an object reference so it cannot be forged. Another view of this can be visualised in Figure 1.

```
public ProtectionDomain(CodeSource codesource,
                        PermissionCollection permissions,
                        ClassLoader classloader,
                        Principal[] principals)
```

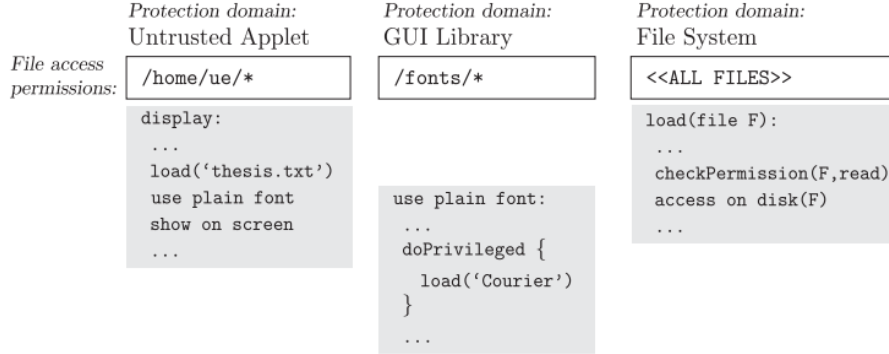


Figure 1: Three Protection Domains. [5]

4 Attacks

The sandbox mechanism relies on that the type system is soundly enforced and that malicious code can not bypass the sandbox environment to cause undesired effects. We will analyse state-of-the-art language-based attacks relevant to access control and stack inspection.

4.1 Type confusion

A language with type-safe components prohibits direct access to memory locations, assuring that the operation does not break the object's encapsulation. Type casting is done through implicit calls, and pointers are correctly tagged with the correct references. CVE reports have over the years shown that type confusion is still one of the biggest challenges in the Java language [16, 17, 18, 19, 20] and the primary strategy attackers use to bypass the sandbox environment. The attack targets the access-control logic of the software to confuse the memory handler about the data objects it is manipulating. The security manager believes it is of object type A when it is actually of type B. To demonstrate how it works, consider the code:

```
public class A {
    SecurityManager x;
}
public class B {
    MyObject x;
}
p1;
p2;

p1.x = System.getSecurityManager();
MyObject object = p2.x;
```

Processes must call the security manager to invoke permission before attempting any sensitive operations. However, in this example, the Java sandbox can be evaded by changing fields of the type `MyObject` by pointing it to something else. Missing checks in the code may result in a crafted executable that can be placed on the current stack frame and produce an unexpected result. William Bonnaventure et al. observed that 76% of the Java releases from 1.6 to 14 were affected by serious security threats, suggesting that the attack can fully bypass sandbox mechanisms [21, 15].

4.2 Buffer overflow

Attacks that try to exercise function boundaries have a lower probability of happening than type confusion if we look into CVE statistics over the last few years. Most likely, the reason is the Java embedded functionalities that prevent malicious data harvesting to a certain extent. However, specific attacks have still bypassed the security mechanism and successfully exploited the memory capabilities of the JVM. CVE reports [22, 23] have shown the impact of out-of-bounds read vulnerabilities and special constructed packets that trigger vulnerabilities in the implementation that bypass the sandbox mechanism. The attacks often result in memory disclosure within the process and are primarily used to confuse the memory handler of the system. Inconsistencies in design or implementation are a growing issue in the industry where the security enforcement mechanism is either misconfigured or lacks capabilities to protect systems against specific attack vectors. Attackers make assumptions about the system by understanding relevant information through stack traces, improper access to sensitive classes, methods, variables, and poorly enforced access-control policies. To demonstrate this fact, we choose to pick one example from Nils Emmerich [24], especially the CVE-2020-2803. Consider the code:

```
public ByteBuffer slice() {
    int rem = this.remaining();
    return new HeapByteBuffer(hb,
        -1,
        0,
        rem,
        rem,
        this.position() + offset);
}
```

The method has some issues with multithreading – based on the fact that if the position is changed between `remaining()` and `position()` invokes from the value 0 to the value `limit()`, then both `remaining()` and `position()` will return `limit()`. This sequence will lead to an out-of-bound read vulnerability and trigger an `ArrayIndexOutOfBoundsException` because that `HeapByteBuffer` is backed by a Java array [24].

5 Defenses

Protecting information from inadvertent disclosure or modifications has been an issue for decades, although more mechanisms have been researched and developed to satisfy code access security [25]. However, information released to undesirable sources is still an ongoing problem, and mechanisms in Java that analyse code behavior to facilitate decisions have been shown unsatisfactory in terms of performance and security in the later years. Therefore, moving beyond file-read permissions and the ambiguous programming model is necessary, focusing on application-level solutions to gain insight and strengthen control in remote code execution [26]. Defenses less sluggish in software and support code access security are therefore desirable to research. We focus on the four general ways of defending against malicious code: analyse, rewrite, monitor and audit.

5.1 Static and dynamic code analysis

A first potential line of defense is using static code analysis and building an abstraction of runtime state [27]. The technique attempt to find potential problems in non-running code and highlight possible unwanted code behavior. However, this approach is challenging in choosing the right abstraction level and is often tedious and error-prone when analysing large models. Moreover, this technique might not be suitable for applications requiring precision [28]. On the other hand, dynamic analysis observes runtime code executions with procedure calls, methods, statements, and computed values [29]. However, this technique requires a good test suite and more knowledge of the parameters to measure but generally provides higher test coverage. There has been research that has investigated and compared these approaches [30, 31, 32] to analyse the state transitions of programs and in terms of security. However, there are tools like SonarQube [33] that perform static and dynamic code searches to check code quality, including security, which we will study in more detail in this project.

5.2 Fuzzers

Fuzzers create awareness of code executions that may affect a system’s confidentiality and privacy properties [34, 35]. In simplistic terms, fuzzers are the science of automatic bug finding by essentially feeding random data into a program or stack to discover derivations in code executions. The fuzz vectors can contain multiple arguments that allow checking string properties, integers, chars, or binary values. Approaches likeClazzFuzz [36], and ClassMing [37] are state-of-the-art fuzzers that deploy differential testing by targeting the Java parser. The idea behind differential testing is that it tests and compares two different JVMs to detect any signs of derivations in the implementations. However, some clear limitations to these approaches are that it does not see any vulnerabilities related to API calls. Another consideration is that if two JVMs contain similar vulnerabilities, they will yield the same result. The implementations heavily rely on user-defined test seeds that limit the ability to detect type confusion

vulnerabilities. Although the techniques serve essential characteristics, they are ineffective in detecting malicious code. The difference between a Java Program fuzzer and a JVM Fuzzer can be visualized in Figure 2.

