As Secure as Dangerous Can Be: Considerations for Secure Auto-Graders in the Context of MOOCs

Sebastian Serth
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
sebastian.serth@hpi.de

Daniel Köhler
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
daniel.koehler@hpi.de

Christoph Meinel
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
christoph.meinel@hpi.de

2024 IEEE Global Engineering Education Conference (EDUCON) | 979-8-3503-9402-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/EDUCON60312.2024.10578853

***Abstract*—In the context of programming education, so-called auto-graders allow learners to receive automated feedback on their submissions. Because assessing learners' code typically involves executing the learners' untrusted code, this commonly used mechanism poses a significant security risk for these systems. Since auto-graders are mostly employed in the context of large-scale learning environments, such as universities or Massive Open Online Courses (MOOCs), security considerations are especially important. In this paper, we first introduce our auto-grader CodeOcean, which is regularly used in MOOCs with thousands of active learners, and in university contexts. As the execution of untrusted code can entail severe security implications, ensuring that the application contains no security vulnerabilities is essential. Hence, we partnered with a security consultancy to assess our auto-grader system landscape through a professional penetration test. This work presents the findings and countermeasures resulting from the performed security analysis for CodeOcean. We contextualize overarching enhancements for three main categories of threat vectors to auto-grader systems. Implementing these in any auto-grader system can improve the security and prevent learners from manipulating the assessment of their code. We also discuss the potential consequences of hardening an auto-grader, such as a reduced system performance. Therewith, we provide valuable recommendations for educators, researchers, and system designers to improve the security of auto-graders in the future, supporting their usage in even larger settings or in the context of exams.**

***Index Terms*—Auto-Grader, Security, Penetration Test, Code Execution, Programming, MOOC**

I. Introduction

Programming education is often accompanied by practical coding exercises, allowing learners to apply the newly acquired knowledge. For this purpose, educators usually provide small to medium-sized tasks to work on. However, simply *providing* the exercises is not yet sufficient for the learning success: Learners also need *feedback* on their submissions, pointing out any potential mistakes. While an educator might be able to provide this feedback manually for a small group of learners, it quickly becomes too laborious for larger groups of learners at universities or within Massive Open Online Courses (MOOCs). Therefore, assessing learners' code submissions and either determining a (final) grade or providing detailed feedback is often automated in these settings with so-called auto-graders.

While providing feedback might be considered "enough" within university courses, learners in other contexts, such as MOOCs, potentially require additional technical support to get started with the required tools: Some auto-graders designed for novices also provide a web-based educational programming environment, allowing learners to start programming directly through their browsers. This approach relieves beginners from installing any programming tools on their devices and thereby removes a major hurdle in the beginning [1]. However, it also poses security challenges, when learners' untrusted code is executed by the auto-grader on an own server [2]. These security challenges induced by the execution of untrusted code need be properly mitigated by the auto-grader through various security best-practices. In the past, failing to do so has even led to severe security breaches across various products. Examples include the commercially-offered auto-grader *Gradescope* [3], self-developed auto-graders in the university context [4], [5] or possibilities for students to manipulate their scores [6]. Therefore, in the research paper at hand, we contribute security recommendations generalized from a professional security analysis of our auto-grader CodeOcean.

The remaining parts of the paper are structured as follows: First, we provide an overview of the possible architectures of auto-graders and the respective attack vectors before introducing our system CodeOcean in Section II. Following, we relate our work to previous research in Section III. Then, we cover general methodologies for security analysis and specific details about the penetration test conducted against our system in Section IV. In Section V, we highlight relevant results and sketch out the necessary steps taken to mitigate the issues. Section VI discusses the findings and mitigation strategies under consideration of performance implications and points out areas for future work. Finally, Section VII concludes our work.

II. Background and Status Quo

Executing arbitrary code is among the key features of auto-graders, and forms the base for assessing learners' submissions. While code written by instructors might be considered as trusted (depending on the security model), code written by learners should be considered untrusted [7]. Potentially, a learner could submit malicious code for execution, and

therewith attack the auto-grader, other users, or simply cheat. Therefore, providers of auto-graders generally choose one of two methods for code execution [7]:

Client-Sided Execution in which all code executions are local and restricted to a learner's browser.

Server-Sided Execution in which code is transferred to the server systems provided by the educational platform and usually executed in dedicated sandboxed environments.

While the former, client-sided approach does not require dedicated security measurements or further resources for code executions, support for other programming languages besides JavaScript might be limited [8]. Even though recent developments such as WebAssembly (also known as WASM) or JavaScript transpilers enable support for other languages or even virtual machines in the browser [9]–[12], some limitations still apply. For example, existing implementations do not support third-party libraries [13] or fundamental concepts such as threads or network access [10], [14]. Resource-constrained devices, such as older smartphone or tablets, might also yield an unsatisfying performance [7]. In addition, the client-sided approach requires that any test cases are also executed in the learners' browser, thus exposing this grading-related information, which in turn also eases cheating. Therefore, such a design is not suitable for a graded exams (in schools or universities), which usually requires a tamper-proof environment.

