Secure Execution of Student Code

Ryan W Sims

May 15, 2012

1 Introduction

To grade projects in a large computer science course requires verifying the correctness of a great many student programs, all against identical inputs and outputs. This situation is an obvious candidate for automation, and calls for a system to which students can remotely upload their project implementations, and which will automatically check those implementations against tests uploaded previously by the instructor. Automated testing is a staple of the industry, and it seems logical to extend this practice to academia. However, in a recent informal survey[1] of 200 computer science faculty and students, only 25% reported using a submission system tailored specifically to computer science project, and of those less than 50% used systems that automatically tested student code. Many respondents complained in the survey of the time it takes to assess student projects, so while it seems that this is an area ready for automation, there is clearly some barrier to adoption that is preventing its widespread use.

While not explicitly addressed in the survey, one concern with such automated submission systems is that they need to execute student code, which is very likely buggy and potentially even malicious. To address that concern, in this paper we investigate the security solutions applicable to automated submission systems, and survey the current practices in some notable examples of such systems.

Many of these systems are are designed primarily to be used within the confines of an institution, where their users are easily auditable and the possibility of administrative reprisal presents a high cost for malicious behavior. However, the trend is strongly moving in the direction of opening these systems to the wider, anonymous Internet, where nearly anyone can access the systems and upload arbitrary code. In that case, these security concerns become paramount, because openly-accessible Web applications have a much harder time auditing their users, and they have little recourse in the case of bad actors.

Every automated testing system reviewed for this paper uses a Web application as a significant part of their interface, meaning that each needs to handle the usual Web security concerns of cross-site scripting and request forgery, distributed denial of service attacks, code injection, etc. Techniques for protecting

[1] Results available at `http://marmoset.cs.umd.edu/surveys/SurveySummary_05072012.pdf`

against these kinds of attacks will not be addressed here. However, each of the systems reviewed also automatically executes untrusted student code, which presents a different set of security challenges. These specific challenges are the focus of this paper.

Web security aside, there are three basic categories of vulnerability that are most relevant to automated submission systems. In order of increasing difficulty of attack implementation, they are denial of service, corruption of the test harness, and leaking of test data.

1.1 Denial of service

One of the most obvious ways of attacking a test harness through submitted code is denial of service (DOS) using a fork bomb, infinite loop, memory leak or excessive output. This particular attack is interesting in that it is also the most likely to be observed even in non-malicious code, since inexpert programmers may inadvertently code an infinite loop. This attack then should be carefully addressed even in relatively situations where actual malicious intent is unlikely.

It is also important to note that merely compiling untrusted code provides a vector for this attack. Both `javac` and `gcc` have had problems in the past [1, 2] where legal source code could send the compiler into an infinite loop. Therefore denial-of-service mitigation techniques should be applied as diligently to the compilation phase as they are to the execution and testing phases.

1.2 Corruption of test harness

The test harness must be able to run the same test consistently across different submissions or repeatedly on the same submission. This implies that test execution must be idempotent with respect to the state of the machine and the test harness must be able to reliably roll back any state changes that occur during testing, or guarantee that tests cannot change state. If it is necessary that student code have sufficient privileges to write files, for example, the test harness must then be able to reliably delete all files created during testing, even those that are not directly evaluated as part of the test, otherwise, if state can leak between test runs, it is possible for a submission to influence subsequent tests, possibly those of other students' code.

Besides machine state such as file systems, network connections, databases, etc., the test harness itself represents part of the state of a test execution. For example, a common way of evaluating student code is to compare program output against a collection of canonical output files stored in the test harness. The testing process obviously must have access to the canonical outputs in order to compare them with the test output, but it must also ensure that they cannot be modified in any way by the tests themselves. More generally, the execution of any given test will require access to a great deal of test harness state: the submission source,

test requirements, database connections, etc. This state must be able to influence the test, but that influence must be only one-way.

Note that ensuring a clean environment for every test conflicts somewhat with the need to audit student activity within the build system. For example, if student code downloads an executable or library that contains the code for the attack, the test harness will remove it after execution of the test, removing the evidence of the malicious code. Therefore, any cleanup performed by the test harness must maintain the auditability of the system as much as possible.

