Revenge for Late Nights: Penetration Testing of University Autograders

Ian Pudney
University of Michigan
ipudney@umich.edu

Ryan Wawrzaszek
University of Michigan
ryanwawr@umich.edu

Austin Yarger
University of Michigan
ayarger@umich.edu

Abstract

Whether it be the promise of world-changing technologies or the allure of well-paying jobs, students are flooding into Computer Science programs at universities worldwide. Such an influx puts stress on traditional methods of assignment evaluation, and has led to an enhanced emphasis on autograding systems — software that takes student code and evaluates it automatically.

In this paper, we analyze the potential for cheating or exploiting such autograders by investigating several real-world systems in use at the University of Michigan. We analyze the defense implemented by autograding systems and demonstrate vulnerabilities in several of them by mounting attacks ranging from privilege escalation to exfiltration of sensitive test cases.

Our findings indicate that the most secure autograders adhere to a "defense in depth" strategy, relying on a combination of security mechanisms that include strict authentication, comprehensive logging, and the use of professionally developed open-source frameworks for sandboxing and containerization of untrusted code.

1 Introduction

Academic cheating, a common result of procrastination and anxiety, manifests traditionally in many forms, including copying answers from another student, altering the gradebook, or obtaining exam solutions ahead of time key. As grading becomes more digital and automated, one must consider these threats in a new digital context, and how our technological approach to grading might be hardened against modern threats to academic integrity.

In sections 2 and 3, we introduce the concept of autograders and establish a simple submission-evaluation model for autograding systems in general. In sections 5-8, we report on the architecture of four distinct autograders in use at the University of Michigan, discussing their strengths and weaknesses, and illustrating our attacks against them and potential defenses. We conclude by distilling some best practices and general security tips for autograder development in general.

2 Related Work

Literature on autograder security is fairly rare, and many papers mention it only in passing [2]. However, a core requirement of autograders is the establishment of a sandbox for evaluation purposes, on which plenty of literature can be found [4, 3]. This paper investigates at a higher level, considering the use of sandboxes in autograder security, but not the low-level details of sandbox implementation.

3 Autograder Model

The role of the typical autograder can be split into two primary tasks– the submission of code, and the evaluation of code. Both include distinct security challenges which are outlined in this section.

In the discussions that follow, "student" refers to someone who submits code to an autograder for evaluation.

3.1 Submission

A student looking to complete an autograded assignment must, at some point, submit that code to be graded. There are a few interesting points of investigation here; namely, Authentication (is the submission taking place on behalf of the proper student?), and Injection (is there something about the submission interface that allows for the unintended execution of scripts or code?).

3.2 Evaluation

Upon submission, code must be compiled, executed, and evaluated. Autograders are unique in that foreign code injection and execution is expected. This presents an ideal situation for an attacker.

Potential attacks can be mitigated by adhering to a few general principles. First, untrusted code should be run in a restricted environment with only the level of privilege necessary to complete the evaluation. Code should be run in a sandbox, and potentially dangerous system calls should be prohibited.

Additionally, the party submitting code to the autograder should not be able to influence the evaluation process once it has begun. A student submitting for a grade has significant incentive to compromise the integrity of an evaluation; therefore, the student should not control the machine in which the autograding system executes.

Finally, test cases, inputs fed to a student program to test for correct output and functionality, serve as a primary means of evaluation in autograder systems. Should test cases become publicly available, the integrity of the evaluation is compromised, as a student knowing the test cases may rely on hard coding outputs for specific cases, rather than developing a correct general solution. Therefore, test case exfiltration must be prevented.

4 Methodology

A study of autograder security benefits heavily from access to production-level autograders. Our team received permission from the staff of four University of Michigan Computer Science courses to perform penetration tests against their autograders. In one of the courses, a dummy system was set up specifically for our testing. In another, we were given access to a virtual machine containing an instance of the autograder. In yet another, our team was allowed to penetration test the production autograder. To obtain these permissions, we agreed not to publish autograder source code or test cases. For the course in which we penetration tested the live autograder, we imposed our own restriction that we would not mount attacks if those attacks might disrupt autograder functionality for normal students.