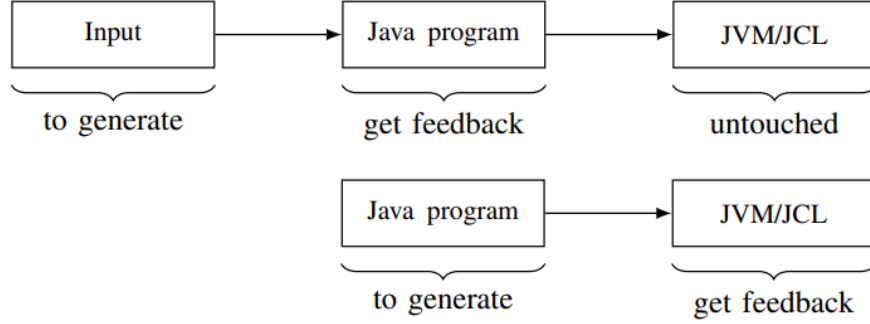


Figure 2: Java Program Fuzzer (top) and JVM Fuzzer (bottom) [21]

To tackle this challenge and enforce soundness, CONFUZZION [21] is a recent approach that has been developed to target security-relevant object-oriented flaws with a particular focus on type confusion vulnerabilities. The testing technique automatically generates Java programs that trigger JVM vulnerabilities through mutations by testing every possible JVM method call combination. The architecture of Confuzzion can be shown in Figure 3, which specifies the process of generating programs, adding mutations and objects inside the master program to provide test feedback with search guidance. Although the approach is new and requires more testing to facilitate a practical implementation, it shows a promising way to detect type confusion or related object-oriented vulnerabilities within a reasonable amount of time.

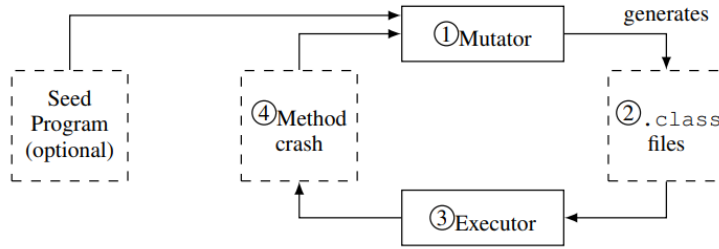


Figure 3: Confuzzion Architecture [21]

5.3 Software-based reference monitors

Inline-reference monitors in Java are a defense mechanism that has been researched for quite some time [38, 39, 40, 41, 42, 43] and provide capabilities for reducing the JVM footprint that is crucial for constrained computing environments. The IRM facilitates frameworks to construct a logic for monitoring, auditing, and controlling data subjects.

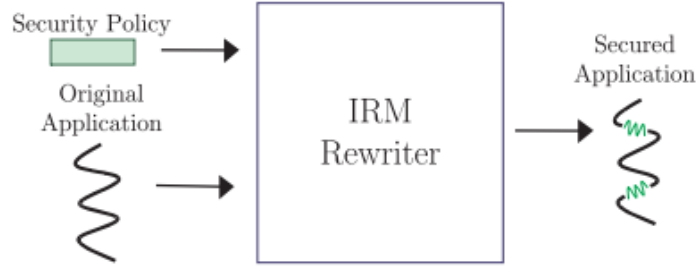


Figure 4: IRM approach to security policy enforcement. [5]

The instrument enforces dynamic technical safeguards that observe code executions of the program to detect any signs of security policy violations [5]. In principle, the policies are specified in a formal language that defines rigors and depth in how principals can access the system and use resources. The security mechanism performs the initial checks by looking into the policy integrated into the binary. These checks should not be possible to circumvent. A security policy in SASI can be visualised to demonstrate some of its functionalities in Figure 5. The policy specification is based on finite-state automata with the security operations read and write in a computer network [44]. The policy assures that no sends are allowed to be performed after a file read during program initialization [45]. When a read occurs, the instrument monitors this action and changes its state to the noSnd state. This security automation allows every state except a send to be executed. However, there is no valid transition out from noSnd, and it represents, in this case, a security policy violation [45].

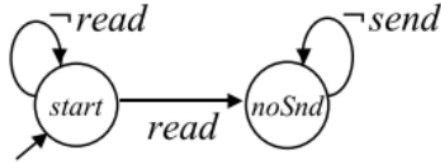


Figure 5: Security automation of no send after read [44]

PSlangs security policy can be visualised in below that only allows 10 open windows at the same time [5].

```

IMPORT LIBRARY Lock;
ADD SECURITY STATE {
  int openWindows = 0;
  Object lock = Lock.create();
}
ON EVENT begin method
WHEN Event.fullMethodNameIs("void java.awt.Window.show()")
PERFORM SECURITY UPDATE {
  Lock.acquire(lock);
  if (openWindows = 10) {
    HALT["Too many open GUI windows"];
  }
  openWindows = openWindows + 1;
  Lock.release(lock);
}
ON EVENT begin method
WHEN Event.fullMethodNameIs("void java.awt.Window.dispose()")
  ")
PERFORM SECURITY UPDATE {
  Lock.acquire(lock);
  openWindows = openWindows - 1;
  Lock.release(lock);
}

```

Although some information leaks must be allowed in computer systems like password checking, we can reduce the JVM footprint and enforce soundness with IRM policies. The security mechanism assures that a write does not affect a sensitive read before releasing the information to a unknown process.

5.4 Proof-carrying code

Proof-carrying code has been around for some time [46], although it suffered some criticism due to performance issues in the past. However, the technique has recently grown in popularity [47, 48, 49] by verifying code in various software applications. The principle behind its implementation is to provide assurance that remote code adheres from the correct code producer. One consideration to keep in mind is that the code producer has to define the security policy through rigorous specifications and distribute it publicly to the clients [50]. The formal proof, along with the narrative code, assures that the code obeys the safety policy. This suggests that the mechanism can fully validate each binary without needing an additional security mechanism.

This process shown in Figure 6 sets, however, requirements of the proof checker that defines the policy. First, the security policy must be developed based on certain sound assumptions to define security requirements. Then, if the security policy is ineffective, the policy needs to be rewritten and republished.

Finally, the code producer needs to redevelop the policy and safety proof for the new specification.

