Security Considerations for Java Graders

Sven Strickroth[1]

Abstract: Dynamic testing of student submitted solutions in evaluation systems requires the automatic compilation and execution of untrusted code. Since the code is usually written by beginners it can contain potentially harmful programming mistakes. However, the code can also be deliberately malicious in order to cheat or even cause damage to the grader. Therefore, it is necessary to run it in a secured environment. This article analyzes possible threats for graders which process Java code and points out Java specific aspects to consider when processing untrusted code.

Keywords: grader security, grader testing, security testing, automatic assessment, automatic grading

1 Introduction

Automatic graders are widely used in a variety of different systems and contexts such as assessment systems for programming courses, programming contest systems, and intelligent tutors [Ih10]. Robustness and trust are two central requirements on such systems (based on [Fo06]): Robustness means that a system does not crash randomly or when used in a way not intended by the developers. Trust includes two layers: (1) the system acts in a reliable way and cannot be cheated (the system behaves as expected at all times, it makes sure test results are gathered in a secure way and that the integrity of stored results is protected) and (2) sensitive data handed to the system remains private (e. g., by restricting access).

Nearly all of these systems require students to submit their code which is then automatically compiled and executed for grading or generating feedback on a server. In order to ensure trust and robustness considering security is important as untrusted code and data of students is processed. On the one hand the code is often from inexperienced programmers who might inadvertently code an endless loop or even harmful code. On the other hand more experienced students might try to contest a system to explore its boundaries (e. g. for cheating). Therefore, it is necessary to secure the environment on which the code is evaluated.

The main goal of this paper is to sensitize to security aspects as security is often not discussed in publications [Ih10] and to give an overview on possible threats. This should help developers of Java graders to test and optimize their systems. This paper focuses on Java as this is one of the most used language for programming support systems [St15].

In the following, the related research and the Java security architecture are presented. Then, key security aspects grouped into six sections are discussed. Finally, a conclusion is drawn.

[1] Universität Potsdam, Institut für Informatik & Computational Science, August-Bebel-Straße 89, 14482 Potsdam sven.strickroth@uni-potsdam.de

doi:10.18420/abp2019-5

2 Related research

Security aspects and advises for graders are discussed in several publications: In [Fo06] a classification of types of attack vectors for programming contest systems (denial of service, privilege escalation, destructive attack, and covert channel) which can occur at different times of the contest (compilation-time, execution-time, and contest-time) is provided. Considered attacks include forcing a high compilation time, consuming resources at compilation time, accessing restricted information, misusing the network, modifying or harming the testing environment, circumventing the time measurement, exploiting covert channels, misusing additional services, and exploiting bugs in the OS – as well as possible preventions. Another list of possible attacks is provided in [TB10] with a quite similar categorization: attacks during compilation (e. g., excessive submit size, referencing forbidden files, and denial of service), attacks during sandboxing (e. g., read/write files/directories, open sockets/access network, spawn multiple threads/processes, raising privileges resp. breaking the sandbox, vulnerabilities in the grader, and denial of service), and exploits during checking. Sims [Si12] categorizes attacks into denial of service (e.g., using a fork bomb, infinite loop, memory leak or excessive output), corruption of test harness, and leaking test data and discusses different security models (user-level restrictions, process-level restrictions, mandatory access control (e. g., SELinux), and virtual machines). Problematic system call groups (e. g., access to files, allocation of memory, creation of processes and threads, inter-process communication, executing other programs, and flushing buffers to disk) and an evaluation of the performance of different sandboxing approaches can be found in [MB12]. However, neither of the articles mentioned above is dedicated to Java and provides an overview of relevant security aspects.