Prior to the presentation of our results, each course staff received a disclosure document outlining discovered vulnerabilities and giving detailed recommendations on how to fix them

5 EECS 281

EECS 281 (Data Structures and Algorithms) is a course at the University of Michigan dedicated to writing efficient code. While merely getting the correct answer is sufficient for a perfect score in many classes, EECS 281 students must write correct code that adheres to strict time and memory limits, which many students struggle with. These efficiency limits, which are unique to this course, will be the focus of our investigation.

5.1 Submission Architecture

Student code is compressed into a tarball (.tar.gz file) and submitted for grading via a web form. Submission results become available on the web form in real time as grading occurs. Results indicate whether the program output was correct, how much time and memory the submission used, the time and memory limits, and the program termination condition, such as a signal. If the student's submission is incorrect, the web form also provides the student a small section of output that differs.

5.2 Evaluation Architecture

After a student submits, their code is unpacked from the tarball, compiled, and run with various text inputs. Student code is not run directly as a child process of a grading script. Instead, it is run as a child of a custom "jail" program, which uses `chroot` to restrict the execution of a student's program to a particular directory and `ptrace` to disallow certain system calls. The `ptrace` API follows the POSIX standard for debugging, providing features such as breakpoints and signal catching. Upon receiving a submission, the jail script first `fork`s a child, then, before that child runs the student executable, it attaches itself to the child via `ptrace` and notifies the jail whenever the student's code makes a system call. If this call is not in the whitelisted set of system calls, the jail terminates the submission with an `EPERM` ("Operation not permitted") signal.

Once the student's code has finished, the autograder compares its output to the expected output, typically with a simple `diff`. If the output is correct, the autograder computes the program runtime as the sum of the `user` and `system` time and computes the memory the program used as the maximum size of the valid portion of the program's arena. These metrics are then used to compute the student's score.

5.3 Exploitation

Jailbreak

Many interesting functions, such as `fork()` and `clone()`, are prohibited by the jail. `vfork()`, however, is not. `vfork()` is an optimized version of `fork()`, which creates a new process but shares the parent's page table until either `_exit()` or an `exec`-family function is called. If the child process does anything other than call one of those functions, its behavior is undefined. We were able to take advantage of this by using `vfork()` to create a child process that was not controlled by the jail and that could, therefore, make system calls which would otherwise be prohibited. This allowed for several interesting exploits that will be discussed below.

Test Case Exfiltration

The EECS 281 autograder runs student code with a variety of different inputs. It is imperative that these test cases remain secret, else students could cheat by simply hardcode their programs to give the correct output for each specific test. One way students could steal these test cases would be to submit a program that connects to a remote server controlled by the student and sends the test input to that server. To ensure these test cases remain secret, the EECS 281 autograder jail prevents students from accessing the network by prohibiting the `socket()` system call. However, after executing the jailbreak described in the previous section, we were able to upload such a specially crafted submission and exfiltrate the test cases from the autograder.

Computational Overhead Hiding

The EECS 281 autograder measures not just the correctness of student programs, but their runtime and memory efficiency as well. If a student could ”offload” computation to somewhere other than the program being measured, they could circumvent the time and memory limits by making their program appear more efficient than it actually is. The Linux `time` operations account for time used by subprocesses, so simply spawning one child process using `vfork()` is not sufficient to accomplish this. However, it is possible to bypass this limitation. Our approach is described below, and a diagram is included in Figure 1, Appendix A:

1. The parent process opens a two-way pipe.
2. The parent process spawns a ”first intermediate” child process using `vfork()`. This causes the parent process to become blocked.
3. The first intermediate process spawns a ”second intermediate” child process using the normal `fork()` function, then begins waiting on the read end of the two-way pipe.
4. The second intermediate process spawns the ”worker” child process using `fork()`, then exits.
5. The worker process has its parent PID changed to 1, because its actual parent has died. Its time and memory usage will no longer be attributed to the original process.
6. The worker process spends as much time and uses as much memory as it likes, then writes the output to the write end of the pipe and exits.
7. The first intermediate process becomes unblocked when it receives data on the pipe. The time spent waiting on the pipe does not contribute to either user or system time. The intermediate process then stores the received data in memory which is shared with the parent process as a result of being spawned with `vfork()`. This process then exits.
8. The parent process is allowed to continue, and simply returns the data it finds in memory.