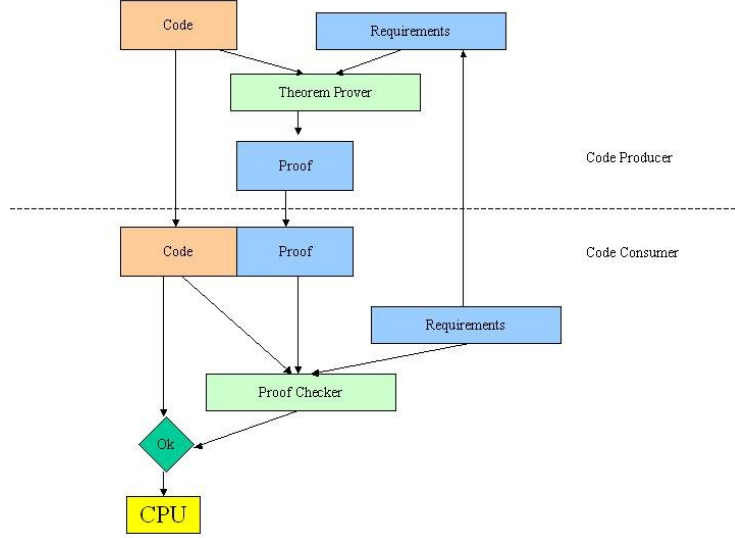


Figure 6: An example of start-to-finish with verification of Proof-carrying code [51]

There are both advantages and disadvantages by using this approach. One could argue that code can be assumed trustworthy by looking into the information specified in the certification header. For example, suppose there is a mismatch between code and proof during proof checker validation. In that case, the operation will be detected, and the code will be rejected before being allowed to access the system. The disadvantage is that the mechanism does not solve the problem if there is any modification before transmission. One example is an unintentional mistake from the developer with the wrong type declaration of a class. A simplistic example is when we have a sensitive class that is not declared final. This design flaw allows the creation of subclasses that could cause undesired effects when finally executed from the host.

5.5 Static certification

A defense mechanism focused on implicit flow analysis is static verification that ensures computer code follows security requirements [52]. Each program must be statically verified as secure with appropriate semantics before being initialized and executed. One that serves attractive characteristics is Jif which extends the Java programming language to support information flow control and enforce access control at runtime [53]. Static information flow control supports end-to-end security to safeguard the confidentiality and integrity properties of the program. After verifying programs and assuring that the variables are not

being manipulated inappropriately by the computing environment, it can be compiled and executed as ordinary Java code. To illustrate how it works, we demonstrate an example from the manual. Consider the label `int Alice→Bob x`; this statement tells principal Alice controls that information in `x` and that Alice permits principal Bob to read the information in `x`. The information in `x` is hence permitted to be affected by principal Bob. Jif implements the logic operations around this label to assure confidentiality and integrity properties in code. In addition, the Jif have certain language features that are important to protect confidential information from improper misuse:

- **Declassification** mechanism that release information whenever it is necessary.
- **Principals** that supports authentication and authorization procedures
- **Statically checked access control** that enables writing authentication procedures without information leaking
- **Principal hierarchy** that can be expressed with defined groups and user-roles
- **Label polymorphism**: allows the code to be executed with respect to the corresponding security class.
- **Principal polymorphism** allows the code to be executed with respect to the corresponding principals.
- **Automatic label inference**: avoids the need to write many type annotations
- **Run-time label checking and first-class label values**: allows the possibility to define new policies a run time. Run-time checks ensure that information is not leaked during failure or success of the execution.

More information about this technique can be found in the manual [53].

6 Experiment

6.1 Implementation

In order to facilitate a practical implementation, we will use simple code and evaluate the components using the tool SonarQube with maven, which uses static code analysis to explain why a particular code is harmful. Sonarqube identifies internal problems such as reliability, security, and maintainability. The tool is beneficial in identifying security vulnerabilities, which may be overlooked while coding. For example, figure 9 shows that a vulnerability has been detected concerning the class `FileReader`, suggesting how to fix the issue with try-with-resources. However, it provides additional capabilities like detecting legacy code

if the right dependency-check module is used to scan the directory. The CWE-456 and CWE-772 were triggered in this case.



Figure 7: Random code tested that shows potential issue with not wrapping try-with-resources

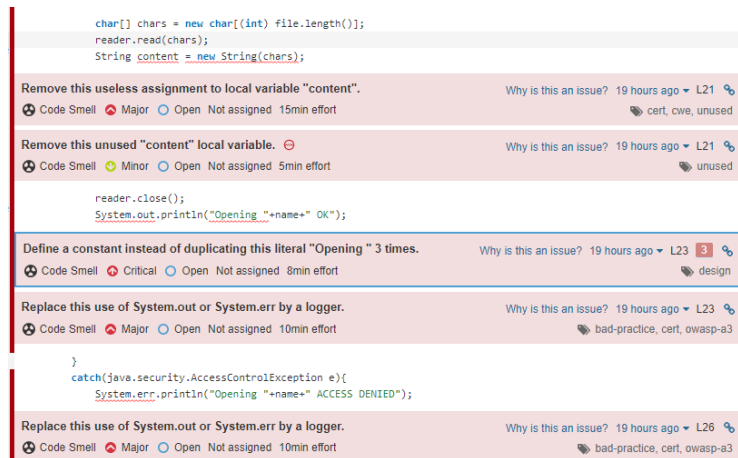


Figure 8: Code smells

To give another example, we run static analyses on the code provided in the Wallet class in Lab1 and analyse its output.

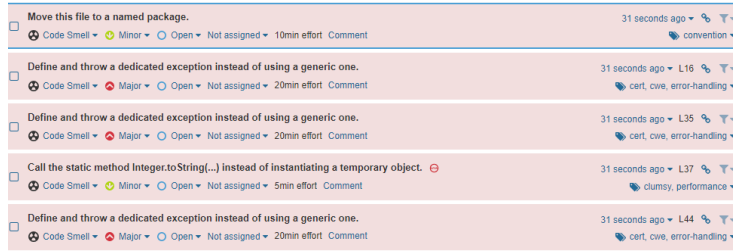


Figure 9: A potential vulnerability detected using generic exceptions that could reveal sensitive stack trace information to an attacker

Clearly, we observe problems regarding exception handling that could have undesired side effects in the application by leaking sensitive information. The problem with throwing generic and broad exceptions is that it makes it easier for callers to anticipate what could go wrong in the application and, therefore, craft code that can circumvent the system’s security. In the course, we learn that it is better to be precise with exceptions whenever possible to better control the information flow in the application. Therefore, the CWE-397 [54] was triggered, which covers the declaration of throws for generic exceptions. Now that we know how it works, it is time to make this more exciting and evaluate code with several known vulnerabilities. The project we choose to display has Java web common vulnerabilities and security code based on spring boot, and spring security [55].