The second approach of server-side code executions does not come with these limitations, but drastically increases the amount of attack vectors as untrusted code is now executed on servers provided by the learning environment. Therefore, this approach requires additional efforts to protect the auto-grader from malicious actions (see Section III). Usually, the execution of untrusted code in an own system is considered a major security risk. In cybersecurity contexts, attacks able to execute code on a foreign server are considered one of the most serious vulnerabilities to date [15]. Yet, server-sided auto-graders offer that feature by design. Hence, an extensive security architecture to ensure isolation of the environment, preventing any executions beyond a pre-defined sandbox is crucial [5]. While some auto-graders in this category started with programming-specific isolation features (see Section III), many tools nowadays employ general purpose sandbox technologies. Those range from user-level programs to kernel features to dedicated virtual machines. As those approaches differ in their level of isolation and complexity, their performance and usage of system resources differs, too [2]. Regardless of the specific technology involved, all approaches aim to prevent an untrusted application to break out of pre-defined sandbox (sometimes referred to as a container escape), potentially harming the system.

A. Common Attack Vectors for Web Applications

Since many auto-graders are web-based [16], the general attack vectors for web applications apply to auto-graders, too. While a detailed analysis of such attack vectors is beyond the scope of this paper, those include insufficient access controls, injections, misconfigurations, or outdated software components. According to the OWASP Top 10, these lead the list of security issues in web applications [17]. Another common attack vector is *path traversal*, a vulnerability that could give an attacker access to an arbitrary file outside the common web server directory [18]. Such vulnerabilities might be exploited in both, the application server and in the auto-grader components in charge of executing learner code (i.e., inside the sandbox).

B. Specific Attack Vectors for Auto-graders

In addition to the aforementioned attack vectors also present for "regular" web applications, we categorize attack vectors for auto-graders as identified in previous literature [4]–[6] as follows:

Server-Sided Attackers could aim to break out of the pre-defined sandbox, either by actually executing arbitrary code (or programs) on the same machine or by using the auto-grader's network to reach other hosts. A container escape might compromise the host, e.g., by misusing the compute resources for crypto-currency mining or could otherwise impact the availability of the service for other users. Undesired network attacks can be further divided based on their target, such as the internet (e.g., to send spam mail) or the internal network accessible from the auto-grader (e.g., to access other internal resources, potentially causing data leaks).

Client-Sided Depending on the feature set of the auto-grader, attackers could also try to exploit client-side errors. For example, if learners can share arbitrary content with their peers that is not appropriately escaped, a traditional cross-site scripting (XSS) attack commonly found in web applications [19] could lead to an information leak or an account take-over. In the context of auto-graders, we consider this type of attack to be even more critical: Some tools allow learners to learn the basics of web programming, giving them control over the resources embedded in a website that could contain malicious components. Such an XSS attack can target supportive fellow learners or members of the teaching team through social engineering, based on the support or sharing features of the auto-grader used. It is important to be aware of this potential vulnerability and take necessary measures to prevent such attacks.

Grading Even without deep cybersecurity knowledge, learners could be motivated to cheat and thereby improve their scores. Instead of solving the given task on their own, they could employ different cheating mechanisms. For example, a learner could aim to access hidden test cases (or even a sample solution if provided), and thereby achieve a personal advantage. Those cheating attempts become an even larger issue for graded exams using auto-graders, e.g., in university contexts, where the learners' motivation to gain a better grade might be higher.

For us, investing in the security of auto-graders in all three categories is essential to ensure the integrity of the service for all users and to provide a reliable mechanism for grading

assignments. Otherwise, learners (or attackers) can abuse the system, as happened with university-developed systems [4]–[6] or the commercial auto-grader *Gradescope*. The latter system, which supports more than just programming assignments, is used by more than 3.2 million students around the world[1] and had several vulnerabilities: According to student reports, it has been possible to manipulate the *grading* process since 2016, either by revealing hidden test cases [3], [20], [21] or by directly writing the submitted score [3], [21], [22]). In addition, the system was already vulnerable to *client-side* threats, including XSS attacks [20], and suffered from a series of *server-side* vulnerabilities, ranging from unrestricted access to the underlying computing resources [20] to the establishment of a reverse shell [3], [21]. However, we want to emphasize that Gradescope is not the only auto-grader vulnerable to these kind of attacks, as an analysis of various educational auto-graders at universities shows [4]–[6].

C. Architecture Overview of our Auto-Grader CodeOcean

Tailored for beginners, we provide our own auto-grader called CodeOcean since 2014 [23], which we have been using regularly in MOOCs ever since. Its architecture is heavily influenced by security and scalability considerations, allowing it to be used in programming courses with more than 17,000 active learners so far. As shown in Figure 1, a micro-service architecture has emerged over time [2], mainly consisting of the CodeOcean web application, the Poseidon executor middleware, and a multi-node Nomad cluster. All code executions are run in Docker containers, utilized as a sandboxed environment and without any network connectivity, within the Nomad cluster. In this context, the Poseidon executor middleware is responsible for providing each learner with a new, previously unused container (and deleting those from inactive learners). The chosen components, namely the Nomad cluster and the Docker containers, were identified through a previous study comparing the performance, resource usage and isolation capabilities among various alternatives [2]: For our use case, the selected combination represented the most suitable combination of all requirements without the resulting architecture suffering from any known security vulnerabilities.