Using this approach, we were able to convince the autograder that our submission met the time and memory restrictions, when, in fact, it ran hundreds of times longer than should have been allowed.

Memory hiding

Even without using the jailbreak, we were able to hide memory by depositing it into files or pipes. It is trivial for a student program to exceed memory limits by creating a temporary file or two-way pipe, then writing its memory into said file/pipe. The program can recover access to this memory simply by reading it back. This allows a student to hide some of their memory usage at the cost of additional system time. We successfully executed this attack using a pipe, and we believe it should work for normal files as well.

Blocking Persistence

The autograder measures user/system time, but not real time. This is sensible, as real time is unstable when the server is under load, whereas user and system time measure the actual amount of time the process is running. However, if a process is blocked, it contributes to neither user nor system time. This means that if a student accidentally or intentionally makes a blocking system call, the autograder will never kill their process, and their submission will become ”stuck”. We were able to achieve this by reading from a pipe that was never written to.

Known Vulnerabilities

The EECS 281 autograder runs a very old version of Linux which has known vulnerabilities (`Linux g281-3.eecs.umich.edu 3.10.0-229.20.1.el7.x86_64 #1 SMP Thu Sep 24 12:23:56 EDT 2015 x86_64 x86_64 x86_64 GNU/Linux`). We were able to cause a memory leak by exploiting CVE-2016-0728[1], and, with additional work, we believe we could gain root access and break out of the jail's `chroot()`.

5.4 Defenses

Short-Term Defenses

Most of the serious vulnerabilities we discovered stemmed from the jailbreak technique we disclosed in section 5.3. These can be trivially fixed by removing `vfork()` from the whitelist of acceptable system calls.

The jailbreak-free memory-hiding technique in section 5.3 stemmed from the ability to hide memory elsewhere. The specific techniques we used can be defended against by disallowing the `pipe()` system call and restricting or preventing the `open()` system call (or simply setting proper file permissions in the `chroot` jail).

To fix the blocking persistence bug, the autograder should kill processes after they exceed a certain amount of real time. Using user/system time is preferable for grading student submissions, but we suggest a maximum real-time cutoff as well. This limit should be several times longer than the user+system time.

Long-Term Defenses

The EECS 281 autograder seeks to properly measure the time and memory usage of student programs, and the jail provides a strong defense against cheating at those goals. However, custom-made security systems frequently have serious vulnerabilities like the ones we described above. We recommend that 281 keep its current jail implementation, but would encourage them to also install commercially-available, open-source security applications, such as Docker, that are specifically designed to contain the execution of untrusted code. We also recommend that the 281 staff make use of built-in Linux security systems, such as process resource limits, and that they install a firewall to prevent student code from connecting to any address other than localhost. Finally, we recommend that the EECS 281 staff keep their autograder's operating system up-to-date, so that students cannot exploit publicly known vulnerabilities.

6 EECS 370

EECS 370 (Introduction to Computer Organization) is a required introductory hardware course for Computer Science students. Typical course projects include assembly programs, processor simulators written in C, and other hardware-related programs.

6.1 Submission Architecture

Submission occurs by way of a publicly-available perl script. This script packages student files, obtains the identity of the current user, and sends the source code to the autograder via unencrypted email.

The acquisition of the session username is of particular interest, as the username included in the email's FROM header represents the one and only means of server-side user identification. Because the submission script runs on the client side, the username may be trivially forged.

6.2 Evaluation Architecture

When the autograder receives an email, the submission is forwarded to a perl script responsible for evaluation. The student code is compiled, and a "student process" is spawned. This student process runs with the same privilege as the autograding script (ability to use various system calls), and a 30 second cpu time limit is enforced to prevent autograder hangups. The student program is fed test cases, and its output is verified against expected output from an instructor-developed solution.

6.3 Exploitation

Impersonation

The submission process for the EECS 370 autograder involves no authentication. The student has complete influence over the submission script, and may trivially alter them to forge an identity of their choosing. We successfully mounted such an attack, impersonating other members of our pen-test team. This exploit could be used to sabotage other students by spending their late days (currency that allows a student to submit an assignment late) or by delivering malicious payloads under their name.

