Real-World Study of the Security of Educational Test Systems

Stefan Gast, Sebastian Daniel Felix, Alexander Steimaurer, Jonas Juffinger, and Daniel Gruss

Graz University of Technology, Austria

Abstract. Computer science education has a unique setting where students write code commonly automatically tested in so-called "test systems". While best practices for sandboxing are known in the academic community, the security of real-world test systems is unclear. In this work, we evaluate the security of 11 real-world test systems from computer science university classes, including computer security classes. We provide a systematic overview of these systems and group them into 3 categories. We identify 13 types of security issues, the most widely spread ones affecting 5-6 of the test systems in our analysis. We practically show that all test systems in our analysis can be compromised and develop new techniques to exfiltrate secret and privileged information, including the use of side channels. We present 3 cases studies where we show the specific bypasses possible in these real-world systems. Finally, in a user study, we assess the impact a potential breach together with the educators using these test systems. Our work shows that educational test systems are particularly critical, as a compromise can lead to the exposure of highly sensitive student and research data, and even embargoed vulnerabilities. Our results highlight that the real-world challenges to run and maintain secure test systems are not solved in practice. While we discuss best practices, our study reveals the need for new systematic security approaches to secure this very common type of software system.

Keywords: Real-World Study, Untrusted Code, Test Systems

1 Introduction

A crucial element of university computer science education is that students also practically write code. Educators review this code and often also used to grade the students [18]. Practical programming courses range from introductory courses to programming languages to advanced courses requiring direct hardware access or courses for practicing the development of secure code and exploitation of insecure code. Especially for larger institutions with thousands of students, reviewing student code can accumulate to a significant workload [34].

A solution to this resource problem is the partial or full automation of the code reviews [34, 26]. For this purpose, an automated "test system" [35] runs student code with test inputs and analyzes the behavior and output. Especially for

classes with thousands of users, this is a unique setting where large amounts of untrusted code are run on university computer systems. There are different approaches on how to run the untrusted student code, ranging from simple scripts that execute one program after another to the use of containers and virtual machines. Wilcox [35] already noted that security is a crucial aspect of these systems and advised the use of chroot or virtual machines. Similarly, Paiva et al. [26] argue that virtual machines and containers are among the most comprehensive solutions to run student code securely. As the students are also often provided with Git repositories via university GitLab instances, one can consider GitLab Runners (the continuous-integration system integrated in GitLab) [11] for testing student code. GitLab Runners can be configured to use a container or a virtual machine. While under normal use, the output would be provided to the developer, educators can restrict the information they hand to students, e.g., via customized interfaces that communicate with GitLab through an API. However, CI systems are intended for trusted environments, whereas in malware analysis, where the fundamental assumption is that the code under analysis is malicious, other tools, such as virtual machines [38], are used to fully isolate the malicious code from the rest of the system. While best practices for sandboxing and isolation are well-known [30, 20, 7], and some even come with provable security guarantees [3], the situation for is less clear for real-world test systems, as these are often not publicly available and not developed by a professional development team. Importantly, prior work used literature surveys to assess the security of automated test systems but not a practical study of real-world systems [26].

In this work, we evaluate the security of 11 real-world test systems used by 3821 students in computer science university classes during the time frame from October 2023 to February 2024, including computer security classes. We studied these systems and provide a systematic overview of the typical approaches these systems follow. We identify 3 categories of systems: GitLab Runners with a Docker registry, GitLab Runners with custom pipelines, and entirely custom test systems that do not rely on a CI system as a basis. We identify 13 types of security issues, the three most widely spread ones affecting 5-6 of the test systems in our analysis.

We practically show that all test systems in our analysis can be compromised and develop new techniques to exfiltrate secret and privileged information, including the usage of side channels. We present 3 cases studies where we show the specific bypasses possible in these real-world systems. In a user study, we assess the impact of a potential breach together with the educators using these test systems. We show that test systems in university contexts are particularly critical, as a compromise can lead to the exposure of hundreds of highly sensitive student records, confidential research data, and in some cases even embargoed vulnerabilities. Our results highlight that the real-world challenges to run and maintain secure test systems are not solved in practice. While we discuss best practices that educators should follow, our study emphasizes the need for new systematic security approaches to secure this very common type of software system.

Contributions. We summarize our contributions as follows.

- 1. We analyze the security of 11 real-world test systems in computer science education practically used by 3821 students.¹
- 2. We identify 3 categories of systems and 13 types of security issues. We showcase in 3 case studies the specific security bypasses possible.
- 3. We assess the impact of potential breaches in a user study, highlighting that a compromise can even lead to the breach of vulnerability embargoes.
- 4. We highlight open challenges for real-world educational test systems, showing that novel security approaches are necessary in practice.²

Outline. Section 2 provides background. Section 3 presents our systematic analysis, Section 4 the security issues identified, Section 5 three case studies, and Section 6 our user study. Section 7 contextualizes our work. Section 8 concludes.

2 Background

In this section, we overview computer science education, focusing on higher education, educational aspects of test systems in coding classes, principles of related continuous integration systems, and misconfiguration of those.

Programming Education at Universities. In computer science-related study programs, programming is a core concept in the curricula. Courses that involve programming often comprise a heterogenous groups of students, continuous assessment, and large performance differences between students [4]. In this context, scalability is one of the major challenges for educators [19], leading to lack of feedback and communication.