6.2 Dependency-check module

The implementation uses dependency modules, and by running the following command `mvn org.owasp:dependency-check-maven:7.0.4:check` essentially means that we will download the OWASP dependency-check Maven plugin that contains a list of known vulnerabilities to crosscheck against the code. However, for this first case we choose to enable all common CVE dependencies. The method collects information about files it scans using Analyzers. Each CVN entry has a list of the vulnerable software, as shown in the example code snippet below.

```
<entry id="CVE-2012-5055">
...
  <vuln:vulnerable-software-list>
    <vuln:product>cpe:/a:vmware:
      springsource_spring_security:3.1.2</vuln:product>
    <vuln:product>cpe:/a:vmware:
      springsource_spring_security:2.0.4</vuln:product>
    <vuln:product>cpe:/a:vmware:
      springsource_spring_security:3.0.1</vuln:product>
  </vuln:vulnerable-software-list>
...
</entry>
```

The analysis starting phase of the code repository can be shown in Figure 10, where it begins to process the code using certain embedded features in the maven build. When the processing is done, the identification phase of vulnerabilities can be seen in Figure 11 with the overall result of the execution in Table 1.

```
[INFO] Analysis Started
[INFO] Finished Archive Analyzer (1 seconds)
[INFO] Finished File Name Analyzer (0 seconds)
[INFO] Finished Jar Analyzer (1 seconds)
[INFO] Finished Dependency Merging Analyzer (0 seconds)
[INFO] Finished Version Filter Analyzer (0 seconds)
[INFO] Finished Hint Analyzer (0 seconds)
[INFO] Created CPE Index (1 seconds)
[INFO] Finished CPE Analyzer (3 seconds)
[INFO] Finished False Positive Analyzer (0 seconds)
[INFO] Finished NVD CVE Analyzer (0 seconds)
[INFO] Finished RetireJS Analyzer (1 seconds)
[INFO] Finished Sonatype OSS Index Analyzer (3 seconds)
[INFO] Finished Vulnerability Suppression Analyzer (0 seconds)
[INFO] Finished Dependency Bundling Analyzer (0 seconds)
[INFO] Analysis Complete (13 seconds)
[INFO] Writing report to: C:\Users\Olofm\Desktop\java-sec-code-master\target\dep
endency-check-report.html
```

Figure 10: A part of the analysis of the repository

```
One or more dependencies were identified with known vulnerabilities in java-sec-code:
```

```
bcpkix-jdk15on-1.55.jar (pkg:maven/org.bouncycastle/bcpkix-jdk15on@1.55, cpe:2.3:a:bouncycastle:legion-of  
bcpv-proj-jdk15on-1.55.jar (pkg:maven/org.bouncycastle/bcpv-proj-jdk15on@1.55, cpe:2.3:a:bouncycastle:bouncy-c  
crypto_packages-1.55:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*:*  
: CVE-2016-1000343, CVE-2016-1000344, CVE-2016-1000345, CVE-2016-1000346, CVE-2016-1000347, CVE-2016-1000348, CVE-2016-1000349,  
commons-beanutils-1.9.3.jar (pkg:maven/commons-beanutils/commons-beanutils@1.9.3, cpe:2.3:a:apache:common  
commons-collections-3.1.jar (pkg:maven/commons-collections/commons-collections@3.1, cpe:2.3:a:apache:commo  
commons-httpclient-3.1.jar (pkg:maven/commons-httpclient/commons-httpclient@3.1, cpe:2.3:a:apache:common  
: CVE-2012-5785, CVE-2010-19956  
commons-io-2.5.jar (pkg:maven/commons-io/commons-io@2.5, cpe:2.3:a:apache:commons-io@2.5:*:*:*:*:*:*:*:*:*:*:*  
dom4j-1.6.1.jar (pkg:maven/dom4j/dom4j@1.6.1, cpe:2.3:a:dom4j_project:dom4j@1.6.1:*:*:*:*:*:*:*:*:*:*:*) CVE-20  
dom4j-2.1.0.jar (pkg:maven/org.dom4j/dom4j@2.1.0, cpe:2.3:a:dom4j_project:dom4j@2.1.0:*:*:*:*:*:*:*:*:*:**)  
fastjson-1.2.24.jar (pkg:maven/com.alibaba/fastjson@1.2.24, cpe:2.3:a:alibaba:libobama:1.2.24:*:*:*:*:*:*:*)*  
2-25845  
Fluent-hc-4.3.6.jar (pkg:maven/org.apache.httpcomponents/fluent-hc@4.3.6, cpe:2.3:a:apache:httpclient@4.  
groovy-2.4.7.jar (pkg:maven/org.codehaus.groovy/groovy@2.4.7, cpe:2.3:a:apache:groovy@2.4.7:*:*:*:*:*:*:*)*  
guava-23.0.jar (pkg:maven/com.google.guava/guava@23.0, cpe:2.3:a:google:guava@23.0:*:*:*:*:*:*:*)* CVE-2016-  
guava-23.0.jar (pkg:maven/com.google.guava/guava@23.0, cpe:2.3:a:google:guava@23.0:*:*:*:*:*:*:*)* CVE-  
hibernate-validator-5.3.4.Final.jar (pkg:maven/org.hibernate/hibernate-validator@5.3.4.Final, cpe:2.3:a:ja  
io:5.3.4:*:*:*:*:*:*:*)* CVE-2017-7536, CVE-2019-10219, CVE-2019-14900, CVE-2020-10693, CVE-2020-25638
```

Figure 11: Identified vulnerabilities in java-sec-code

sec:java-sec-code:1.0.0	
Dependency-check version:	7.0.4
Report Generated On:	28 Aug 2022
Dependencies Scanned:	194 (161 unique)
Vulnerable Dependencies:	56
Vulnerabilities Found:	342
Vulnerabilities Suppressed:	0
NVD CVE Checked:	2022-08-28 14:42:18
NVD CVE Modified:	2022-08-28 14:00:01
VersionCheckOn:	2022-08-28 14:42:18

Table 1: Result of the maven dependency execution

Type	Source	Name	Value	Confidence
Vendor	file	name	bcprov-jdk15on	High
Vendor	jar	package name	bouncycastle	High
Vendor	jar	package name	crypto	High
Vendor	jar	package name	jca	High
Vendor	jar	package name	provider	High
Vendor	Manifest	application-binary-allowable-codenames	*	Low
Vendor	Manifest	application-name	Bouncy Castle Provider	Medium
Vendor	Manifest	bundle-symbolicname	2.0.0-1.0, JavaSE-1.6, JavaSE-1.7, JavaSE-1.8, JavaSE-1.9	Low
Vendor	Manifest	bundle-symbolicname	bcprov	Medium
Vendor	Manifest	runtime-allowable-codenames	*	Low
Vendor	Manifest	codename	*	Low
Vendor	Manifest	extension-name	org.bouncycastle.bcpovider	Medium
Vendor	Manifest	implementation-vendor	BouncyCastle.org	High
Vendor	Manifest	implementation-vendor-id	org.bouncycastle	Medium
Vendor	Manifest	originally-created-by	24-95401 (Oracle Corporation)	Low
Vendor	Manifest	permissions	all-permissions	Low
Vendor	Manifest	specification-vendor	BouncyCastle.org	Low
Vendor	Manifest	system-binary	true	Low
Vendor	pom	artifactId	bcprov-jdk15on	High
Vendor	pom	artifactId	bcprov-jdk15on	Low


Figure 14: The amount of evidence found in the repository categorized either as file, jar, manifest or pom