Test Case Exfiltration

As with the EECS 281 autograder, we were able to successfully exfiltrate test cases. The task was trivial in this case, as there are no system call restrictions on the student-code process and no firewalls to prevent a socket from being opened to an external server. Logging defenses present no issue due to the ease of impersonating other users, as described above. The privilege and lax limits placed on the student process could, beyond mere exfiltration, allow a student to damage the autograder in an attempt to delay a project deadline.

6.4 Defenses

Web-Interface Based authentication

Most universities provide a web-based means of authentication, such as that found at `weblogin.umich.edu`, that governs access to a variety of university systems. Such centralized authentication systems are typically maintained by dedicated teams, and tend to be more robust and better-tested than homemade counterparts such as the EECS 370 submission script. Professionally maintained systems such as these increase security by removing the authentication process from the student's sphere of influence.

Policy-based protection

EECS 370 assigns project grades based on each student's *highest scoring* submission. While this does not provide a defense against the attacks described above, it does prevent malicious users from sabotaging a student's grade by submitting broken or low-scoring code under the victim's name near the project deadline.

Proper Sandboxing of Foreign Code

Student submitted code is compiled and executed with an unnecessarily high level of privilege. As stated in our analysis of the EECS 281 autograder, a proper sandbox comprised of firewalls, process limits, and filesystem isolation would reduce the potential destructive capability of untrusted code.

7 EECS 280

EECS 280 (Programming and Introductory Data Structures) is the second programming course students take at the University of Michigan. In it, students write C++ programs on topics such as recursion, object-oriented programming, and dynamic memory management.

7.1 Submission Architecture

Students submit source code files via a Django-based web form. After submitting, students see tests with descriptive names, and whether they passed those tests or not. Not all test results are revealed to students. Some test cases provide additional output regarding whether the student leaked memory, which may affect the final grade. The use of Django in the autograder's web-interface prevents the web developer from writing any SQL code, thus protecting against some forms of injection.

7.2 Evaluation Architecture

When a student submits code, the code is extracted from the input form, compiled, and run with a variety of inputs. The code is run inside a Docker container configured to have no external network stack and an `NPROC` limit of zero, i.e, the code cannot create processes. The container may also be configured to prevent opening and closing of files, depending on the test. Processes exceeding a set limit on wall clock time are terminated with `kill -9`. Output is then checked for correctness outside the container, and results are sent back to the student.

7.3 Exploitation

We discovered only minor vulnerabilities in the EECS 280 autograder. These are discussed below.

Memory Leak Hiding

Some EECS 280 test cases deduct points if they detect a memory leak in student code. This is measured with Valgrind, which reports a leak if a block of allocated memory has not been freed when a program exits. This method can only detect a subset of memory leaks, failing to detect memory that is deleted before the program exits but is retained longer than needed. Therefore, we were able to construct a submission which automatically frees all allocated memory when the program ends, even if the student does not write a main() function and, therefore, does not have control over when the program ends. We believe EECS 280 students have sufficient skill to mount this attack.

Known Vulnerabilities

The EECS 280 autograder uses a fairly new Linux version: `Linux eecs280-VirtualBox 3.19.0-25-generic #26˜14.04.1-Ubuntu SMP Fri Jul 24 21:16:20 UTC 2015 x86_64 x86_64 x86_64 GNU/Linux`. Therefore, it is safe from most known kernel exploits, but it is still susceptible to CVE-2016-0728, with the same ramifications as described in section 5.

7.4 Defenses

Short-Term Defenses

Defending against the memory leak attack requires a different method of measuring leaked memory. Instead of looking for memory which remains leaked at the end of the program, the autograder should enforce the following rule: *while the number of objects allocated by the autograder remains bounded, the amount of memory used by those objects must also remain bounded.* Consider how this rule could be applied if the student's assignment involved creating a class which allocates memory in its constructor and deletes it in its destructor. The autograder can repeatedly create and destroy instances of that class. If the memory in use by the program continues to increase over time, then the autograder has detected a memory leak. We recommend implementing this detection method, but we do not recommend removing the Valgrind-based leak check, as it provides easily debugged feedback to students.