During the subsequent development, we primarily focused on mitigating server-side attack vectors and preventing client-side attacks by learners. The auto-grader was designed for novices voluntarily enrolling in a MOOC on their own and not a graded university context. Therefore, preventing technical cheating to improve one's grade was a secondary consideration. We further expected that those learners who were able to access the hidden test cases and leverage them to their advantages were already ahead of most of the learning goals we had defined for beginner courses anyway. Therefore, in the recent developments, we prioritized other security risks, such as privilege escalation, account take-overs, or information leaks. These technical measures were well considered and secured by appropriate (basic) safeguards.

[1]More information: https://gradescope.com

III. Related Work

For many years, *server-sided* security considerations of auto-graders have been subject to research. For example, programs executed by the Java Virtual Machine (JVM) can employ specific security mechanisms, i.e., the Java Security Manager, and thereby limit potential actions [24]. Regardless of the specific programming language used, Docker Containers can be used as a sandbox [25], which supposedly eliminates the need for isolation through virtual machines [26]. Docker containers use Linux kernel features to provide an isolation between processes and restrict file system usage, originally intended to ease portability by providing reproducible environments across different systems [27].

With regard to manipulating the automated *grading* of learner submissions, four different categories of threat vectors were identified in previous research [6]: Those attempts ranged from discovering auto-grading internals, such as inspecting

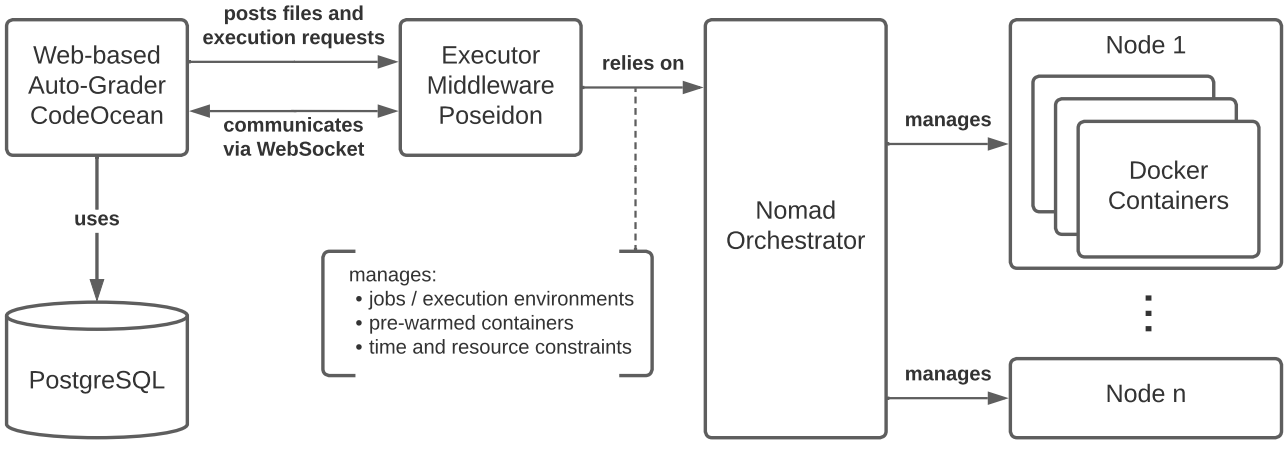

Fig. 1. System components of our auto-grader CodeOcean (see [2] for details). While the learner-facing web server manages all files for an actual execution (using a PostgreSQL database) and defines the command to execute, the executor middleware Poseidon is responsible for copying the desired files to an execution environment. All executions of learners' untrusted code are performed in Docker containers, which are managed by a distributed Nomad cluster.

test cases, over fetching a solution to overwriting the scoring functionality. Depending on the test setup, a cheating attempt might also fake some characteristics of a solution, e.g., by mocking a required recursion step [6]. This also becomes especially important in the context of (coding) competitions, where participants might be interested in overcoming the execution time limit for a given scenario [5].

Considering these specific attack vectors of auto-graders and the use cases (ranging from grades in universities to coding competitions), security researchers have also conducted structured analyses to strengthen auto-graders against those malicious actions. For example, Pudney et al. identified several vulnerabilities during a penetration test performed against auto-grading systems at their university [4]. They found several vulnerabilities in the sandboxed environments, which were partially caused by outdated software versions, but also by inadequate architectural decisions [4]. Further, the authors discovered multiple opportunities to cheat or impersonate as another user, and were able to prevent other users from submitting their solutions to the auto-grader [4]. Finally, the authors suggest extending the test scope to other auto-graders, which we do with the paper at hand.

In contrast to previous work by fellow researchers, our focus was on testing a large-scale auto-grader that has been constantly used and developed by more than 100,000 learners in different contexts since 2014. Originally designed as a research prototype, CodeOcean has recently seen adoption by other universities and in the context of enterprise MOOCs, highlighting the relevance of a security analysis. To our knowledge, no such in-depth analysis of an auto-grader, followed by a scientific evaluation of the underlying issues, has been conducted to date.