3 The Java Virtual Machine and the Java Security Architecture

Java provides its own sandbox – the Java Virtual Machine (JVM) – in which the bytecode is executed. It is not a full virtual machine, but more like an additional layer of abstraction of the host operating system. Since version 1.2 of the JDK Java includes an enhanced security architecture which allows to restrict possible harmful actions programs can perform at run time. Java uses a stack-based permission approach.[2] When an action requires a special permission the `SecurityManager` (which delegates to `AccessController` since Java 1.3) is asked before executing the action which then checks whether all code blocks on the stack have the proper permissions. If and only if all code blocks on the call-stack have a specific permission, the permission is granted. Otherwise a runtime `java.security.AccessControlException` is thrown and the action is denied. There is, however, a special `PrivilegedAction` block which changes this logic in order to allow trusted code take responsibility for executing privileged actions on the behalf of otherwise untrusted code: If the call stack includes a `PrivilegedAction` block and the calling code has the proper permissions, then the action is permitted – the rest of the call stack is not checked.

[2] https://www.oracle.com/technetwork/java/seccodeguide-139067.html#9, https://docs.oracle.com/en/java/javase/12/security/java-se-platform-security-architecture.html

Therefore, it is important that `PrivilegedAction` blocks are as short as possible and used with care in order to not open additional attack vectors (also check loaded libraries).[3] Such an `PrivilegedAction` block (maybe in a trusted wrapper class) would be required for testing/mocking frameworks in order to generate and load test doubles on the fly.

The Java security manager can be enabled by passing `java -Djava.security.manager` to the JVM on the command line and a policy file can be specified using `-Djava.security.policy=FILENAME` (cf. List. 1).[4] An easy way to check whether a security manager is active is to check whether `System.getSecurityManager()` is `null` or not. Using `System.out.println(System.getSecurityManager().getClass().getName());` the active security manager can be detected (default is `java.lang.SecurityManager`).

```
grant codeBase "file:/FULLPATH/TESTCASE.jar" {
  permission java.lang.RuntimePermission "setIO";
  permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
};
grant {
  permission java.io.FilePermission "-", "read,␣write,␣delete";
  permission java.lang.RuntimePermission "accessDeclaredMembers";
};
```

List. 1: An example of a typical policy file which grants more permission to the TESTCASE than to the student code; e.g., `ReflectPermission ``accessDeclaredMembers''` is required for JUnit

Concluding, the Java sandbox together with the security manager allow to restrict access of untrusted student code to the operating system. Permissions need to be explicitly granted (with care, i. e. least privileges at the smallest scope possible). Not (fully) restricted by the `SecurityManager` are, however, the execution duration as well as memory and disk usage.

4 Memory Restrictions (Heap Size)

Allocating as much memory as possible can be used as a denial of service attack. It is possible to limit the memory usage by the host system (e. g., using POSIX `setrlimit`, cf. [Si12]), however, the Java Virtual Machine already automatically limits the maximum usable memory (usually to 25 % of available physical memory, depends on the host system architecture etc.; in order to see your Oracle/OpenJDK JVM details issue `java "-XX:+PrintFlagsFinal"` on the CLI and look for `MaxHeapSize`)[5] and throws an `OutOfMemoryError` exception in case an object runs out of memory and no more memory could be made available by the garbage

[3] see guidelines on `https://www.oracle.com/technetwork/java/seccodeguide-139067.html#9` also see `https://www.exploit-db.com/papers/45517`

[4] `https://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyFiles.html`, certain placeholders are possible; `https://docs.oracle.com/en/java/javase/12/security/permissions-jdk1.html`

[5] `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html#default_heap_size`

collector. If a JVM is exclusively used for evaluating a single student solution, the JVM dies cleanly with an exit code $! = 0$. However, if the student code is executed in a shared VM with other system code, this might be a more complex issue on how to react on the error (the Java documentation explicitly says "a reasonable application should not try to catch" an `Error` exception)[6]. A different maximum can be set by passing `-Xmx1G` to the JVM.[7] Generally, the maximum heap size should be chosen with care (how many processes might run in parallel, can the host go into an out of memory condition or start swapping). A lower limit could also be considered for the Java compiler (cf. end of Sect. 5).