The Need for Automated Test Systems. Automated testing systems play a crucial role in accommodating this complexity by providing personalized feedback, assessment, and support to students at different skill levels in many computer science courses. Educators carefully design test suites and evaluation criteria to assess students' mastery of key concepts, problem-solving skills, and software development practices, ensuring that assessments are meaningful and relevant to the course objectives. Wrenn et al. [37] show that this widely used approach may negatively impact the quality of the assessment of students due to cases not fully considered in the test suites. However, Mitra [21] showed that it still supports students to develop independent testing skills and positively impacts their work, especially for underrepresented student groups. Test systems exist for different programming languages, environments, and testing needs.

Continuous Integration Systems. Continuous integration (CI) refers to the widespread [15, 31] practice of frequently integrating changes into a shared mainline repository. Instead of postponing the integration to a later development

¹ Students counted per course, *i.e.*, number of individuals will be lower. Still, students per course is the better metric for how much these systems were used.

² We emphasize that, although we investigate educational test systems, the insights of our work are not in the domain of educational science, *i.e.*, what educators can do to improve education, but in the domain of security research, *i.e.*, what we need from security research to improve the security of these real-world systems.

phase, it is performed regularly, at least once per day. By this, CI aims to reduce and simplify integration conflicts, as changes do not accumulate over an extended period, making conflicts more manageable and easier to resolve.

In addition to a shared and frequently updated mainline code base, CI includes automated builds and tests triggered by every mainline code update. This ensures quick feedback about programming and integration errors, minimizing the effort to narrow them down to specific changes.

CI systems, e.g., Jenkins [17], Travis CI [32], or GitLab Runner [10], enable development teams to set up automated build and test pipelines. When triggered by a code update, these systems fetch the current source code from the repository, run the project-defined build and test pipeline, and report back the result via email or a web interface. Pipelines are usually run on specialized executors, either hosted by the vendor of the CI system or on a project-managed instance.

The actions to be executed during a pipeline run are defined by scripts. These can be stored in the code base, together with the code of the project itself, or in a separate code repository. Executors run these scripts and, during testing, also the code of the project. Effectively, anyone with write access to the code base can run arbitrary code on the executor. However, in usual development settings, only development team members have write access. Consequently, executors do not run untrusted code in these settings.

Furthermore, as best practices recommend [14, 12], pipelines typically run ephemeral containers or virtual machines. Thus, pipelines always run the same environment, without any traces of prior runs influencing the results. In addition, when configured correctly [12], this also enhances security, as potentially dangerous code is restricted to interacting with its ephemeral sandbox.

Security Misconfigurations. To support a wide variety of programming languages, project management styles, application domains, and test cases, CI systems have to provide a large degree of flexibility. In addition to the ability to execute arbitrary code within the pipeline, these systems also have a rich set of configuration options. In GitLab, for example, the overall execution of the pipeline is controlled by a configuration file within the code base [11], whereas the web interface is used to assign executors (*i.e.*, runners in GitLab terms) to specific projects, pass environment variables to the executor and set up access tokens. Especially in the context of isolating workloads, e.g., with Docker, or container orchestration tools like Kubernetes, security misconfigurations can compromise the entire system setup [36, 28]. However, misconfigurations can occur on any layer of the software stack, especially for use cases with a wide range of configuration options [39], including the application level [8]. Based on their analysis, Dietrich et al. [5] conclude that there are countless undiscovered security issues in systems connected to the Internet. In educational test systems, misconfiguration issues can occur on all layers, e.g., within the container, the runner, and the virtual machine hosting the runner. Finally, misconfiguration can also be a reason for deferred software updates, among other reasons, again leading to vulnerable applications [9].

3 Executing Untrusted Code in Educational Contexts

In this section, we systematically analyze the typical approaches of test systems with respect to their security properties. We focus on 11 test systems from computer science university classes, including computer security classes. All test systems were in productive use in the time frame from October 2023 to February 2024, with 57 to 770 students per course, totalling to 3821 students.

3.1 Decentral Development of Test Systems

A solution to the resource problem of reviewing student code is automation [34, 26] with so-called "test systems" [35]. However, test systems are often developed and deployed by educators of the courses rather than professionals aware of security best practices. Several of the test systems we analyzed were developed over the past 3 years and, in some cases, moved away from previously unified and centralized test systems due to the specific requirements the different courses have. Hence, there are different strategies to develop such systems. Among the 11 test systems³ we analyzed, 6 opted for GitLab Runners with Docker registries. This is in line with the best practices recommended by Paiva et al. [26], using containers to run student code in isolation.

One test system used GitLab Runners but with a custom pipeline. In this testing system, each push triggers the execution of a Python script. The script builds and spawns three separate Docker containers that communicate with each other. This approach allows for more precise management of the container spawning process and isolation between different parts of the test system and untrusted code, e.g., checks are not performed within the same container and also not directly on the host system but run in another container, to also mitigate exploitation of this code. However, this approach requires significantly more upfront development and maintenance.

Four test systems were not based on CI systems and developed from scratch. The first (C4) is testing student code inside virtual machines, that are run inside a chroot environment, following the recommendations by Paiva et al. [26] and Wilcox [35]. This is also the case for the other three: System C8 is testing student code by compiling it and running unit tests within a reference Ubuntu virtual machine without any further isolation. Systems C10 and C11 are testing student code with custom **bash** and **python** scripts inside Docker containers.