IV. Methodologies for Security Analysis

Proper security analysis is fundamental to engineering any software project and application. It describes applying different types and forms of (structured) analysis to existing or conceptual software or hardware to derive potential flaws in the security of the architecture or implementation [28]. Many current research approaches promise more accessible and extensive analysis of a software (and hardware) landscape, such as using Graph-based solutions [29], [30], or Machine learning-based methods [31], [32].

A. Types of Vulnerability Assessment

Generally, a vulnerability assessment contains *static* and *dynamic code analysis* as well as *manual testing*. During *static analysis*, automated tools scan software code for common or known vulnerabilities caused, e.g., by using unsafe operations of the respective programming language. During *dynamic analysis*, the software is often run in isolated environments in which its behavior is observed. At the same time, different inputs are entered to test the robustness of the software towards errors potentially caused by various (user) inputs. Once the software is deployed onto its server infrastructure, tools such as vulnerability scanners, which automatically assess, e.g., all open endpoints of the system, can be used. Such tools automatically enumerate known vulnerabilities or known concepts for vulnerabilities inside the system.

Vulnerability scanners provide a good cost-effective starting point to achieve an overview of a system's security. However, they are limited to enumerating known types of vulnerabilities. To identify in-depth vulnerabilities and issues within a system, *manual penetration tests* by cybersecurity experts can be employed [33], [34]. During a *penetration test*, the tester manually investigates software and systems in-depth, uncovering issues that automated testing usually does not discover.

B. Penetration Test against our System

In 2022, we contracted a professional company to perform a penetration test on our auto-grader. To ensure that no production systems, such as our online education platform, were impacted by the penetration test, we scheduled the testing period when no active programming courses required our auto-grader. The penetration test took three three weeks and was conducted by a dedicated team of three security specialists.

1) Assessment Scope: Through the penetration testing of our auto-grader CodeOcean, we wanted to uncover potential vulnerabilities in all aspects of the platform's lifecycle. This included the development process and Continuous Integration (CI) pipeline, the current version of our software, the integration with external platforms (such as Learning Management Systems), and our production deployment. Regarding our custom software, the test explicitly covered the learner-facing web server, the executor middleware Poseidon, and the Docker images used to run and grade learners' code. This also included the actual code executions performed through the Nomad orchestrator, considering potential interference with the regular operation of our service, or manipulating executions to a better grade through cheating. Furthermore, we were interested in hearing the experts' opinion about a potential integration of network access we previously prohibited in all exercises. Hence, we allowed the penetration testers to access the network. Finally, the integration of the Learning Tools Interoperability (LTI) standard [35] for the communication with external systems, the drafting of emails with user-generated content, and the admin login procedures were also under test, since those components play a crucial role in the authentication of any user. Older versions of our open-source software, potentially operated by partners, or any learning management system used to integrate with CodeOcean were explicitly not part of the testing. Similarly, the scope did not include any dedicated training of teaching team members (i.e., regarding their account security or social engineering attacks).

2) Testing Approach: Provided with a regular learner account to access different programming exercises, and a teacher account for our production environment, we started the penetration test with an introduction to the system and pointed the testers to the relevant source code. First, the team started by discovering the possibilities of our auto-grader through

implementing typical exercises, thereby getting an overview of the involved components and potential attack vectors. Then, throughout the test, we highlighted some atypical exercises drafted for single courses or research experiments, where we knew that more uncommon features were used. Later, the testers switched their focus to the functionality exclusively designed for teachers, also investigating whether a privilege escalation could be performed. Since the experts also had access to the code and a locally hosted version of the auto-grader, they also attempted to draft vulnerable requests, not possible through the regular user interface. Simultaneously, we observed the test with our regular monitoring tools. We considered this a practice run for spotting potential irregularities in monitoring that could be helpful during actual attacks. Thereby, we were able to identify and resolve the most severe issues exploited by the experts even before they were formally reported to us. The formal detailed report with all findings and reproduction steps for the attacks concluded the assessment period. Once we had implemented appropriate security measures for the outlined issues, the hardened application was retested to ensure that our mitigation was working as expected without introducing new vulnerabilities.

V. Generalized Findings of the Penetration Test

As part of the penetration test, we learned about a total of 23 findings affecting our auto-grader CodeOcean with all components used for regular operation. different types of vulnerabilities, were discovered. Those vulnerabilities included vulnerabilities common in web applications, such as XSS or programming errors, listed in the OWASP Top 10 [17], but also included dedicated attack vectors for auto-graders. Table I presents an overview of all findings identified for the respective categories of issues, ranging from manipulation of *Grading* results over *Server-Sided* issues such as weak authentication tokens to *Client-Sided* vulnerabilities such as Cross-Site-Scripting (XSS).

For example, one of the issues commonly affecting web applications was caused by insecure configurations of a third-party library that also affected other projects [36]. Another could be solved easily by applying programming best-practices for secure software, such as limiting the validity period of authentication tokens to a shorter period of time, rather then the previously chosen two-week period. While we consider a detailed list of all findings not to provide valuable insights beyond our own system, we will focus on a few surprising highlights in the remainder of this work. Therefore, we generalize the findings uncovered in our penetration test to support other maintainers of auto-grader solutions in assessing their environment. We present one generalized issue for each of the three categories, which we consider to be a representative example and relevant for other systems as well.