5 Time Restrictions

Limiting the maximum time a student written program runs is very important so that lingering around processes can not occupy limited evaluation slots. An endless loop, a dead lock or infinite wait (e. g., attempts to acquire a mutex twice) can easily occur as a mistake. Generally, restricting time is considered more complex [Ma07]. Special attention must be paid, as a time restriction (e. g., based on POSIX `setrlimit`, cf. [Ma07]) does not limit the real execution time, but only the used CPU seconds. It is important that the real execution time (wall clock time) gets limited, as a waiting process might run for several hours (more than 3 hours on a test system) with a CPU time limitation of only 5 seconds (`ulimit -t 5`).

```
class A {{
  int a;
  try { a=0; } finally { try { a=0; } finally { try { a=0; } finally {
  try { a=0; } finally { try { a=0; } finally { try { a=0; } finally {
  try { a=0; } finally { try { a=0; } finally { try { a=0; } finally {
  try { a=0; } finally { try { a=0; } finally { try { a=0; } finally {
   a=0;
  }}}}}}}}}}}}
}}
```

List. 2: Java "Compiler Bomb" with 12 nested try-finally-blocks; `https://habr.com/en/post/245333/`

Time restrictions should not just be enforced for the run time of the student code, but also for the compilation of the code. Take the code listed in List. 2, it shows an example of a so-called compiler bomb (i. e. (relatively) small files which produce enormous output or consume a lot of resources), which is outputs an approx. 6 MiB class-file and takes about 3 seconds to compile (with OpenJDK 1.8.0_222). By just inserting 10 additional `try {a=0;} finally {` lines and corresponding closing braces, the compile time grows to about 5 minutes (but finally fails with `java.lang.OutOfMemoryError: Java heap space`). Additionally, there already were compiler bugs which caused infinite loops [Si12]. Such attacks can be used for a denial of service regarding compile time and potentially disk space (cf. Sect. 6).

[6] `https://docs.oracle.com/javase/9/docs/api/java/lang/Error.html`

[7] `https://docs.oracle.com/javase/9/tools/java.htm`

6 Filesystem access

File system access is an attack vector which is considered in all related research articles. It is important that student code can neither access restricted material (e. g., correct outputs for the test data, author's solution), modify or harm the testing environment (e. g., delete all files, replace evaluators, store state information between runs, overwrite test results), fill the hard disk (disk space and/or inodes; can also happen indirectly by outputting a huge amount of data to `stderr` or `stdout`), nor causing a high IO load by performing a lot of small writes.

Using `FilePermission` fine grained permissions can be granted, however, it is important to know that "code always automatically has permission to read files from its same (URL) location, and sub directories of that location; it does not need explicit permission to do so"[8] (as long as the user running Java has the appropriate rights; [Si12] provides a user/group model) and that code "can also obtain the pathname of the directory it is executed from, and this pathname may contain sensitive information"[8]. Indeed, in tests it was possible to access files and directories using a priori known names in and below the mentioned directory, however, it was not possible to list all files/directories using `File.listFiles()`. Yet, files a student should not access must be placed somewhere else as *security by obscurity* does not work (cf. [Fo06]). Access to other files requires explicit permissions in the policy file.

For some tasks it is required, however, that students can create and write to files. Particularly in these cases it is important that a clean(ed) environment is used for every evaluation of a student solution (called test "idempotency" resp. "hygiene" in [Si12]). In any case, it should be ensured that no arbitrary files can be placed into a directory which is on the class path. This is especially true for precompiled Java bytecode (`.class`) files. Loading these classes could be used to circumvent static analysis for detecting forbidden code/calls (ineffectivity is discussed in [Fo06]), package-private access to test code, or to "override" existing library classes (e. g., by creating a directory structure such as `org/junit` and dropping a prepared `.class` file there). This is not an issue for classes in a package whose name starts with `java.` as the default class loader refuses to load these from the user defined class path. Generally, the student code should be at the end of the class path after the test code for mitigation reasons. At best, use different class loader instances to separate classes.[9] There also exists a related problem: If custom security managers are used which rely solely on the absolute name of Java packages/classes for granting permissions (instead of the code location as the default security manager does) those can be tricked by using the approach described above.