Table 1 provides an overview of the test systems, programming languages, and educational context. The four systems-related courses use custom test systems (prefixed with a **C**), whereas all others rely on a GitLab-based approach (prefixed with a **G**). Language-wise, there is a focus on C and C++, and in about half of these systems, the build environment is also controlled by the user. Finally, for the interfaces, most test systems expose an interface via Git, e.g., commit hooks or CI pipeline triggers, and to the web to check the results. Most test systems directly rely on the GitLab interface for the students.

³ We note that one of the systems was set up by one of the authors of this paper.

System	Educational Context	Languages	User Interface	Students
GD1	Introductory	С	Git	770
GD2	Introductory	\mathbf{C}	Git	366
GD3	Introductory	C++	Git	547
C4	Systems	C, C++, Assembly	Git, Web	134
GD5	Formal Methods	Python	Git	125
GD6	Privacy	C++, Python	Git	57
GC7	Software Development	Java	Git	413
$\mathbf{C8}$	Systems	C++	-	547
GD9	Security	Python	Git	413
C10	Systems	C, C++, Assembly	Git, Web	374
C11	Security	С	Web	75

 Table 1. Overview of the analyzed test systems.

3.2 Threat Model of Running Untrusted Code

In our threat model, we assume malicious users intentionally submitting code intended to exploit the test system. We assume the user has no knowledge about the test system other than publicly available to all participants. We assume that the host system spawning the test containers is running on the most recent kernel and that it uses the most recent packages from its distribution. Furthermore, we assume established software security features to be enabled, e.g., KPTI [13], ASLR and KASLR [27,6], hardware-assisted control-flow integrity [1,16]. We assume the educators' credentials are appropriately chosen, secure against unpermitted access, and possibly even protected with second-factor authentication.

For both GitLab Runners with Docker registries and GitLab Runners with custom pipelines, we assume that student code is usually run in a container and not directly under the user account of the educator. We also assume that GitLab itself is up-to-date, *i.e.*, no exploitation of bugs in the GitLab implementation.

A malicious user can obtain information such as the total runtime, log files, and other custom artifacts the test system provides. The goal is to extract information about the test system, the test cases, and access confidential data and configurations, also confidential data unrelated to the course.

4 Security of Real-World Educational Test Systems

Educational test systems execute untrusted code by hundreds of potentially malicious students. They also hold various sensitive information, e.g., test cases, solutions, student records, and confidential data of the educators. In our analysis, we follow a 3-step approach: First, the security-critical assets on the test system are identified (Section 4.1). Secondly, intentional features of the educational test system that may facilitate attacks are examined (Section 4.2). Finally, the test system has to be assessed for security vulnerabilities, with the unique setting of educational test systems in mind (Section 4.3).

4.1 Asset-oriented Security Evaluation

In this section, we discuss which assets might be of interest to an attacker. We want to emphasize that this discussion is not comprehensive. However, there are not only assets that are of direct interest to the attacker (primary interest assets) but also assets that are useful in intermediate steps (secondary interest assets). **Primary Interest Assets.** We consider assets to be of primary interest if they contain information the attacker is directly interested in: Such information includes everything that gives the attacker an unfair grading advantage, as well as everything that directly breaks user privacy and confidentiality.

To gain a grading advantage, the attacker might want to exfiltrate nonpublicly available test cases, reference solutions, or other students' solutions. The attacker might develop against the test cases in a trial-and-error manner or copy exfiltrated solutions, without actually reaching the learning goals. A test system storing grades or points might also leak other students' grades, real names, email addresses, contact data, or other personally-identifying information of the students or the educator, violating privacy and confidentiality.

Secondary Interest Assets. We consider assets to be of secondary interest if they provide an intermediate step to obtain primary interest assets. Such assets include credentials or access tokens protecting confidential data, e.g., private repositories or container registries. While they do not contain the critical information *directly*, the attacker can use them in a subsequent step to gain access.

For example, the test system might leak the access token for the container registry, exposing test cases provided in a private docker container. If, due to misconfiguration, the access token even grants write access, the attacker can perform unauthorized modifications to the container. Similar attacks can be imagined for access tokens protecting external test case repositories.

As another example, code execution as root might also be used to bypass access control [33] and, in turn, to obtain a primary interest asset. The same argument applies to container-to-host escapes. Furthermore, injecting malicious code into the test pipeline might lead to primary interest assets.

4.2 Continuous Integration for the Attacker

Using CI systems to test student code has the advantage of providing feedback quickly. Usually, within minutes after submitting their code, systems give students feedback and a log file, indicating which tests passed. However, while such short feedback loops are desired under normal circumstances, these might also be helpful to an attacker. Attackers, too, get immediate feedback from the test system, allowing them to test their exploits frequently and quickly.

The feedback usually includes a log file of the test run, which also typically includes any custom output of the user program. If attackers gain unauthorized access to security-critical information, they can directly print it and exfiltrate it through the log. Similarly, information might also be exfiltrated via other artifacts the pipeline creates. In addition, if Internet access is not blocked, the attacker might send critical data to a web server they control.

Even without log files, artifacts, or Internet access, the attacker can exfiltrate data via covert channels, e.g., by encoding a byte in delayed execution time dependent on the numeric value of the byte to leak. A multi-byte secret could be transmitted in multiple pipeline executions, allowing the attacker to reconstruct the secret byte-by-byte from the execution times.