1.3 Leaking test data

The term "test data" refers to two different kinds of data: protected data such as database passwords or the list of canonical test outputs, which should never be accessible to student code, and unprotected data like test inputs, which *must* be accessible to student code. There are many straightforward methods for limiting access to protected data, but unprotected data presents a problem, since it must be read by student code during a test, but must not escape the scope of that test. For example, in the case of testing against canonical outputs, the test processes must at some point have read access to the files containing the test case inputs, but it is desirable that students remain ignorant of their contents. The test harness therefore must prevent these canonical outputs from leaking back to the author of the code. Assuming student accounts do not have access to the build servers, satisfying this requirement becomes a question of preventing test inputs from being sent out over the network.

To summarize, a secure test harness must do four things:

1. Strictly limit access to system resources, most especially CPU time.
2. Prevent student code from accessing protected test harness data, such as database passwords or canonical test outputs.
3. Ensure that test runs are idempotent (we refer to this as "test hygiene") by preventing modification of test harness state.
4. Prevent even unprotected test data (e.g. test inputs) from escaping the scope of the test itself.

It is worth reiterating that, because of institutional controls external to the submission system, taking extraordinary measures to prevent against the more exotic of these attacks might not be warranted. In such situations, those efforts would be more usefully applied to increasing system auditability so that institutional

reactions can be appropriately directed. However, if the system is going to be accessible to users outside the institution, then all three of these concerns must be addressed in detail.

2 Related Work

Luck and Joy [6] enumerate a list of five requirements for automated submission systems, but the only security-related requirement is that student code be submitted to a location that is only accessible to the instructor; no mention is made of sandboxing or secure execution. Later in their paper, they describe a security framework which uses user-level restrictions for code isolation. Untch et al. [7] describe a system that uses separate processes to ensure student accounts do not have access to test harness data, but they do not address the issue of malicious student code. In a 2010 review of automatic submission systems, Ihantola et al. [5] discovered that "a large portion" of the systems they surveyed omitted any discussion on secure sandboxing for student code; in addition to the methods described in our paper, they suggest static analysis and client-side testing as possible security solutions. We rejected client-side testing as a security solution, since it necessarily leaks test data to the client. Static analysis could potentially identify many security risks such as infinite loops and fork bombs, but this essentially boils down to a blacklist of forbidden semantics, and blacklists are notoriously dangerous from a security standpoint. Forišek [4] identifies, in the context of programming contests, a list of possible attacks on an automated testing system (similar to those laid out above) and general suggestions for preventing each.

3 Security Models

In this section, we introduce three broad security models for addressing the vulnerabilities explained above: user-level controls, process-level controls and virtual machines. Each model has its own advantages and drawbacks, depending on the infrastructure and administrative support available.

Regardless of which method is preferred, compilation and execution of student code should be run on a machine that is physically distinct from that which handles user interaction (e.g. the Web server). This separation requires a minimum of implementation effort, and has many benefits:

Isolation The physical separation of the servers automatically isolates untrusted code, regardless of other security measures.

Scalability Experience with the Marmoset [**?**] system has shown that build server load can be extremely high right before a major deadline, or when a test setup is found to be faulty and hundreds of submissions need to be retested. This load should not be shared by the user-facing server.

Disposability Updating user-facing software needs to be done with care to prevent productivity loss due to regressions and update problems. If build servers are separate from the interface, they can be upgraded piecemeal at any time, with minimal degradation of user experience.

Anonymity Students cannot interact directly with separate build servers, which makes it difficult or impossible to discover their location on the network, providing a barrier to malicious activity.

Henceforth, we assume that that the two servers are physically separate as described, and we will adopt the term "submit server" to refer to the server that handles user interaction (accepting submissions, presenting test results, etc.) and "build server" to refer to the server that compiles, executes and tests untrusted student code.

In the discussion that follows, each security model is described in terms of two different permission levels, named TEST and HARNESS. The HARNESS level comprises the privileges that is needed for the general operation of the test harness, which includes access to protected data and unlimited usage of system resources[2]. The TEST level is a much more restricted set of permissions, limited to those required for compiling and executing untrusted student code. Because the HARNESS permissions are so broad, no untrusted code can ever be allowed to execute with HARNESS-level permissions.