A limitation of the explicit used disk space (and inodes) can be enforced e. g. by using quota features, by using distinct (logical) partitions or fixed-sized image files (those could be set up in advance and maybe pooled in order to not require root permissions). A limitation should also be enforced for the compiler (cf. Sect. 5). Generally, granted `FilePermissions` need to be tested carefully. There is no way in the standard JVM to limit a high IO and overwhelming output to `stderr` or `stdout` – here, other approaches such as Linux containers (cf. [MB12]) and trimming the data before storing in databases are necessary.

[8] `https://docs.oracle.com/en/java/javase/12/security/permissions-jdk1.html`, at least since Java 1.6

[9] `https://www.oracle.com/technetwork/java/seccodeguide-139067.html#4`; also prevents package-priv. access

7 Reflection

A quite special feature of Java is reflection, which has to be considered. Using reflection is always possible with Java regardless of any granted permissions. This way access to any classes on the class path is generally possible, e. g. directly calling the model solution or loading injected `.class` files. However, for listing as well as accessing protected/private methods/fields special permissions are required (`RuntimePermission "accessDeclaredMembers"` resp. `ReflectPermission "suppressAccessChecks"`; additional checks for packages of the JDK which are not considered public API such as `com.sun.proxy` exist and require further permissions (`accessClassInPackage.{package name}`). Whether or not reflection can be exploited needs to be carefully considered on a case-by-case basis (with the stack-based security model and class path in mind, cf. Sect. 9 for a case). There is no general solution.

8 Serialization and Deserialization

Deserialization of untrusted data can be used for denial of service (e. g., resource exhaustion, deserialization bombs) or also for (remote) code execution [Sv16]. This is especially an issue if serialized data is unserialized in code which has more privileges as this could be used for privilege escalation: That could be grader code which unserializes a student-created object or also any existing `PrivilegedAction` block within the JDK or any used library. Serialized data should be checked without deserializing it (e. g., compare the byte stream).

Serialization could also be used to access/modify otherwise protected/private data (using the serialized byte stream) or to instantiate classes with a private constructor. In principle, all classes on the class path that implement the `Serializable` interface are affected. Note, that no file system access is required for serialization (e. g., by using `ByteArrayOutputStream`). Therefore, it should be ensured that potential critical classes do not implement the `Serializable` interface.

9 Grader-Result Interface

How is the result of the student code resp. the grader passed back to a calling system? Possible ways include using `stdout`, a specific results file, calling an API of the surrounding system by the grader and/or checking the exit code of the JVM process. In all cases it is important to make sure that the untrusted student code cannot mimik (e. g., by printing a fake output similar to the one of the text based JUnit runner or calling the very same API) or overwrite the results (file) of the grader (e. g., by writing to and write protecting it or by using a JVM shutdown hook, the latter requires the `RuntimePermission "shutdownHooks"`). It might even be possible the fake the exit code and/or terminating the JVM by using `System.exit(0);` (after faking a positive result or just as a denial of service if the student code runs in a shared JVM), because the “"exitVM.*" permission is automatically granted to all code loaded from the application class path”[10] by default. In order to prevent this an extended security manager is required (cf. List. 3). No generic satisfying solution is known to the author.

```
public class NoExitSecurityManager extends SecurityManager {
  public void checkExit(int status) {
    super.checkPermission(new RuntimePermission("exitTheVM." + status));
  }
}
```

List. 3: An example for an extended `SecurityManager` which checks `System.exit(int)` and introduces a new permission `exitTheVM.*` which might need to be granted to the grader code

10 Discussion and Conclusions

Denial of service vulnerabilities may seem less critical, however, there are often peaks which often occur before deadlines in assessments [SG11]. A lot of issues can be mitigated by just safely and strictly enforcing a maximum run time, usable hard disk space as well as a proper queuing of evaluations of student solutions (especially important of the evaluation takes a long time).

Despite from errors or attacks in the student code, graders and testing code can also contain security vulnerabilities or programming errors (cf. [TB10]) and should, therefore, also run with least privileges possible (who writes the test code and how trustful is that person?). A general advise is to keep the grader and the policy file as simple as possible in order to allow easy review for security and correctness. Sample testing runs should be conducted in order to test whether the permissions are correctly applied and actually do work. Some Java source code which addresses/triggers the mentioned attacks for testing purposes can be found on `https://gitlab.com/javagradersec/examples`.