TABLE I
An overview of the 23 findings identified, divided into the respective categories of issues

Severity	Server-Sided	Client-Sided	Grading
4: Very High	2	0	0
3: High	2	1	0
2: Medium	2	6	0
1: Low	7	1	2

A. Server-Sided: Securing Execution with Sandbox Runtimes

As introduced in Section II-C and in line with similar auto-graders, we chose Docker containers to isolate code executions from each other and protect our host infrastructure. By default, Docker uses the execution engine `runc`, which in turn uses capabilities of the Linux kernel [26]. Therewith, the Docker ecosystem has a comparatively "large" attack vector, since vulnerabilities in the Linux kernel were previously exploited to escape from the container boundaries [37]. By employing a sandboxed runtime such as gVisor[2], the risk can be reduced further [26], [37]. gVisor integrates with Docker by providing a replacement for `runc` that filters system calls before passing them to the Linux kernel. Hence, gVisor provides a kernel isolation to the application executed in the sandbox [26], and therewith reduces the attack vector in comparison to the plain Linux kernel. While gVisor might still be vulnerable to some attacks (similar to any other software), it has been shown to successfully defeat a vulnerability otherwise leading to a container escape [37].

Generally, we found adding gVisor to our auto-grader as being rather simple. For us, installing and configuring gVisor was possible without major changes to our existing code base, allowing an seamless integration. However, after the integration, we also identified two non-security related bugs. The first was caused by a different reporting of out-of-memory errors by gVisor compared to Docker, which we used to display a corresponding alert to learners. The other one was actually a bug in gVisor, preventing JVM-based languages to receive any input from an interactive shell. This bug was fixed after we reported it upstream, allowing our auto-grader to run as expected on gVisor since then. As gVisor induces a new level of security, it unfortunately also induces performance penalty. Our monitoring showed that when gVisor was active, the performance of code executions dropped on average by around 20% (~650ms), with some outliers being about 30%-50% slower (up to 1.5 seconds). This performance impact of enabling gVisor might be mitigated by leveraging native hypervisor technologies using bare-metal machines[3], which, however, is not yet achievable in our private cloud environment.

B. Client-Sided: Protecting from Malicious Web Resources

For some exercises, having a program that is limited to text-based output is not enough. For example, in a course teaching the basics of modern web development with HTML, CSS, and JavaScript, server-sided executions of source code

[2]More information: https://gvisor.dev

[3]gVisor supports multiple so-called platforms to filter system calls, including Linux `KVM` to leverage the advantages of modern processors for the best possible performance on bare-metal machines.

is not enough. While the container can still be used for functional tests with a browser-based regression suite such as Selenium [38], having a native access to the website under construction is highly beneficial for learners and more closely resembles the actual development workflow in this area. Therefore, our auto-grader allows learners to render their website as HTML with the option to embed own images, style sheets or scripts. While we originally rendered the resulting page for the authoring user through our regular domain, the penetration test revealed a possibility for an attacker to misuse this functionality with social engineering skills. By leveraging a feature to ask peers for help with an exercise, a learner can make the raw, unrendered code available to fellow learners and ask ask them to manually copy as well as execute it, supposedly to reproduce a problem. When following this request, supportive learners would effectively run arbitrary code in the context of their cookie-based session on CodeOcean, opening up a whole range of XSS attacks.

We fixed this vulnerability by introducing a new domain for rendering user-generated content, the so-called *render host*, and by leveraging a content security policy[4] for our main site. A content security policy restricts the external resources allowed to be loaded during rendering of a webpage, thereby enhancing its robustness to XSS attacks. While the render host is technically operated by the same web server providing the regular CodeOcean service, the newly introduced domain separates the browsing context between the editor and render host. Therewith, an attacker's maliciously crafted page can no longer make authenticated requests to the CodeOcean domain, since that is now considered a third-party context. Access to individual pages on the render host is secured through a short-lived server-generated authentication token appended to the URL, minimizing the risk that someone else could access the learner's creation by accident.

C. Grading: Loose Access Control allows Cheating

Since the programming exercises we designed are primarily targeted at novices, preventing advanced cheating techniques was not our primary goal during development (cf. Section II-C). Nevertheless, we considered technical cheating to be a vulnerability during our penetration test to further reduce its chances of success. Even though we successfully prevented the exfiltration of the hidden teacher-defined test cases, learners were previously able to raise their assessment scores by manipulating the test cases within the execution environment. Since we previously copied all files (the learner's submission and the teacher-defined test cases) with the same file permission, the code invoked during dynamic unit testing could overwrite existing, teacher-defined test cases.

To counteract this issue, we first considered making the directory (including all files) containing the submission read-only using Linux file permissions before executing any code. However, we found no combination that would prevent an advanced learner from circumventing our protection while still allowing other files and subfolders to be created or modified as part of an exercise. Therefore, we introduced a new two-user approach for preparing and running code executions in our auto-grader: Since then, teacher-defined files are only writable by the superuser `root`, whereas all code executions are performed by a regular, non-privileged user. So far, this two-user approach covers all our requirements, and restricts learners from manipulating our scoring routine, which is now protected by a core concept of the Linux kernel.