Note that this paper deals only with Unix-type operating systems, specifically Linux and BSD variants. OS X systems fall roughly under the category of "BSD variants," but are not dealt with specifically. Windows provides a type of mandatory access control (see section 3.2.2) through system calls, but secure execution on Windows is best ensured through virtual machines, as in section 3.3.

3.1 User-level restrictions

The most obvious way to model the two permission levels is with the usual Unix user and group access controls. In this case, TEST and HARNESS are two separate user accounts on the build server, where HARNESS will run all processes that need to administer the test harness itself, and hand over control to the minimally-privileged TEST user for the compilation and execution of student code. TEST and HARNESS will also be the names of user groups, where the TEST is a member of the TEST group only, and HARNESS is a member of both the HARNESS and the TEST groups; this group arrangement models the fact that HARNESS's privileges are a strict superset of TEST's.

[2]The HARNESS level is privileged with respect to test harness data and resources, but it is neither necessary nor desirable that HARNESS privileges be equated with superuser privileges.

3.1.1 Filesystem permissions

Given the two users and groups, standard Unix filesystem permissions can address the access control and test hygiene requirements. Access control couldn't be easier: HARNESS can restrict TEST's access to protected data simply by putting it in a directory that is readable only to users in the HARNESS group. Then, when a student submission needs to be tested, HARNESS can create a directory accessible to the TEST group, and put into it only the unprotected data necessary for the execution of that single test. Note that this working directory should not contain the files with canonical outputs, since otherwise the code under test could could simply echo their contents as its own output. Instead, TEST should execute the code and store the output to files, where HARNESS can check them for correctness. After checking the produced output files against the canonical outputs, HARNESS deletes the working directory (since HARNESS created in the first place, it is owned by HARNESS), which will ensure test hygiene[3]. As added protection, the so-called "sticky" bit (01000 in octal) can be set on the test working directories. When applied to directories, the sticky bit prevents the deletion or renaming of files by any user other than the file's owner. This allows HARNESS to ensure that the initial file set for a test cannot be altered by deletion or renaming; if desired they can also be made read-only by setting the appropriate permission bits.

3.1.2 Resource limitation

While filesystem controls address the questions of limiting access and test hygiene, they cannot prevent DOS attacks, or prevent unprotected test data from leaking. These two requirements must be addressed by limiting TEST's access to system resources. "System resources" can refer to a great many things, from printers to displays, but there are essentially four that concern automated testing: CPU time, system memory, the filesystem, and network connections. Limiting TEST's consumption of these resources will address the final two requirements for a secure build server.

Any of these four resources could be involved in a DOS attack, which can be attempted in many different ways: code can silently loop forever, write huge amounts of data to the filesystem, etc. These attacks are all predicated on abusing access to a system resource in such a way as to consume all of it, denying its availability to other processes. Each of the four resources listed above can be subject to DOS attacks, additionally, network connections can be used to attempt DOS attacks on other systems. Clearly a way must be found to limit the maximum amount of a resource that TEST may consume[4]. Unix systems provide some resource limitation through the POSIX `setrlimit` system call, specifically it can limit the maximum

[3] It may help auditability to snapshot a list of all the files in each directory after each test run to ensure that no unexpected files are being created, such as downloaded executables.

[4] Again, HARNESS is assumed to have unfettered access to system resources, and so may not be allowed to execute or compile untrusted code.

amount of CPU time or resident memory[5] a process may have, and it can limit the number of simultaneous processes a user may have. These limits apply to all processes created by a user. The call actually sets two limits, a "hard" limit and a "soft" limit; only superusers can increase the hard limit, user process may only lower their hard limit or set their soft limit to any value between 0 and the hard limit. Any process that violates its soft limit is terminated by the kernel. These limits can be set permanently in a `limits.conf` file, which is read at startup.

Besides looping or forking indefinitely, a process can also attempt a DOS attack by outputting an infinite amount of data. In this case, both the number of files created and the total amount of data written must be limited, since there is not only a finite amount of disk space but also a finite amount of space in the kernel's inode table, which can be overwhelmed by creating many small files. Excessive output to the terminal can be dealt with in the same way by ensuring that TEST's processes can only write to files, not standard output or standard error; this also helps with auditability by keeping a log of all the output from student code. Both inode count and disk space can be limited through the use of filesystem quotas. The details of setting up quotas are platform dependent, but all modern Unix systems provide for per-user limitations on inodes and total disk space.