4.3 Categorization of Security Issues

To identify the issues in test systems, we analyze the systems with respect to the following four categories of security issues:

Output sanitization. As Paiva et al. [26] note, a secure test system should block any attempt to leak sensible test data, e.g., "output data", to the outside. However, for virtually all systems we analyzed, the output data is intended to be seen by the students. This setup introduces a significant challenge, requiring filtering illegitimate output from legitimate output. This challenge is far from trivial to solve, as an attacker can choose an arbitrary encoding for the secrets. **Permission issues.** The test system must protect sensitive information from the execution of student-provided code, *i.e.*, access to critical assets should be denied to untrusted code. The operating system supports this by enforcing file access permissions or limiting access to other resources, such as processes. However, this requires that the permissions are set correctly by the test system. For example, if student code runs under the same system account as the test case owner, a malicious student could exfiltrate the test cases.

While it is challenging to design a test system with correct separation, the confidentiality of the test cases and the integrity of the test system is still crucial. **Misconfiguration issues.** Misconfiguration is a common source of security problems. One of the most efficient ways to leak information about the system or data is to use the Internet. If the test system allows Internet access, the attacker can exfiltrate data to a web server under their control.

Another common misconfiguration is the use of outdated software. While we assume the host system running the test containers to be up to date, this might not necessarily be the case for the container images, where outdated software might also be exploited. A large-scale study by Shu et al. [29] showed that about 30% of the Docker images available on Docker Hub have not been updated within 400 days when that study was performed. Container images might also ship custom software that is no longer maintained, making updating the images challenging. Educators may be negligent with updating containers.

Environment sanitization issues. The test system must prevent students from modifying the environment in a way that would yield unauthorized access to the system or test cases. For instance, students should not be able to modify the build system by modifying the Makefile. Therefore, build files are often replaced with a reference file or checked for modifications by the test system.

Similarly, with GitLab Runners, the .gitlab-ci.yml file defines the pipeline. However, this file should not be modifiable by the student, as this would allow the student to modify the pipeline. Consequently, students can change pipeline variables such as the entry point or add additional steps to the pipeline.

Security Issue	GD1	GD2	GD3	C4	GD5	GD6	GC7	C 8	GD9	C10	C11
Unsanitized Output	×	×	×	X	X	X	1	1	1	1	1
Timing Side Channels	×	×	×	~	×	×	~	~	×	~	~
No Container	1	1	~	1	1	~	1	X	1	1	1
Privileged Git Tokens	X	X	×	\checkmark	~	~	\checkmark	×	\checkmark	\checkmark	\checkmark
Reference Solution	×	×	×	\checkmark	×	~	1	×	\checkmark	\checkmark	\checkmark
Exec as Root	~	~	~	1	×	\checkmark	~	1	\checkmark	×	×
DOCKER_AUTH_CONFIG	×	×	×	1	×	×	1	1	×	1	~
ptrace	\checkmark	\checkmark	1	X	×	×	~	×	\checkmark	×	\checkmark
Internet Access	×	~	~	X	1	\checkmark	×	X	\checkmark	\checkmark	\checkmark
Outdated Software	\checkmark	~	1	X	1	\checkmark	\checkmark	1	\checkmark	\checkmark	\checkmark
Docker Socket Mount	1	1	~	✓	~	~	×	✓	1	~	\checkmark
Modify Pipeline	×	X	X	1	1	1	X	1	1	1	~
Build System Takeover	×	×	×	×	\checkmark	~	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark

Table 2. Test systems and what attack vectors they mitigated. \checkmark means vulnerable, \checkmark means not vulnerable, \sim means inconclusive.

5 Case Studies

Following the methodology described in Section 4, we analyzed 11 test systems. As Table 2 shows, each of them was affected by at least one vulnerability. In this section, we briefly summarize our most critical findings before we continue by discussing 3 case studies in more detail.

Out of the 11 systems, 6 systems did not sanitize their output. At least 4 of them allowed the tested code to establish outbound Internet connections, and none of them fully mitigated timing side channels. Consequently, on all of these, if student code has access to sensitive information, it can also exfiltrate it from the test system, as we discuss in all our 3 case studies in more detail.

All of the 6 systems based on GitLab runners and Docker registries exposed credentials for the registry to untrusted student code via the DOCKER_AUTH_CONFIG environment variable. Effectively, this gives an attacker read access to the entire Docker container, possibly including reference solutions and test cases. In at least 3 cases, the exposed credentials even granted write permissions to the Docker registry, enabling an attacker to modify the container image persistently. For instance, this misconfiguration issue affects the test systems **GD5** and **GD1**, discussed in Section 5.1 and Section 5.2, respectively. We emphasize that even following best practices, this exposure of credentials is not documented as a security risk and even following the documented best practices on securing Gitlab runners and Docker registries leaves this security issue unmitigated.

On 5 of the test systems, the reference solutions were accessible for an attacker, either directly by examining the docker container or during the execution of the tests (e.g., due to misconfigured file permissions). Examples of this are provided in Section 5.1 and Section 5.2 for the **GD5** and **GD1** test systems.