D. Other Relevant Findings and Observations

Besides the findings across the three aforementioned attack vectors, we made some further observations during the penetration testing worth reporting.

1) Monitoring and Error Reporting: Long before considering a vulnerability assessment, we started integrating an extensive performance monitoring and error reporting tool chain into our application. While primarily designed to uncover performance issues or bugs, we valued the insights during the penetration test and when validating the impact of changes. Most importantly, the error reporting allowed us to understand and fix the most severe issues discovered by the testers even before receiving their corresponding report.

2) Network Access: While being aware of the security implications of granting network access, the penetration team supported us in securing our internal network resources from being attacked through code executions. Using the container networking plugins (CNI)[5], we created a custom network bridge, effectively filtering traffic to non-public destinations. Applying this practice, other internal resources are now well protected from (malicious) learners with access to networking functionalities from within the auto-grader. Still, previously mentioned security risks covering, e.g., learners using our platform as proxy to perform harmful actions on the internet, need to be considered. Deriving an appropriate concept for securing internet access is subject to future research before rolling out the network support widely in our courses.

3) Development Process: The analysis of our Continuous Integration (CI) pipeline and otherwise publicly available development information did not reveal any findings. The only remark concerned one old Docker image that downloaded a third-party library through an unencrypted connection at build time. We mitigated the issue by switching to a secure connection.

VI. Discussion and Future Work

Even while the penetration test was still being carried out, we started already fixing the issues discovered. As a side effect, we also reflected on certain implementation details and subsequently changed these. For example, the solution chosen to prevent a manipulation of the automated *grading* of learner submissions had several implications: On the one hand, it made copying files into the Docker container more complex and also requires a user switch for code executions.

[4]More information: https://content-security-policy.com

[5]More information: https://github.com/containernetworking/plugins

On the other hand, we considered the implementation effort as a chance to improve the internal architecture of the Poseidon executor middleware in this respect. Simultaneously, the additional security measurements negatively affected the performance of our auto-grader, even though, for example, the user switching only has a minor impact of about 200ms (~7%) on average with gVisor being enabled. Nevertheless, we consider security of the infrastructure and users to be a top priority that should not be undermined by (slightly) degraded performance. Therefore, we also accept the noticeable performance impact caused by enabling gVisor, which we might be able to further reduce in the future by using a non-virtualized server for the Nomad agents.

Besides further securing our infrastructure by restricting functionality, the penetration test also allowed us to work on new features. For example, we previously did not equip the execution environments with any network access (cf. Section II-C). The solution developed with the external penetration testers successfully prevents attack vectors against our internal resources. As a result of this development, we are now able to optionally offer internet access to the Docker containers in selected cases. However, this success represents only a first partial step in offering public internet access while preventing the misuse of our systems for cybercrime. Therefore, further efforts are needed to enable network-based code executions.

A critical comparison of the security analysis and subsequent hardening of our auto-grader CodeOcean with the available details of other auto-graders generally reveals similar attack vectors, but partially different mitigation strategies. For example, manipulating the *grading* is effectively prevented in our auto-grader CodeOcean, whereas others have had open vulnerabilities for years (e.g., Gradescope [3], or various other university-offered solutions [5], [6], [16]). While fellow researchers have further compared the performance impact of security measurements extensively (e.g., for the auto-grader Submitty[6]) [39] or identified potential breakouts of the sandbox employed in custom university-designed auto-graders [4], we only provide a rough baseline in this regard throughout the paper at hand. Other *server-side* attack vectors, especially in network-enabled scenarios, remain a challenge for auto-graders, with the safest option being to disable public Internet access wherever possible. Regarding *client-side* XSS attacks, we consider our solution developed for rendering learner-defined web pages to be secure and reliable, and we are currently not aware of any other auto-grader that uses a similar architecture.

Overall, our work has identified several additional considerations that should be taken into account when implementing or considering security mechanisms for auto-graders. Similarly, fellow researchers have previously highlighted their experiences in their contexts [4]–[6]. Consequently, attempting to provide an auto-grader service today would require aggregating best practices from different studies and scenarios. Hence, future work should target contextualising the various experiences and deriving an overview for auto-graders, similar to the OWASP list of common vulnerabilities.

[6]More information: https://submitty.org

VII. Conclusion

Automatically assessing learners' code with so-called auto-graders has become common practice in modern programming education, and is regularly carried out by universities and in Massive Open Online Courses. However, the server-side code execution of learners' untrusted code, as performed by our auto-grader CodeOcean, is subject to numerous security considerations. Based on an extensive security analysis of CodeOcean by security experts, we present generalizable insights for maintainers of other auto-graders in this paper. These can be categorized into three main threat vectors: (1) *Server-sided* attacks, which directly threat the provider's infrastructure and could potentially affect any user accessing the system, might be mitigated through a hardened sandbox environment. (2) *Client-sided* attacks, which pose a security risk for the data integrity of learners and instructors working with the tool, can be circumvented by a dedicated domain for user-generated content. (3) *Grading* related attacks, which aim to manipulate the scoring routine and thus could grant attackers an unfair advantage towards certification, are prevented by stricter access control. For each of the three categories, we present a typical example uncovered during the analysis of our auto-grader, reveal our mitigation strategy, and discuss its impact on system performance. Thereby, we contribute to a more profound overview of attack vectors and security considerations, making auto-graders more secure given their growing popularity.