To defend against the kernel-based privilege escalation, we recommend that the autograder operating system be frequently updated, at least with regards to security-sensitive patches.

Long-Term Defenses

Because the EECS 280 autograder already implements the defense mechanisms we recommend for secure autoograder implementation, and because we found no serious vulnerabilities, we have no recommendations for long-term changes to the this system. Instead, we simply recommend that the autograder infrastructure be kept up-to-date and security patches be applied in a timely fashion.

8 EECS 485

EECS 485 (Web Databases and Information Systems) is a web development class in which several early projects require students to build a photo sharing website. The projects integrate a Python Flask/MySQL back end with a front end that uses Jinja templates and JavaScript.

8.1 Architecture

EECS 485 uses the same autograder software as EECS 280. Therefore, it has the same exceptional level of security against malicious student code. However, EECS 485 has a very different submission and evaluation architecture. In EECS 280, students submit files to be compiled and run on a separate server. In EECS 485, however, students run public facing websites on an assigned port of a university-provided server, which hosts several student groups. In order to prevent theft or sharing of code, students only have read and write permissions in their assigned directory on the server.

When a student wishes for their project to be evaluated, they submit a request to the autograder, rather than submitting their code to be run on an external server as in the previously discussed models. The autograder then issues a series of requests corresponding to hidden test cases, evaluates the responses provided by the student site, and returns a summary of which tests the submission passed/failed. This project structure provided an additional attack surface not present in the EECS 280 model and allowed us to successfully mount a variety of attacks ranging from exfiltration of test case details to stealing code written by other students.

8.2 Exploitation

Exposing Student Code

EECS 485 students run their servers on remote machines, with several groups assigned to each machine. While students do not have read access to other groups' directories, the hosted websites are public facing, so anyone with the URL could easily navigate to another student's site and access their HTML and JavaScript files. Because the URLs are fixed for each project, it would be a simple matter for a student to scan the server for active ports and access another group's site. To prevent this, each student group is provided with a "secret key" comprising 20 hexadecimal characters that must be included in the URL for each page on their site. The key space is 2^{80}, which makes a brute force attack infeasible, even if the port number is known.

While URLs that point to dynamically generated content are protected by this "secret key" system, static files, such as images and, more importantly, JavaScript files, are served from a directory whose file path does not contain this secret string. Therefore, a student can mount a fairly simple attack by running a port scan on the server to determine which ports are open, then searching under /static/js/ for common file names such as main.js or app.js. Additionally, if a student site uses the base HTML template provided by the course staff to serve pages for invalid URLs, the static files will be served on the 404 page, removing the burden of guessing the JavaScript filename. Furthermore, these JavaScript files frequently contain asynchronous calls to controller methods whose URLs contain the secret string, allowing an attacker to gain full access to the front end of another group's site.

In order to test the validity of this attack without running a port scan on the EECS 485 server, one member of our team set up a personal server and hosted a dummy website on a randomly selected port, using a file structure analogous to that used in an EECS 485 project. A second team member, given only the domain name of the site, was able to determine the port number, access the JavaScript, and acquire the secret string.

Port Squatting

The autograder assigns each group a particular port to use for running their server. Because multiple groups share the same server, port squatting becomes a serious issue. Because a process cannot bind to a port which has already been bound, any student with access to the server may block another group's submission by binding their server to that group's port. This is a particularly concerning issue, since students could leverage this to prevent other groups from submitting. Furthermore, it would be difficult to prove malicious intent, as the student blocking the port could claim that they had done so accidentally. Additionally, if one member of a group runs a server on their group's port, the other students in the group will be unable to submit until that server is shut down.

8.3 Request Logging

EECS 485 is unique among the autograders we tested in that students control the directory in which their code executes during the grading process. Therefore, a student could easily log requests made by the autograder to a local file during grading. This represents a partial leak of test cases, as a student is able to record requests sent to their server by the autograder, although never the correct behavior that the autograder is checking for.