Identifiers

- [pkg:maven/org.bouncycastle/bcprov-jdk15on@1.55](#) (Confidence:High)
- [cpe:2.3:a:bouncycastle:bouncy-castle-crypto-package:1.55:*:*:*:*:* \(Confidence:Low\)](#) [suppress](#)
- [cpe:2.3:a:bouncycastle:bouncy-castle-crypto-package:1.55:*:*:*:*:* \(Confidence:Low\)](#) [suppress](#)
- [cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle:1.55:*:*:*:*:* \(Confidence:Low\)](#) [suppress](#)
- [cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle-java-cryptography-api:1.55:*:*:*:*:* \(Confidence:Low\)](#) [suppress](#)
- [cpe:2.3:a:bouncycastle:the_bouncy-castle_crypto_package_for_java:1.55:*:*:*:*:* \(Confidence:Low\)](#) [suppress](#)

Figure 15: Identified vulnerabilities in java-sec-code

For instance, if we press the first link in the identifier column we get the more information about the vulnerability in question.



bcprov-jdk15on
 org.bouncycastle
 Version 1.55

[Report advisory or correction](#)

Vulnerabilities

7 HIGH
 7 MEDIUM
 2 LOW

Sign up and see:
 Detailed component information including:

- ✓ Version history
- ✓ Declared licenses
- ✓ Vulnerability details

Figure 16: Identification of the vulnerabilities

We got intrigued by this result and tried the same technique against production environment ready code, where we found vulnerabilities related to improper uses of classes and variables. Some of the most common issues were public variables that should be private, classes that should be declared final, and legacy methods that were not deprecated, most likely because of backward compatibility. We observed that some code leaked information through wrongly thrown exceptions and that some `inputHandlers` were wrongly configured. To summarise this experiment, the method seems to have the ability to catch dan-

gerous security flaws, but also the most basic in the code repository by using this code-dependency technique.

7 Discussion

7.1 Java and access-control

In this limited study, we have investigated the Java stack inspection with practical examples and explored the underlying access-control mechanism used to protect systems against malicious code by searching the caller stack. Some apparent issues we found throughout the study concerned performance and security with the separation of permissions. The past approach of granting or disallowing requests based on permission is inevitably a slow process, especially when over hundreds of authorization checks are processed [26]. In principle, sensitive information needs to be appropriately controlled; thus, having partial security for specific functions would be desirable in terms of performance. However, it is challenging due to architectural structure, as every call to access a privileged object needs to be validated appropriately [56]. There are still platforms and vendors relying on security managers for security purposes. Most Java security features have not been considered security manager for a long time, requiring developers to be extra cautious when designing software applications with these features enabled. For instance, the complexity of changing access-control policy to meet new custom-based solutions without affecting other permissions in the system is a challenging task. This procedure requires infrastructure modifications, as the virtual machine needs to be correctly aligned to meet the new requirements derived from the security policy as discussed in [5]. The ability to trace requests could still be a desirable property to use with the security manager. If we consider that we have several requests to sensitive file `x`, the security manager can represent who and how many attempts have been made to access that resource. However, the general conclusion is that non-java solutions are more desirable for production environments.

7.2 Language-based defenses

Language-based security allows for more precise definitions of security in computer systems. Effort in implementing sound methodologies is needed to distinguish secret from public information to ensure that the computing system manipulates variables appropriately [25, 14, 57, 58]. Static code analysis is undoubtedly a costly option but ensures to some degree that the code follows best practices with an expert's interpretation of the runtime state. Mistakes can be discovered and adjusted before execution but require the developer has complete knowledge about the domain, system, and computer security. In principle, it is suggested not to be the optimal setting. However, the technique could serve as the only option to fully sandbox the environment if we have confidential information that cannot be released without being heavily scrutinized. While the

other methods could make error predictions with higher accuracy, they would not satisfy this property because of this insecurity. Sonarqube is a popular way of analysing code in the industry and could assist developers to some degree with the quality of the code. The tool is already widely used in the industry and could detect bad code practices, code smells, and to some degree, security vulnerabilities. Another argument for this technique is that it can be integrated into the production pipeline, where code can be tested automatically into the build process. The experiment of this project shows that SonarQube could be a viable tool to be integrated into a production pipeline. Cross-checking code by running maven dependency with CVEs gives a portion of assurance that the committed code does not include classified vulnerabilities from the community. However, this technique has some limitations compared to the fuzzing technique CONFUZZION, which uses mutations. SonarQube will likely be harder to detect zero-day or unusual code fragments since it derives its intelligence from already known vulnerabilities using dependency-checking modules.

Fuzzers provide a more functional setting to facilitate access-control decisions in software. Testing program properties with varying mutations are essential to detect type-confusion vulnerabilities or unusual crafted code that targets specific functionality. CONFUZZION is a competitor to be heavily considered in the future that target object-oriented flaws and detects code derivations in the implementations. Research has shown promising results that it can detect the most critical type-confusion vulnerabilities with high probability.

Software-based reference monitors have shown to be a contender, especially with the ability to reduce the JVM footprint and detect violations of security policies. Security automation using finite-state automata policies keeps a state of transitions that ensures that the computing system does not perform consecutive functions that violate the system’s integrity. Injecting security checks in the application is an essential property that both provides insight and strengthens control in remote code executions. The reference mechanism is compared to manual code analysis, and fuzzers are a cheap option to be implemented that requires minimal overhead. When discussing this topic with developers, there seems to be a consensus on using software-based reference monitors for validating untrusted code. We believe that the reason is that the design requirements are relatively easier to implement with security automation and the ease of integration into existing code. Furthermore, if a security policy like the PSLangs is soundly enforced with correct properties, it is difficult for an attacker to circumvent the security setting and break out of the current transition.

Proof-carrying code is helpful if we assume that the code producer has created a robust theorem and the security policy is scalable to changes in the computing environment. Although it is worth noting that the technique assumes that we trust the code producer in the first place and that the code works as intended. Still, we suffer some issues with integrity. The problem occurs when a code producer gets hijacked, and a rogue actor modifies the original code by adding malicious invocation and reproducing the security policy. As the clients trust the code producer and confirm that the theorem is correct, it would allow clients to run malicious code. We see some difficulties in using this type of

technique in the production environment and partly is performance and security concerns. The method does not provide any integrity guarantees that the code performs its intended and correct function, even if it is from a legitimate code producer. We argue that the code still needs to be checked before execution, and therefore it loses its purpose.

Java provides some confidence in privacy by using private variables to hide the program's internal structure. However, it can be assumed to be less powerful because of its static property and vulnerability to human error [58]. Information-flow techniques that use labels to enforce code access security could be considered more powerful to be enforced. Jif is a powerful tool that extends the Java programming language by verifying programs using labels. The extension, together with the Java compiler, assures the variables are controlled adequately by principals. Information can be tracked statically at compile time, and tracking information flow provides security assurance.