Furthermore, on 4 test systems, attackers could manipulate the test pipeline, enabling them to inject arbitrary commands and run unauthorized code. For instance, we found this environment issue in the GD1 test system (cf. Section 5.2). Similarly, on at least 5 systems, attackers could inject arbitrary commands into the build process by modifying the Makefile or other build scripts. For instance, the **GD1** and **C4** test systems both had this issue (cf. Sections 5.2 and 5.3).

5.1Case Study 1: Test System GD5

Our first case study is **GD5**, and we investigated the 4 categories of security issues outlined in Section 4. GD5 is a system based on GitLab Runners, combined with a Docker registry, which is not publicly accessible. Students can push code into their GitLab repository, which is then tested within a Docker container by the GitLab Runner. Subsequently, the log file is available as an artifact to the students. Notably, the test cases and runtime are not randomized and highly stable, *i.e.*, 6 seconds with an empty submission where no tasks have been solved. **Output Sanitization.** As a first category, we analyzed the system's output sanitization. Given the intention to provide users with the output, we observe that system **GD5** does not sanitize the output at all. However, this introduces significant security issues, as a malicious user can print any data of interest into the log file, which is then accessible through GitLab's web interface, *i.e.*, as an artifact. While this does not imply that leakage of valuable information is possible, it is a direct path to information leakage when combined with any misconfiguration, permission issue, or missing environment sanitization.

For completeness and going beyond the issues Paiva et al. [26] identified based on their literature survey, we also tested the timing channel as a possible output vector. In contrast to related work, which focused on guessing the class of environment from inside a sandbox [22], we leak environment variables or user credentials. These are often stored as **base64**-encoded strings (typically 26 characters), and hence, we focus on leaking such base64 strings. We encode and transmit character by character, one per test run. For a single character, we let the GitLab Runner sleep between 0 and 64 seconds, depending on the base64 character to leak. Thus, after deducing the base runtime, the attacker can infer the specific character transmitted from within the GitLab Runner. Consequently, we can leak a GitLab authentication token within 156 s to 1820 s, with an average around 988s.

Permission Issues. Primary assets for a malicious user are test cases and the reference solution. We analyzed system **GD5** and discovered three main issues: First, the reference solution was accessible to the malicious user from within the Docker container, *i.e.*, it was bundled into the Docker container. It is not clear why this is necessary for testing, as the test system uses test cases. Hence, a solution is to not include unnecessary secret data in the Docker containers. Second, however, the test cases were also accessible to the malicious user from within the Docker container, also bundled into the Docker container. In line with Paiva et al. [26], neither the reference solution nor the test cases should be available to the user. A solution could be to restrict the permissions, e.g., provide

10

not read permissions to the user executing the student code. Finally, even if these permission issues were fixed, the student-provided submission is executed with root privileges inside the Docker container. Consequently, the student effectively has unrestricted access to all data within the Docker container, regardless of other restrictions, e.g., file permissions, made. Furthermore, root privileges within containers are particularly dangerous when combined with misconfiguration issues, e.g., unnecessary provision of capabilities to the container.

Misconfiguration Issues. As a third attack vector, misconfiguration issues could expose confidential information, including secret test cases or a reference solution. We identified two main issues in system GD5: The first issue is the provision of the ptrace capability. As we already noted, in system **GD5**, untrusted code runs with root privileges within the containers. However, this combination of root privileges and the **ptrace** capability effectively allows the student code to attach (*i.e.*, like a debugger) to any process within the container including the parent process responsible for running the tests and evaluating the output of the student-provided code, *i.e.*, the process with full access to the test cases. The second issue is the **implicit** provision of the DOCKER_AUTH_CONFIG environment variable by the GitLab runner. GitLab runners use the DOCKER_AUTH_CONFIG to access an image from a private container registry [11]. However, it is not documented that containers **implicitly** inherit this environment variable, exposing the credentials to authenticate against the private container registry. Consequently, neither GD5 nor any of the other test systems we tested unset this environment variable. We emphasize the significance of this issue, as the DOCKER_AUTH_CONFIG credentials often include both read and write permissions to the container registry. Thus, this misconfiguration not only exposes the confidentiality of the test cases of the educators: A malicious user could overwrite the existing image in the container registry with their own image, running arbitrary code on every subsequent push of every student.

Environment Issues. For completeness, we also investigated the environment issues but did not identify any particular issues that would allow for a takeover of the build system or pipeline modifications.

5.2 Case Study 2: Test System GD1

Our second case study is system **GD1**, which we analyzed as described in Section 4. **GD1** is similar to our first case study test system, **GD1**, which is also based on GitLab runners combined with a Docker registry.

With each git push, the student code is tested inside a Docker container by the GitLab Runner. Once the test run is complete, students can view the corresponding pipeline log. Additionally, students are provided with a report.html document that includes both public and a select few redacted private test cases, along with their corresponding inputs and outputs for the public test cases.

Output Sanitization. The first category we analyze is the output sanitization of the test system. The output data is intended for students to receive feedback on their submissions. In system **GD1**, the output data for each public test case can be obtained from the **report.html** document. In addition, the pipeline offers

metrics during runtime, such as testing times. We observe that no output sanitization occurs for the output data in the **report.html** document. Consequently, a user with malicious intent can print any data of interest, such as environment variables and the file system structure.

We also tested the timing side channel as a possible output vector. The test cases have a constant runtime, which allows us to leak environment variables such as the DOCKER_AUTH_CONFIG environment variable encoded in the runtime. The approach we follow, is the same as the one described in the first case study: We encode and transmit character by character, one per test run, where each character is represented by 1 second of timing delay. More efficient leakage is possible if sub-second runtime information is available.