Acknowledgment

We would like to thank Fabian Bräunlein, Lukas Euler, and Maximilian Kirchmeier from Positive Security[7] for their extensive penetration test and their valuable advice on hardening our auto-grader CodeOcean.

Funding

This work was funded by the Federal Ministry of Education and Research (BMBF) within the framework of the "Initiative National Education Platform".
BMBF Grant Number: CoHaP2 (16INB2009)

Funded by the European Union – NextGenerationEU. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or European Commission. Neither the European Union nor the European Commission can be held responsible for them.

References

[1] H. T. Tran, H. H. Dang, K. N. Do, T. D. Tran, and Vu Nguyen, "An interactive Web-based IDE towards teaching and learning in programming courses," in *Proceedings of 2013 IEEE TALE Conference)*. Bali, Indonesia: IEEE, Aug. 2013, pp. 439–444. [Online]. Available: http://ieeexplore.ieee.org/document/6654478/

[7]Homepage and contact details: https://positive.security

[2] S. Serth, D. Köhler, L. Marschke, F. Auringer, K. Hanff, J.-E. Hellenberg, T. Kantusch, M. Paß, and C. Meinel, "Improving the Scalability and Security of Execution Environments for Auto-Graders in the Context of MOOCs," in *Proceedings of the Fifth Workshop "Automatische Bewertung von Programmieraufgaben" (2021), Virtual Event, Germany, October 28-29, 2019*. Gesellschaft für Informatik e.V. (GI), Oct. 2021, pp. 3–10. [Online]. Available: https://dl.gi.de/handle/20.500.12116/37539

[3] A. Saligrama, "A student's dream: hacking (then fixing) Gradescope's autograder," Feb. 2023. [Online]. Available: https://saligrama.io/blog/post/gradescope-autograder-security/

[4] I. Pudney, R. Wawrzaszek, and A. Yarger, "Revenge for Late Nights: Penetration Testing of University Autograders," University of Michigan, Ann Arbor, May 2016. [Online]. Available: http://rgdoi.net/10.13140/RG.2.1.5126.7448

[5] M. Mareš, "Security of Grading Systems," *Olympiads in Informatics*, pp. 37–52, 2021. [Online]. Available: https://ioinformatics.org/journal/v15_2021_37_52.pdf

[6] N. Kratzke, "Smart Like a Fox: How Clever Students Trick Dumb Automated Programming Assignment Assessment Systems," in *Proceedings of the 11th ITiCSE Conference*. Heraklion, Crete, Greece: SciTePress, 2019, pp. 15–26. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0007424800150026

[7] S. Serth, M. Paß, and C. Meinel, "On the Feasibility of Serverless Functions in the Context of Auto-Graders," in *2023 IEEE German Education Conference (GeCon)*, Berlin, Germany, Aug. 2023, p. 6.

[8] V. Karavirta and P. Ihantola, "Serverless automatic assessment of Javascript exercises," in *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. Bilkent Ankara Turkey: ACM, Jun. 2010, pp. 303–303. [Online]. Available: https://dl.acm.org/doi/10.1145/1822090.1822179

[9] B. Powers, J. Vilk, and E. D. Berger, "Browsix: Bridging the Gap Between Unix and the Browser," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 253–266, May 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3093336.3037727

[10] R. Sharrock, L. Angrave, and E. Hamonic, "WebLinux: a scalable in-browser and client-side Linux and IDE," in *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. London United Kingdom: ACM, Jun. 2018, pp. 1–2. [Online]. Available: https://dl.acm.org/doi/10.1145/3231644.3231703

[11] S. Graham and contributors to the Skulpt Project, "skulpt - Python. Client Side." May 2021. [Online]. Available: https://skulpt.org/

[12] M. Pabst, "[Online-IDE] LearnJ - ein interaktiver Java-Kurs für Schüler/innen," Nov. 2022. [Online]. Available: https://www.learnj.de/doku.php?id=start#die_programmiersprache_auf_dieser_website_-_ist_das_wirklich_java

[13] ——, "[Online-IDE] LernJ vs. Java: Unterschiede," Mar. 2023. [Online]. Available: https://www.learnj.de/doku.php?id=unterschiede_zu_java:start

[14] Y. Saito and open-source contributors, "ruby.wasm - Notable Limitations," Dec. 2023. [Online]. Available: https://github.com/ruby/ruby.wasm#notable-limitations

[15] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 652–661.