Not considered in this paper are exploiting bugs in the operating system, misusing additional services, spawning processes/threads, sending signals, network access, covert channels and the security of surrounding systems such as the web interface where the grader results are presented. The JVM comes with a sophisticated security architecture which requires permissions to be explicitly granted if the security manager is enabled (exceptions see Sect. 6 and Sect. 9 as well as default read permissions for some system properties). Spawning threads inside the JVM is always possible. Permissions should be granted with care (cf. Sect. 3) and possible new attack vectors need to be anticipated. Network access for example could be abused for starting a denial of service attack on third party systems or exfiltrate data. The Secure Coding Guidelines for Java SE state: “Utilizing lower level isolation mechanisms available from operating systems or containers is also recommended.”[11]

[10] `https://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html`, explicitly stated until Java 8

[11] `https://www.oracle.com/technetwork/java/seccodeguide-139067.html#9`

JVM and SecurityManager:
- ☐ Is the Java `SecurityManager` active?
- ☐ Are permissions granted and `PrivilegedAction` blocks used as restrictive as possible (how trustful are the tests and used/loaded libraries)?
- ☐ Is the student code tested in a separate JVM?

Memory:
- ☐ Is the maximum usable (heap) memory for compilation and execution reasonably restricted according to the host hardware (e. g., multiple tests running in parallel)?

Time:
- ☐ Are the program compilation and execution reasonably wall-clock time limited?

Reflection (Reflection is always possible):
- ☐ Are critical classes reachable inside the JVM (e. g., model solution code)?

Serialization:
- ☐ Untrusted objects must not be unserialized.

Filesystem:
- ☐ Is the test environment reset after each test run?
- ☐ Are model solutions or other critical files inaccessible from student code?
- ☐ Is the class path secured against injection of (.class) files (via upload or student code)?
- ☐ Are code permissions granted based on code path (instead of package name)?
- ☐ Are reasonable filesystem quotas enforced (for compilation and execution)?
- ☐ Is IO load limited?
- ☐ Is the output of the compiler/student program limited before saving (to a database)?

Grader-Result Interface:
- ☐ Is the interface safeguarded against skipping of tests or faking the test outputs?
- ☐ Is the JVM secured against prematurely exit?

Fig. 1: Java grader security checklist

Fig. 1 sums up all aspects of this paper. Some discussed aspects might seem to be academic, but still need to be considered. Of course this paper cannot be complete or guarantee perfect security. Hopefully, it sensitized readers to optimize their own systems for less known cases.

References

[Fo06] Forišek, M.: Security of programming contest systems. Information Technologies at School/, pp. 553–563, 2006.

[Ih10] Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O.: Review of recent systems for automatic assessment of programming assignments. In: Proc. Int. Conf. on Computing Education Research. ACM, New York, NY, USA, pp. 86–93, 2010.

[Ma07] Mareš, M.: Perspectives on grading systems. Olympiads in Informatics 1/, pp. 124–130, 2007.

[MB12] Mareš, M.; Blackham, B.: A New Contest Sandbox. Olympiads in Informatics 6/, pp. 100–109, 2012.

[SG11] Striewe, M.; Goedicke, M.: Studentische Interaktion mit automatischen Prüfungssystemen. In: Proc. DeLFI 2011. GI, pp. 209–220, 2011.

[Si12] Sims, R. W.: Secure Execution of Student Code, tech. rep., University of Maryland, Department of Computer Science, 2012.

[St15] Strickroth, S.; Striewe, M.; Müller, O.; Priss, U.; Becker, S.; Rod, O.; Garmann, R.; Bott, J. O.; Pinkwart, N.: ProFormA: An XML-based exchange format for programming tasks. eleed 11/1, 2015, ISSN: 1860-7470.

[Sv16] Svoboda, D.: Exploiting Java Deserialization for Fun and Profit, 2016.

[TB10] Tochev, T.; Bogdanov, T.: Validating the Security and Stability of the Grader for a Programming Contest System. Olympiads in Informatics 4/, pp. 113–119, 2010.