We can see no simple way to definitively protect against this attack, other than running student code directly on the autograder machine. The scope of this vulnerability could be limited by not running certain tests until after the submission deadline, but this both inconveniences students by obscuring their full grade, and provides only a short-term solution, as a student could record the test cases to distribute to students who take the course in subsequent terms.

8.4 Defenses

The exploit that allowed students to circumvent the secret key system by examining static files can be remedied by requiring students to serve their static content from paths that include the secret string. Flask's "send from directory" function provides one simple way to do this.

There are a few ways to solve the intra-group port squatting problem. One solution is to assign port numbers to each student, rather than to each group; another is to provide only a single user account per group, rather than independent user accounts per student, so that one student can kill the server of another student in the group and reclaim their port.

Solving the inter-group problem is somewhat harder. Linux provides no built-in way to restrict which ports a particular user can open (beyond the well-known-ports restriction). One solution is to run each group's code in a Docker container (or VM) and to publish only the appropriate port on the container.

There is, however, a solution that solves both problems. Rather than assigning ports to each group, let a student pick any port and specify their chosen port upon submission. This would prevent any other users from squatting on their port, as they could simply choose another. We originally worried this would allow a malicious student to submit the port for another student's site, thereby claiming credit for the victim's work. However, the secret strings described previously would prevent this. This change requires minimal modification to the existing autograder; therefore, this is the solution we recommend.

9 Discussion and Conclusions

This section will serve primarily as a resource for instructors wishing to implement autograders for their own courses. We observed serious vulnerabilities in three of the four autograders we surveyed. The one without serious vulnerabilities, EECS 280, was written from the ground up with security as a primary concern and implements modern defenses. Thus, we will use it as our model of a strong, well-sandboxed autograder.

9.1 Defense in Depth

"Defense in Depth" is a classic military idea promoting the benefits of multi-layer defense as opposed to one-layer defense. Additional layers of security provide redundancy, such that the failure of one layer doesn't compromise the entire system. We were able to bypass a single layer of defense on many of the autograders; our attacks may have been thwarted with the presence of multiple layers.

The jail implemented in the EECS 281 autograder is certainly a useful system, but once we broke free of the jail, we were able to execute arbitrary code with the same permissions as the autograder process. Had the autograder implemented a firewall in addition to the jail, we would have been unable to exfiltrate test cases. Had Linux process limits been used, we would have been unable to fork processes, which would have prevented us from breaking the jail in the first place. Each of these security elements provide valuable defenses, but implementing them all together minimizes the damage that can be done in the event that one component fails. We strongly recommend that autograders rely not just on their own custom security systems, but on firewalls, containerization, and well-configured process limits in combination. Our inability to break the EECS 280 autograder, which implements all these defenses, is a testament to the effectiveness of the "defense in depth" strategy.

9.2 Professional / Open-Source Defenses

Problems frequently occur when individuals attempt to create their own security software. This does not imply that these developers are incompetent. Rather, it speaks to the difficulty of implementing sound security measures with limited resources, a situation not uncommon for course staffs developing autograders. Their efforts are typically split between development, teaching, and research, and they simply do not have the time to perform a comprehensive security analysis of their autograding systems.

The value of having many eyes examine a system (in the case of open source) or a dedicated team to develop and review security software (in the case of a professional product) cannot be overstated, as this increases the odds of successfully detecting vulnerabilities and results in hardened and well-tested systems. This paper serves as an example of how even competently designed custom systems, such as the EECS 281 autograder, can contain subtle but serious vulnerabilities, while EECS 280, which relied on thoroughly tested defenses such as Docker, firewalls, and built-in Linux security measures, resisted all our exploitation efforts. Thus, we strongly recommend that autograder developers choose to use open source or professionally developed security frameworks and defenses, rather than implementing custom solutions.

9.3 Detection

A distinguishing characteristic of the autograder model is that submissions are tied to a student's identity. If a student wishes to cheat, they need not only exploit flaws in the autograder, but also do so without being detected. Logging and notification mechanisms, then, are invaluable security tools. The jail in the EECS 281 system, for example, logs any use of system calls which it considers particularly illicit. While an attack that managed to escape the jail would have gone unnoticed, the logging system would have detected any failed exploitation attempts leading up to the successful attack. Logs are only useful if someone reads them; therefore, systems should notify administrators in the event of notable violations.