[16] J. C. Paiva, J. P. Leal, and Á. Figueira, "Automated Assessment in Computer Science Education: A State-of-the-Art Review," *ACM Transactions on Computing Education*, vol. 22, no. 3, pp. 1–40, Sep. 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3513140

[17] OWASP Foundation, Inc., "OWASP Top 10 - 2021," Sep. 2021. [Online]. Available: https://owasp.org/Top10/

[18] ——, "Path Traversal - OWASP," Dec. 2020. [Online]. Available: https://owasp.org/www-community/attacks/Path_Traversal

[19] OWASP Foundation, Inc. and KirstenS, "Cross Site Scripting (XSS) - OWASP," Nov. 2022. [Online]. Available: https://owasp.org/www-community/attacks/xss/

[20] S. Hao, E. Park, D. Wong, and D. Zheng, "Security Analysis of Gradescope," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep., May 2016. [Online]. Available: https://courses.csail.mit.edu/6.857/2016/files/20.pdf

[21] A. Lyu, "How a frustrating computer science assignment gave me the idea to take over the server that graded it (GradeScope)," Mar. 2020. [Online]. Available: https://medium.com/@andylyu/how-a-frustrating-computer-science-assignment-lead-to-me-gaining-access-to-the-server-that-graded-502310cf03ae

[22] H. Wang, "Hacking Gradescope Autograder," Dec. 2019. [Online]. Available: https://hanbang.wang/blog/hack-gs/

[23] S. Serth, T. Staubitz, R. Teusner, and C. Meinel, "CodeOcean and CodeHarbor: Auto-Grader and Code Repository," in *SPLICE 2021 workshop CS Education Infrastructure for All III: From Ideas to Practice*, Virtual Event, Mar. 2021, p. 5. [Online]. Available: https://csplice.github.io/SIGCSE21/proc/SPLICE2021_SIGCSE_paper_13.pdf

[24] S. Strickroth, "Security Considerations for Java Graders," in *Proceedings of the Fourth Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2019), Essen, Germany, October 8-9, 2019*, S. Strickroth, M. Striewe, and O. Rod, Eds. Essen, Germany: Gesellschaft für Informatik e.V. (GI), Oct. 2019, pp. 35–43. [Online]. Available: http://dl.gi.de/handle/20.500.12116/27944

[25] J. Breitner, M. Hecker, and G. Snelting, "Der Grader Praktomat," in *Automatisierte Bewertung in der Programmierausbildung*. Münster, Germany: Waxmann Verlag, Jul. 2016, vol. Digitale Medien in der Hochschullehre, pp. 159–172. [Online]. Available: https://elan-ev.de/dateien/band6-openaccess.pdf

[26] O. Flauzac, F. Mauhourat, and F. Nolot, "A review of native container security for running applications," *Procedia computer science*, vol. 175, pp. 157–164, Aug. 2020. [Online]. Available: https://doi.org/10.1016/j.procs.2020.07.025

[27] C. Anderson, "Docker [Software engineering]," *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7093032/

[28] A. T. Murray, T. C. Matisziw, and T. H. Grubesic, "A Methodological Overview of Network Vulnerability Analysis," *Growth and Change*, vol. 39, no. 4, pp. 573–592, 2008. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1468-2257.2008.00447.x

[29] S. M. Ghaffarian and H. R. Shahriari, "Neural software vulnerability analysis using rich intermediate graph representations of programs," *Information Sciences*, vol. 553, pp. 189–207, Apr. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025520311579

[30] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of the 9th ACM conference on Computer and communications security*, ser. CCS '02. New York, NY, USA: ACM, 2002, pp. 217–224. [Online]. Available: https://dl.acm.org/doi/10.1145/586110.586140

[31] A. O. A. Semasaba, W. Zheng, X. Wu, and S. A. Agyemang, "Literature survey of deep learning-based vulnerability analysis on source code," *IET Software*, vol. 14, no. 6, pp. 654–664, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1049/iet-sen.2020.0084

[32] S. M. Ghaffarian and H. R. Shahriari, "Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey," *ACM Computing Surveys*, vol. 50, no. 4, pp. 56:1–56:36, Aug. 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3092566

[33] H. H. Thompson, "Application penetration testing," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 66–69, Jan. 2005.

[34] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 84–87, Jan. 2005.

[35] 1EdTech Consortium, Inc., "Learning Tools Interoperability (LTI)®," Jul. 2019. [Online]. Available: https://www.imsglobal.org/spec/lti/security-update/v1p0

[36] L. Euler, "Ransacking your password reset tokens," Jan. 2023. [Online]. Available: https://positive.security/blog/ransack-data-exfiltration

[37] F. Voznika, "Containing a Real Vulnerability," Sep. 2020. [Online]. Available: https://gvisor.dev/blog/2020/09/18/containing-a-real-vulnerability/

[38] A. Holmes and M. Kellogg, "Automating Functional Tests Using Selenium," in *AGILE 2006 (AGILE'06)*. Minneapolis, MN, USA: IEEE, 2006, pp. 270–275. [Online]. Available: http://ieeexplore.ieee.org/document/1667589/

[39] M. Peveler, E. Maicus, and B. Cutler, "Comparing Jailed Sandboxes vs Containers Within an Autograding System," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. Minneapolis MN USA: ACM, Feb. 2019, pp. 139–145. [Online]. Available: https://dl.acm.org/doi/10.1145/3287324.3287507