Permission Issues. In test system **GD1**, grading is based on the passed test cases. There are both public and private test cases. Students have access to the input and output data for all public test cases, but private test cases are always redacted in the test report. Therefore, the test cases and the reference solutions are the most interesting resources for a potential attacker. Based on this assumption, we identified two issues in **GD1**:

Before building and testing the student's submission, the test pipeline retrieves the latest test cases from private repositories. Authentication is required to pull from a private Git repository. Hence, in **GD1**, GitLab API tokens are provided via files, accessible to the unprivileged user. An attacker can leak an API token through unsanitized output or side channels. The attacker can then use these tokens to access and inspect repositories on a local machine. This cannot only result in data leakage but also more severe security issues: API tokens often have not only read but also write privileges. As a result, by modifying files in the private repository, a student with malicious intent could delete or add test cases, or even execute arbitrary code. In combination with misconfiguration errors, this can lead to more devastating security compromises.

Secondly, the private Git repositories contain not only the test cases but also the reference solution to the assignments. As the test system does not require the reference solution, it should also not be present while testing the student's submission with the test cases. This is a significant issue, as the reference solution is the most valuable resource for a potential attacker in our scenario.

Misconfiguration Issues. System GD1 suffers mainly from two misconfiguration issues. The first issue is yet again the presence of the DOCKER_AUTH_CONFIG environment variable. As we noted earlier, the DOCKER_AUTH_CONFIG credentials often include both read and write permissions to the container registry. Combined with the fact that the test system did not sanitize the output data, the leakage of the DOCKER_AUTH_CONFIG credentials is a significant problem. We also verified that the credentials provided had write permissions to the container registry. As a result, a malicious user could have overwritten the existing image in the container registry with their new image.

The second issue is that outbound Internet connections are not blocked throughout the entire pipeline. Internet connections are only blocked during the build and test stages. As mentioned before, the pipeline requires this connectivity to retrieve the latest test cases from private repositories. Once the test cases are retrieved, the Internet connection is blocked for the duration of the tests. After the tests are completed, the pipeline can connect to the Internet and upload artifacts to a dedicated server.

Environment Issues. The fourth category of attack vectors are environment issues. System GD1 also suffers mainly from two environment-related security issues: First, students receive a .gitlab-ci.yml file containing the pipeline configuration for the test system. This file defines the pipeline stages, the jobs executed in each stage, and their order. The .gitlab-ci.yml file is stored in the student's repository and executed by the GitLab Runner. It is a critical asset for a potential attacker, as it can be modified to manipulate the pipeline. For instance, an attacker can modify the entrypoint of the Docker container to run unauthorized code. Additionally, the attacker obtains a command injection vulnerability since the script sections of the .gitlab-ci.yml file are executed as shell commands following the Docker container entrypoint.

Secondly, the build process relies on the Makefile provided by the student. The Makefile contains the build instructions for the student's submission. However, the test system does not check the integrity of the Makefile or the commands it contains. This is a significant issue, as the Makefile can be altered to execute arbitrary commands instead of compiling the student's code.

5.3 Case Study 3: Test system C4

For our third case study, we investigated C4. C4 is a completely custom and independently developed system for testing low-level system code.

After students push their code into a GitLab repository, a web hook triggers the system to pull the submission and to execute the following two steps: First, the student code is built in a chroot environment following the best practices described by Paiva et al. [26]. The chroot environment is a separate root file system that contains all the necessary tools and libraries to build and test the student's submission. Secondly, the resulting binary is tested within a customized QEMU fork to obtain specific low-level event metrics. From each test run, students get the build log and a report of unintended behavior or crashes. **Output Sanitization.** We again first analyze the system's output sanitization. The system performs output sanitization in several places, yet not in all stages of the test pipeline. For example, an attacker can write arbitrary text during the build and startup phases. As the test cases reside in the chroot tree of the build process, an attacker can directly exfiltrate them by writing them to the log. Furthermore, the test cases are included in the built image, allowing the attacker to dump them into the log during the startup phase.

Permission Issues. For completeness, we checked for permission issues but did not identify any particular security-relevant issues.

Misconfiguration Issues. We observed three major misconfiguration issues:

Firstly, outbound Internet connections are unrestricted while running the entire pipeline. This allows an attacker to exfiltrate arbitrary information to a server, enabling similar attacks as with incomplete output sanitization.



Fig. 1. Severity assessment by the participants.

Secondly, C4 uses outdated software versions in multiple places, e.g., a QEMU fork unpatched against multiple published vulnerabilities [23, 24], allowing the tested code to escape the virtual machine. Furthermore, the software within the chroot environment has also not been updated for years. For example, the sudo binary is vulnerable against CVE-2021-3156 [25], enabling an attacker to gain root privileges and escape the chroot environment.

Thirdly, **C4** runs pipelines from different students always under the same user and does not prohibit **ptrace** across different processes of the same user. Thus, an attacker can embed code into the build phase, attaching itself to the pipeline of another student. Hence, the attacker can read memory or files within the victim's **chroot**, obtaining the solution of other students. The attacker can also hijack the victim's pipeline process to execute arbitrary shell code, e.g., to disrupt the victim's process and manipulate the test results.