Although every system needs security definitions and well-formed principles of authentication protocols to establish secure communication channels, as in the paper [59], we believe that all the mentioned techniques in this project could serve as good starting points to protect against vectors if used appropriately. However, another important consideration when designing and implementing protocols is the cost of the security mechanism. For example, what is the likelihood of a particular attack happening, and how much are we willing to invest in protecting the system against the vector? Besides, how will the implementation impact the users and the general performance of the computing environment? Unfortunately, these estimates are often overlooked when designing secure protocols, and the mechanisms often compromise the usability of the system rather than enhancing the security properties. These were interesting but hard questions that we discussed during the course, and there is no right answer but it all depends on the organisations time, money and usability.

However, language and run-time mitigation are unsatisfactory if users can infer with higher operating system-level computations. The military principle "need-to-know" derived from least privilege is useful when designing access-control policies for objects in a system. The Bell-LaPadula, Biba, and Clarke Wilson Security are well-known models for enforcing security at an operating system level. Although we create appropriate frameworks and design a robust system with sufficient countermeasures, failure is sometimes avoidable, and what we learn from this course is that we atleast should try to fail securely. The developers need to implement processes to deal with fall-back methods using internal recovery so that the attacker cannot derive any conclusion or take over the computations of a system.

8 Conclusion

This project investigates language-based security techniques and how they can be applied in the Java programming language. The underlying structure of the language enforces access control, permission models, and protection domains

which are well-needed to protect against attacks that attempt to break out from the sandbox environment. Although this type of security enforcement technique is used less in industry, it embodies the fundamental principles of how various stack inspection algorithms work. Based on the current threat landscape, the current architecture is not equipped enough to protect against the most crucial security threats. Thinking beyond the Java model is necessary to research techniques that satisfy code access security on a more fine-grained level. To accomplish this, we have covered the most common attacks towards the Java Language that we consider relevant to access control and presented language-based mitigation strategies by discussing defenses with their advantages and disadvantages in certain applications. We provided an experiment showing the possibility of detecting security vulnerabilities in Java code by cross-checking code in a vulnerable repository. The method had promising results in a production environment where potential vulnerabilities related to classes, methods, and variables were detected. We believe that this method could assist developers in discovering mistakes before the final commit to a branch.

The course TDA602 has been an eye-opener in many ways, and the ability to study language-based security provides a new opportunity to argue about security properties in systems in a more precise manner. Traditional security techniques like antivirus, firewalls, and intrusion detection systems are desirable tools to protect systems. Still, a remaining issue is that the techniques treat information like a black box and are vulnerable to zero days. Most companies, however, rely heavily on these techniques, and there is a probability that other attack windows will get overlooked. Language-based techniques are more sophisticated by looking into what is happening inside the box to detect malicious activity and could be more resource friendly to the system if compared to an IPS or IDS system. There is a clear need for such research and technology in academia and industry that can assist developers and security engineers in facilitating better decisions. Attackers are moving their artillery closer to the endpoints, and it is clear that we need to rethink our solutions on how to protect against malicious code.

Another technique that we believe has a promising future is Fuzzers and where we expect more research in the future. Automation testing techniques using varying mutations allow the ability to defend the client-side of the application and especially, find the most crucial threat that has plagued the Java language – the type-confusion vulnerability.

Appendix



Dependency-Check is an open source tool performing a best effort analysis of 3rd party dependencies; false positives and false negatives may exist in the analysis performed by the tool. Use of the tool and the reporting provided constitutes acceptance for use in an AS-IS condition, and there are NO warranties, implied or otherwise, with regard to the analysis or its use. Any use of the tool and the reporting provided is at the user's risk. In no event shall the copyright holder or OWASP be held liable for any damages whatsoever arising out of or in connection with the use of this tool, the analysis performed, or the resulting report.

[How to read the report](#) | [Suppressing false positives](#) | [Getting Help: github issues](#)

♥ [Sponsor](#)

Project: java-sec-code

sec:java-sec-code:1.0.0

Scan Information ([show all](#))

- dependency-check version: 7.0.4
- Report Generated On: Sun, 28 Aug 2022 14:55:10 +0200
- Dependencies Scanned: 194 (161 unique)
- Vulnerable Dependencies: 56
- Vulnerabilities Found: 342
- Vulnerabilities Suppressed: 0
- ...