The benefits of effective detection are predicated on the truth of our assumption that autograder submissions are tied to the identity of the submitting student. Two of the autograders we surveyed, EECS 485 and EECS 370, failed in this regard. EECS 485 allowed students to access their classmates' code over the internet, rendering it difficult or impossible to identify the culprit, while EECS 370 allowed students to tie arbitrary submissions to the names of other students. This vulnerability is particularly serious, as it not only allows students to get away with cheating or bringing down the autograder, but also allows them to blame the attack on someone else, which may endanger the victim's career or academic standing. Because detection can significantly strengthen security and because incorrect attribution of submissions can cause serious harm, we conclude that accurate identification of students is the single most important defensive measure an autograder can have.

10 Future Work

While our examination of the four department autograders has both increased the security of courses at the University of Michigan and provided valuable insights into the topic of autograder design and implementation, four systems comprise a small sample size. Ideally, future work would study a larger number of systems from several different universities. A more diverse set of subjects would provide a better picture of the state of the art and would likely yield more widely applicable conclusions regarding autograder security.

11 Acknowledgments

We would like to thank the staffs of EECS 280, EECS 281, EECS 370, and EECS 485 for providing access to their autograding systems. In particular, we would like to thank James Perretta, Waleed Khan, and Don Winsor for their considerable assistance in setting up and using their autograder systems.

The support of our professor, Alex Halderman, allowed this project to get off the ground, and his weekly security lectures provided us the proper skills and mindset to succeed in our penetration testing.

12 Availability

Code for the attacks described in this document is available at:

`https://ianpudney.com/autograder-audit`

Source code for the autograders and exfiltrated test cases will not be made public, per section 4.

References

[1] Analysis and exploitation of a linux kernel vulnerability (cve-2016-0728). http://perception-point.io/2016/01/14/analysis-and-exploitation-of-a-linux-kernel-vulnerability-cve-2016-0728. Accessed: 2016-04-19.

[2] Danutama, K., and Liem, I. Scalable autograder and {LMS} integration. *Procedia Technology 11* (2013), 388 – 395. 4th International Conference on Electrical Engineering and Informatics, {ICEEI} 2013.

[3] Li, Y., McCune, J., Newsome, J., Perrig, A., Baker, B., and Drewry, W. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 409–420.

[4] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 79–93.

Appendix A Diagrams

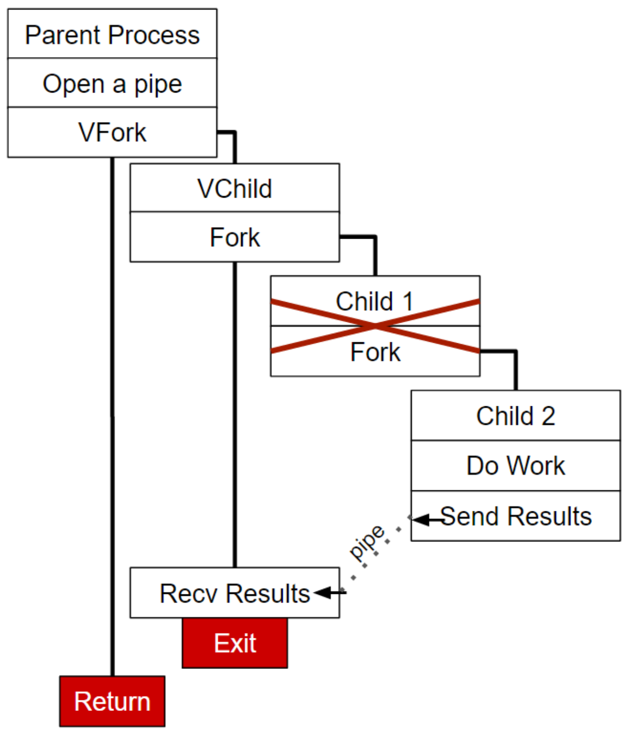

Figure 1: A diagram showing the process of bypassing EECS 281 performance limitations, as described in section 5.3