Environment Issues. Our investigation of environment issues yielded two command injection vulnerabilities. First, the system executes several helper programs during the build process. These are built from source files, which can be overwritten by an attacker to execute arbitrary code in the chroot environment, enabling the attacks discussed above. Secondly, an attacker can also overwrite the Makefile, enabling similar attacks.

6 User Study: Severity Assessment

To assess the severity of the security issues, we performed a user study with 6 educators responsible. The severity assessment of the read access (Figure 1a) is strongly connected with the frequency with which student assignments change. For courses where the assignments change every semester, the severity is assessed

as low. The main reason given was that there can always be communication between students leading to plagiarism. Therefore, solutions must always be checked for plagiarism, uncovering copied solutions from the test system. The medium-high assessment of one participant was reasoned by possible access to the solutions of assignments and because it is difficult to detect plagiarism. The severity of write access (Figure 1b) to the solutions of other students was assessed low because manipulation could easily be detected and there is little to gain.

The severity of read access to the test cases (Figure 1c) is assessed low for courses where the test cases are public and use randomized inputs on the test system. In this case, there is no advantage for an attacker. Most participants assessed the severity medium because access to the test cases could allow them to reverse engineer the solution. For courses where the test cases do not change frequently, accessing them would have high severity. Manipulating test cases with write access (Figure 1d) would be detected, *i.e.*, has a low severity. Still, it could be used to interfere with other groups.

The severity assessments of read or write access to all files in the test container (Figure 1e and Figure 1f) are very similar to the assessment of accessing the test cases. The reason is that the test systems do not contain other secret data that an attacker could exploit. One participant notes that modifications of student grades would be evident in log files.

In most cases, read access to the container registry (Figure 1g) gives an attacker access to the test cases. This may be a security issue if the test cases are private and not frequently changed. On the test system of one participant, read access to the container registry gives an attacker access to the reference solution. On one test system, the container does not contain the test cases. The severity of a write access (Figure 1h) is assessed as medium, based on a few factors, including the possibility to leak all student solutions. In some courses, the containers from the registry are also run locally on student's and educator's systems. If started in privileged mode or even without the container, which educators acknowledged to happen in practice, this could cause a complete exploitation of a student's or an educator's system. Still, they felt this is unlikely to happen.

Access to the host system (Figure 1i and Figure 1j) provides an attacker access to sensitive student records, including students' points and grades. Educators typically do not know or are in charge of the host system and they expressed the concern that access to the host will give an attacker also access to other information not related to their course. The severity assessment for execution permission (Figure 1k) is the same as for write access on the host system, as they ultimately give an attacker the same capabilities.

In multiple cases, educators ran student code on their own local system. Educators who also engaged in vulnerability disclosures as part of their scientific work reported that they malicious code and containers could theoretically attempt to obtain embargoed information from their systems. Consequently, a real-world risk is not only the disclosure of sensitive student records stored on the educator's system but also the breach of a vulnerability embargo if information about the vulnerability is leaked by a test system user.

7 Discussion and Limitations

Hardening educational test systems against security vulnerabilities is challenging, given that they have to execute untrusted code while they also process a variety of security-critical information. Furthermore, resources to develop and maintain these test systems are often limited, highlighting a more generic issue with the current security best practices. Automated approaches have been presented for similar cases in cloud scenarios and could potentially be adopted for the test system use case [2]. Security-critical assets have to be identified (see Section 4.1), and it is crucial to reduce these to a minimum. For example, reference solutions and other assets should not be stored on the test system if they are not required for testing the submissions.

Providing users with the precise execution time of their tests, inherently opens a timing side channel. Therefore, test system maintainers should consider hiding the execution time from the students. Common misconfiguration errors also include read access to student or testcase repositories, access to Internet or communication sockets, but also setups that implicitly lead to outdated software, e.g., manually patched software or environment configurations that prevent automatic updates. Another common error here are container capabilities that are not required for testing, such as the **ptrace** capability. As these capabilities usually configure the interaction between processes inside the container and the host kernel, they implicitly allow bypassing the sandbox boundaries in certain ways. It is crucial that test system maintainers carefully review the capabilities granted to a container. Furthermore, environment issues allow an attacker to take over the build process or the entire test pipeline. Test system maintainers should ensure critical control files are not overwritten by student code.

While arguably students may be most interested in bypassing the security of the test system in an attempt to learn the test cases and improve their own scores illegitimately, it is also comparably easy for a malicious actor to have a person register as a student and then register for a course, in order to attack the system of an educator frequently engaged in vulnerability disclosures. This attack vector comes at a low cost and entry hurdle in many countries and with a very low risk of legal consequences.

8 Conclusion

We evaluated the security of 11 real-world test systems from computer science university classes, including computer security classes. We studied these systems during the time frame from October 2023 to February 2024, involving 3821 student accounts, and provide a systematic overview of the typical approaches these systems follow. We found the potential security impact of a compromise to be the exposure of hundreds of highly sensitive student records, confidential research data, and in some cases even embargoed vulnerabilities. Our results highlight that the real-world challenges to run and maintain secure test systems are not solved in practice. We emphasize the need for new systematic security approaches to secure this very common type of software system.