Summary

Display: [Showing Vulnerable Dependencies \(click to show all\)](#)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
log4j-core-1.5.5.jar	cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle:1.55:*****	pkg:maven/org.bouncycastle:bcpkix-jdk15on@1.56	MEDIUM	1	Low	62
log4j-core-1.5.5.jar	cpe:2.3:a:bouncycastle:bouncy-castle-crypto-package:1.55:***** cpe:2.3:a:bouncycastle:bouncy-castle-crypto-package:1.55:***** cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle:1.55:***** cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle-java-cryptography-api:1.55:***** cpe:2.3:a:bouncycastle:the_bouncy_castle_crypto_package_for_java:1.55:*****	pkg:maven/org.bouncycastle:bcpkix-jdk15on@1.56	HIGH	15	Low	54
commons-beanutils-1.9.3.jar	cpe:2.3:a:apache:commons-beanutils:1.9.3:*****	pkg:maven/commons-beanutils/commons-beanutils@1.9.3	HIGH	2	Highest	164
commons-collections-3.1.jar	cpe:2.3:a:apache:commons-collections:3.1:*****	pkg:maven/commons-collections/commons-collections@3.1	HIGH	1	Highest	62
commons-httpclient-3.1.jar	cpe:2.3:a:apache:commons-httpclient:3.1:***** cpe:2.3:a:apache:httpclient:3.1:*****	pkg:maven/commons-httpclient/commons-httpclient@3.1	MEDIUM	2	Highest	91
commons-io-2.6.jar	cpe:2.3:a:apache:commons-io:2.6:*****	pkg:maven/commons-io/commons-io@2.6	MEDIUM	1	Highest	119
dom4j-1.6.1.jar	cpe:2.3:a:dom4j:project-dom4j:1.6.1:*****	pkg:maven/dom4j/dom4j@1.6.1	CRITICAL	2	Highest	120
dom4j-2.1.0.jar	cpe:2.3:a:dom4j:project-dom4j:2.1.0:*****	pkg:maven/org.dom4j/dom4j@2.1.0	CRITICAL	2	Highest	20
fastjson-1.2.24.jar	cpe:2.3:a:alibaba:fastjson:1.2.24:***** cpe:2.3:a:alibaba:fastjson:1.2.24:*****	pkg:maven/com.alibaba:fastjson@1.2.24	CRITICAL	2	Highest	38
fluent-hc-4.3.6.jar	cpe:2.3:a:apache:httpclient:4.3.6:*****	pkg:maven/org.apache.httpcomponents:fluent-hc@4.3.6	MEDIUM	1	Low	32
groovy-2.4.7.jar	cpe:2.3:a:apache:groovy:2.4.7:*****	pkg:maven/org.codehaus.groovy:groovy@2.4.7	CRITICAL	2	High	275
gson-2.8.0.jar	cpe:2.3:a:google:gson:2.8.0:*****	pkg:maven/com.google.code.gson:gson@2.8.0	HIGH	1	Highest	22
guava-23.0.jar	cpe:2.3:a:google:guava:23.0:*****	pkg:maven/com.google.guava:guava@23.0	MEDIUM	2	Highest	22
hibernate-validator-5.3.4.Final.jar	cpe:2.3:a:hibernate:hibernate-orm:5.3.4:***** cpe:2.3:a:hibernate:hibernate-validator:5.3.4:*****	pkg:maven/org.hibernate:hibernate-validator@5.3.4.Final	HIGH	5	Highest	33
httpclient-4.5.12.jar	cpe:2.3:a:apache:httpclient:4.5.12:*****	pkg:maven/org.apache.httpcomponents:httpclient@4.5.12	MEDIUM	1	Highest	32
icu4j-4.6.jar	cpe:2.3:a:apple:java:4.6:***** cpe:2.3:a:apple:java:1.54.6:*****	pkg:maven/com.ibm.icu:icu4j@4.6	MEDIUM	1	Low	65
jackson-annotations-2.8.0.jar	cpe:2.3:a:fastermj:jackson-modules-java8:2.8.0:*****	pkg:maven/com.fastermj:jackson-core:jackson-annotations@2.8.0	MEDIUM	1	Low	40
jackson-core-2.8.6.jar	cpe:2.3:a:fastermj:jackson-modules-java8:2.8.6:*****	pkg:maven/com.fastermj:jackson-core:jackson-core@2.8.6	MEDIUM	1	Low	46
jackson-databind-2.8.6.jar	cpe:2.3:a:fastermj:jackson-databind:2.8.6:***** cpe:2.3:a:fastermj:jackson-modules-java8:2.8.6:*****	pkg:maven/com.fastermj:jackson-core:jackson-databind@2.8.6	CRITICAL	39	Highest	42
jdom2-2.0.6.jar	cpe:2.3:a:jdom:jdom:2.0.6:*****	pkg:maven/org.jdom:jdom2@2.0.6	HIGH	1	Highest	65
jokix-core-1.6.0.jar	cpe:2.3:a:jokix:jokix:1.6.0:*****	pkg:maven/org.jokix:jokix-core@1.6.0	HIGH	1	Highest	18
jsoup-1.10.2.jar	cpe:2.3:a:jsoup:jsoup:1.10.2:*****	pkg:maven/org.jsoup:jsoup@1.10.2	HIGH	2	Highest	35
junit-4.12.jar	cpe:2.3:a:junit:junit4:4.12:*****	pkg:maven/junit:junit@4.12	MEDIUM	1	Low	49
log4j-api-2.9.1.jar	cpe:2.3:a:apache:log4j:2.9.1:*****	pkg:maven/org.apache.logging.log4j:log4j-api@2.9.1	LOW	1	Highest	44
log4j-core-2.9.1.jar	cpe:2.3:a:apache:log4j:2.9.1:*****	pkg:maven/org.apache.logging.log4j:log4j-core@2.9.1	CRITICAL	5	Highest	42
logback-core-1.1.5.jar	cpe:2.3:a:qos:logback:1.1.9:*****	pkg:maven/ch.qos.logback:logback-core@1.1.9	CRITICAL	2	Highest	33
mybatis-3.4.6.jar	cpe:2.3:a:mybatis:mybatis:3.4.6:*****	pkg:maven/org.mybatis:mybatis@3.4.6	HIGH	1	Highest	46
mysql-connector-java-8.0.12.jar	cpe:2.3:a:mysql:mysql:8.0.12:***** cpe:2.3:a:mysql:mysql-connector/j:8.0.12:*****	pkg:maven/mysql:mysql-connector-java@8.0.12	HIGH	6	Highest	44
netty-transport-4.0.27.Final.jar	cpe:2.3:a:netty:netty:4.0.27:*****	pkg:maven/io.netty:netty-transport@4.0.27.Final	CRITICAL	12	Highest	26
ogni-3.0.8.jar	cpe:2.3:a:ogni:project-ogni:3.0.8:*****	pkg:maven/ogni/ogni@3.0.8	MEDIUM	1	Highest	24
okhttp-2.5.0.jar	cpe:2.3:a:squareup:okhttp:2.5.0:*****	pkg:maven/com.squareup.okhttp:okhttp@2.5.0	HIGH	2	Highest	22
poi-3.10-FINAL.jar	cpe:2.3:a:apache:poi:3.10:*****	pkg:maven/org.apache.poi:poi@3.10-FINAL	HIGH	8	Highest	28
poi-ooxml-3.8.jar	cpe:2.3:a:apache:poi:3.8:*****	pkg:maven/org.apache.poi:poi-ooxml@3.8	HIGH	8	Highest	27
protobuf-java-2.6.0.jar	cpe:2.3:a:google:protobuf-java:2.6.0:*****	pkg:maven/com.google.protobuf:protobuf-java@2.6.0	MEDIUM	1	Highest	28
servo-core-0.10.1.jar	cpe:2.3:a:docker:docker:0.10:***** cpe:2.3:a:travis:ci-travis-ci:0.10:1:*****	pkg:maven/com.netflix.servo:servo-core@0.10.1	CRITICAL	25	Low	47
snakeyaml-1.21.jar	cpe:2.3:a:snakeyaml:project-snakeyaml:1.21:***** cpe:2.3:a:yaml:project-yaml:1.21:*****	pkg:maven/org.yaml:snakeyaml@1.21	HIGH	1	Highest	44

References

- [1] B. Venners, “The java virtual machine,” *Java and the Java virtual machine: definition, verification, validation*, 1998.
- [2] D. S. Wallach and E. W. Felten, “Understanding java stack inspection,” in *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No. 98CB36186)*, pp. 52–63, IEEE, 1998.
- [3] “Security developer’s guide.”
<https://docs.oracle.com/en/java/javase/17/security/java-security-overview1.html>. Accessed: 2022-06-24.
- [4] R. Toledo, A. Núñez, E. Tanter, and J. Noyé, “Aspectizing java access control,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 101–117, 2011.
- [5] U. Erlingsson and F. B. Schneider, “Irm enforcement of java stack inspection,” in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pp. 246–255, IEEE, 2000.
- [6] A. Bartel and J. Doe, “Twenty years of escaping the java sandbox,” *Phrack*, 2018.
- [7] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, “A calculus for access control in distributed systems,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 4, pp. 706–734, 1993.
- [8] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, “Authentication in distributed systems: Theory and practice,” *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 265–310, 1992.
- [9] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, “Secure program partitioning,” *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 3, pp. 283–328, 2002.
- [10] B. W. Lampson, “A note on the confinement problem,” *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973.
- [11] P. Bonatti and P. Samarati, “Regulating service access and information release on the web,” in *Proceedings of the 7th ACM conference on Computer and communications security*, pp. 134–143, 2000.
- [12] I. Bastys, M. Balliu, and A. Sabelfeld, “If this then what? controlling flows in iot apps,” in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 1102–1119, 2018.
- [13] A. Sabelfeld and D. Sands, “Dimensions and principles of declassification,” in *18th IEEE Computer Security Foundations Workshop (CSFW’05)*, pp. 255–269, IEEE, 2005.