References

- Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-Flow Integrity. In: CCS (2005)
- Abbadini, M., Beretta, M., Facchinetti, D., Oldani, G., Rossi, M., Paraboschi, S.: Lightweight Cloud Application Sandboxing. In: CloudCom (2023)
- Bosamiya, J., Lim, W.S., Parno, B.: Provably-Safe Multilingual Software Sandboxing using WebAssembly. In: USENIX Security (2022)
- Capovilla, D., Berges, M., Mühling, A., Hubwieser, P.: Handling heterogeneity in programming courses for freshmen. In: Learning and Teaching in Computing and Engineering (2015)
- Dietrich, C., Krombholz, K., Borgolte, K., Fiebig, T.: Investigating system operators' perspective on security misconfigurations. In: CCS (2018)
- Edge, J.: Kernel address space layout randomization (2013), https://lwn.net/ Articles/569635/
- 7. Edge, J.: A seccomp overview (2015), https://lwn.net/Articles/656307/
- 8. Eshete, B., Villafiorita, A., Weldemariam, K.: Early detection of security misconfiguration vulnerabilities in web applications. In: ARES (2011)
- 9. Ferrari, D., Carminati, M., Polino, M., Zanero, S.: Nosql breakdown: A large-scale analysis of misconfigured nosql services. In: ACSAC (2020)
- 10. GitLab: GitLab Runner (2024), https://docs.gitlab.com/runner/
- Gitlab: Run your CI/CD jobs in Docker containers (2024), https://docs.gitlab. com/ee/ci/docker/using_docker_images.html
- GitLab B.V.: Security for self-managed runners (2024), https://docs.gitlab. com/runner/security/
- 13. Gruss, D., Hansen, D., Gregg, B.: Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. USENIX ;login (2018)
- 14. Hilton, M., Nelson, N., Tunnell, T., Marinov, D., Dig, D.: Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In: ESEC/FSE (2017)
- 15. Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D.: Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In: ASE (2016)
- 16. Intel: Control-flow Enforcement Technology Preview (6 2017), revision 2.0
- 17. Jenkins: Jenkins (2024), https://www.jenkins.io/
- Kubincová, Z., Homola, M.: Code review in computer science courses: Take one. In: Advances in Web-Based Learning (ICWL). Springer (2017)
- Medeiros, R.P., Ramalho, G.L., Falcão, T.P.: A systematic literature review on teaching and learning introductory programming in higher education. IEEE Transactions on Education 62(2), 77–90 (2019)
- 20. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux Journal (2014)
- Mitra, J.: Studying the impact of auto-graders giving immediate feedback in programming assignments. In: ACM Symposium on Computer Science Education (SIGCSE) (2023)
- Nappa, A., Papadopoulos, P., Varvello, M., Gomez, D.A., Tapiador, J., Lanzi, A.: PoW-How: An Enduring Timing Side-Channel to Evade Online Malware Sandboxes. In: ESORICS (2021)
- NIST National Vulnerability Database: CVE-2019-14378 (2019), https://cve. mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14378
- 24. NIST National Vulnerability Database: CVE-2020-24165 (2019), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-24165

- 18 S. Gast et al.
- 25. NIST National Vulnerability Database: CVE-2021-3156 (2021), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156
- Paiva, J.C., Leal, J.P., Figueira, A.: Automated assessment in computer science education: A state-of-the-art review. ACM Transactions on Computing Education (TOCE) 22(3), 1–40 (2022)
- 27. PaX Team: Address space layout randomization (ASLR) (2003), http://pax.grsecurity.net/docs/aslr.txt
- Rahman, A., Shamim, S.I., Bose, D.B., Pandita, R.: Security misconfigurations in open source kubernetes manifests: An empirical study. ACM Transactions on Software Engineering and Methodology **32**(4), 1–36 (2023)
- Shu, R., Gu, X., Enck, W.: A Study of Security Vulnerabilities on Docker Hub. In: CODASPY (2017)
- Shu, R., Wang, P., Gorski III, S.A., Andow, B., Nadkarni, A., Deshotels, L., Gionta, J., Enck, W., Gu, X.: A Study of Security Isolation Techniques. ACM Computing Surveys (CSUR) 49(3), 1–37 (2016)
- Ståhl, D., Bosch, J.: Automated software integration flows in industry: a multiplecase study. In: ICSE Companion (2014)
- 32. Travis CI: Home (2024), https://www.travis-ci.com/
- De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: Access control: principles and solutions. Software: Practice and Experience 33(5), 397–421 (2003)
- Wilcox, C.: The role of automation in undergraduate computer science education. In: Computer Science Education (CSE) (2015)
- 35. Wilcox, C.: Testing strategies for the automated grading of student programs. In: Computing Science Education (CSE) (2016)
- Wong, A.Y., Chekole, E.G., Ochoa, M., Zhou, J.: On the Security of Containers: Threat Modeling, Attack Analysis, and Mitigation Strategies. Computers & Security 128, 103140 (2023)
- 37. Wrenn, J., Krishnamurthi, S., Fisler, K.: Who tests the testers? In: International Computing Education Research (ICER) (2018)
- Yong Wong, M., Landen, M., Antonakakis, M., Blough, D.M., Redmiles, E.M., Ahamad, M.: An Inside Look into the Practice of Malware Analysis. In: CCS (2021)
- Zhang, J., Renganarayana, L., Zhang, X., Ge, N., Bala, V., Xu, T., Zhou, Y.: Encore: Exploiting system environment and correlation information for misconfiguration detection. In: ASPLOS (2014)