- [14] K. J. Biba, “Integrity considerations for secure computer systems,” tech. rep., MITRE CORP BEDFORD MA, 1977.
- [15] G. McGraw and E. W. Felten, *Securing Java: getting down to business with mobile code*. John Wiley & Sons, Inc., 1999.
- [16] “MITRE cve-2014-0456.”
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0456>.
Accessed: 2022-06-24.
- [17] “MITRE cve-2015-4843.”
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4843>.
Accessed: 2022-07-21.
- [18] “MITRE cve-2016-3587.”
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3587>.
Accessed: 2022-07-21.
- [19] “MITRE cve-2017-3272.”
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-3272>.
Accessed: 2022-07-21.
- [20] “MITRE cve-2018-2826.”
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-2826>.
Accessed: 2022-07-21.
- [21] W. Bonnaventure, A. Khanfir, A. Bartel, M. Papadakis, and Y. Le Traon, “Confuzzion: A java virtual machine fuzzer for type confusion vulnerabilities,” in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pp. 586–597, IEEE, 2021.
- [22] “MITRE cve-2016-0264.”
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0264>.
Accessed: 2022-08-01.
- [23] “MITRE cve-2020-27221.”
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-27221>.
Accessed: 2022-08-01.
- [24] “insinuator sandbox escapes.”
<https://insinuator.net/2020/09/java-buffer-overflow-with-bytebuffer-cve-2020-2803-and-mutable-methodtype-cve-2020-2805-sandbox-escapes/>.
Accessed: 2022-06-24.
- [25] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.
- [26] “JEP411 deprecate the security manager for removal.”
<https://openjdk.org/jeps/411>. Accessed: 2022-10-07.

- [27] P. Louridas, “Static code analysis,” *Ieee Software*, vol. 23, no. 4, pp. 58–61, 2006.
- [28] C. Artho and A. Biere, “Combined static and dynamic analysis,” *Electronic Notes in Theoretical Computer Science*, vol. 131, pp. 3–14, 2005.
- [29] M. Ernst, C. Kaplan, and C. Chambers, “Predicate dispatching: A unified theory of dispatch,” in *European Conference on Object-Oriented Programming*, pp. 186–211, Springer, 1998.
- [30] M. D. Ernst, “Static and dynamic analysis: Synergy and duality,” 2003.
- [31] A. Aggarwal and P. Jalote, “Integrating static and dynamic analysis for detecting vulnerabilities,” in *30th Annual International Computer Software and Applications Conference (COMPSAC’06)*, vol. 1, pp. 343–350, IEEE, 2006.
- [32] P. Shijo and A. Salim, “Integrated static and dynamic analysis for malware detection,” *Procedia Computer Science*, vol. 46, pp. 804–811, 2015.
- [33] “github sonarqube.” <https://github.com/SonarSource/sonarqube>. Accessed: 2022-10-07.
- [34] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, “Finding software vulnerabilities by smart fuzzing,” in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 427–430, IEEE, 2011.
- [35] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.
- [36] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, “Coverage-directed differential testing of jvm implementations,” in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 85–99, 2016.
- [37] Y. Chen, T. Su, and Z. Su, “Deep differential testing of jvm implementations,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1257–1268, IEEE, 2019.
- [38] I. Aktug and K. Naliuka, “Conspec—a formal language for policy specification,” *Science of Computer Programming*, vol. 74, no. 1-2, pp. 2–12, 2008.
- [39] L. Bauer, J. Ligatti, and D. Walker, “Composing security policies with polymer,” in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 305–314, 2005.

- [40] F. Chen and G. Roşu, “Java-mop: A monitoring oriented programming environment for java,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 546–550, Springer, 2005.
- [41] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens, “Security monitor inlining for multithreaded java,” in *European Conference on Object-Oriented Programming*, pp. 546–569, Springer, 2009.
- [42] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, “Java-mac: A run-time assurance approach for java programs,” *Formal methods in system design*, vol. 24, no. 2, pp. 129–155, 2004.
- [43] J. Ligatti, L. Bauer, and D. Walker, “Enforcing non-safety security policies with program monitors,” in *European Symposium on Research in Computer Security*, pp. 355–373, Springer, 2005.
- [44] U. Erlingsson and F. B. Schneider, “Sasi enforcement of security policies: A retrospective,” in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX’00*, vol. 2, pp. 287–295, IEEE, 2000.
- [45] F. Yilmaz and M. Sridhar, “A survey of in-lined reference monitors: Policies, applications and challenges,” in *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, pp. 1–8, IEEE, 2019.
- [46] G. C. Necula and P. Lee, “Research on proof-carrying code for untrusted-code security,” in *IEEE symposium on security and privacy*, vol. 204, 1997.
- [47] C. Skalka, J. Ring, D. Darais, M. Kwon, S. Gupta, K. Diller, S. Smolka, and N. Foster, “Proof-carrying network code,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1115–1129, 2019.
- [48] Y. Lin, X. Cheng, and D. Gao, “Control-flow carrying code,” in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pp. 3–14, 2019.
- [49] H.-S. Kim and E. Lee, “Verifying code toward trustworthy software,” *Journal of Information Processing Systems*, vol. 14, no. 2, pp. 309–321, 2018.
- [50] G. C. Necula, “Proof-carrying code,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 106–119, 1997.
- [51] A. Vargun and D. R. Musser, “Code-carrying theory,” in *Proceedings of the 2008 ACM symposium on Applied computing*, pp. 376–383, 2008.

- [52] D. E. Denning and P. J. Denning, “Certification of programs for secure information flow,” *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.
- [53] S. Chong, A. C. Myers, K. Vikram, and L. Zheng, “Jif reference manual,” *June 2006. Available via*, 2009.
- [54] “MITRE cwe-397.” <https://cwe.mitre.org/data/definitions/397>. Accessed: 2022-06-24.
- [55] “github java-sec-code.” <https://github.com/JoyChou93/java-sec-code>. Accessed: 2022-08-21.
- [56] S. L. Nita and M. I. Mihailescu, “Jdk 17: New features,” in *Cryptography and Cryptanalysis in Java*, pp. 9–19, Springer, 2022.
- [57] A. Birrell, G. Nelson, S. Owicki, and E. Wobber, “Network objects,” *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 217–230, 1993.
- [58] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.
- [59] M. Abadi and R. Needham, “Prudent engineering practice for cryptographic protocols,” *IEEE transactions on Software Engineering*, vol. 22, no. 1, pp. 6–15, 1996.