Besides filesystem, memory and CPU limitations, a way must be found to control TEST's access to the build server's network. If TEST has unrestricted network access, it might be possible to use build servers as part of a distributed denial of service (DDOS) attack on remote hosts, or to leak test inputs across the network. Obviously the build servers themselves must have some network access, if only to query the submit server for submissions that need to be tested. Except in the case of network-related projects, however, TEST must not have any network access whatsoever. Part of this limitation is simply good security practice for any server: build servers should accept connections on only those ports necessary for the operation of the server, and those ports should be bound to active system processes at all times. Build servers must also prevent TEST from sending information out of the machine. Facilities for accomplishing this vary widely from system to system, so implementations will need to be tailored to the specific operating system on which they are intended to run. On Linux, the kernel firewall "iptables" can filter outgoing packets by originating user name, dropping any sent by TEST. On BSD systems (including OS X Lion and later), the "PF" packet filter works a similar way; PF differs from iptables in that its stateful nature can also filter *incoming* packets by the user or group who owns the listening socket. In general, restricting the allowable ports and preventing TEST from sending information out should suffice to prevent inappropriate use of the network.

While many projects may function without any network access at all, this policy must permit relaxation

[5]Note that the limit on memory usage may not be enforced immediately: the documentation for `setrlimit` says only that "if memory is tight, the system will prefer to take memory from processes that are exceeding their declared resident set size."

for networking-oriented projects. Since both BSD and Linux allow packet filtering by group or user, a simple solution is to have two unprivileged users, perhaps TEST and NETWORK_TEST, where the latter is accorded limited network access. The extent of access permitted should be dependent on the project, and should be limited as much as possible, since this necessarily opens the build server to abuse. Automatically-tested, network-enabled projects may not be appropriate for submission systems where the student base comprises the wider Internet.

User-level restrictions are the most basic available on any Unix system, and they can provide all the isolation and access guarantees needed for an automated test execution framework. This combination makes them an ideal mechanism for protecting build servers that need to execute untrusted code. Their simplicity makes them easy for system administrators to implement, but does not sacrifice protective power. However, privilege escalation attacks can completely subvert all the controls listed above, so user-level restrictions do rely on the underlying security of the programs involved in test execution. This is somewhat troublesome, because executing even simple tests may involve the coordination of a great many programs, each of which represents a potential attack vector. Additionally, since user-level restrictions apply to every action taken by a particular user relaxing restrictions generally requires the creation of additional users or groups. If more granular access controls or fewer attack points are desired, it is necessary to investigate different security paradigms.

3.2 Process-level restrictions

Modeling permissions based on individual processes is the most granular of the three paradigms, since they allow a single user can transition through several subtly different permission sets as the they execute different processes. Unfortunately, process-level permissions are not a standardized Unix feature (though they are built into most modern Unix implementations), so their configuration is platform-dependent and often quite complicated. There are two broad categories of process-level restrictions: process "jails," which restrict a process's view of the environment, and mandatory access control, which specifies detailed rules about how processes can interact with objects in the environment.

3.2.1 Process jails

A "jail" refers to a restricted process environment that is a subset of the normal process environment. Under this model, the TEST permission level is modeled not by restricting access to protected data, but rather by excluding the data from TEST's environment entirely. A jail environment seems to be complete, in that it has a filesystem root with the usual configuration files and utilities, but it is an artificial environment contained

within a larger, real environment. In this sense, it is similar to the virtual machine model (see below), but a process jail involves no virtualization of the hardware or kernel.

Jails are normally created by populating a directory tree somewhere in the filesystem that mirrors the usual Unix directory tree, with `usr/bin`, `etc` directories, and so on. These directories contain only the tools that TEST would require for test compilation and execution, and then the superuser would initiate the test daemon using a special command or system call that confines the process to the jail. From the jailed process's perspective, the jail environment appears[6] to be a normal environment, and all its filesystem accesses are restricted to the jail. Additionally, any process forked by a jailed process inherits its jail, so untrusted code cannot escape by creating new processes.

To create a jail, both Linux and BSD provide the `chroot` system call and commandline utility, which set the filesystem root of a process. The system call is restricted to the superuser, but the `chroot` utility has arguments for setting the user and group of the process after the filesystem root has been changed. It is important to note that there are ways for superuser processes to escape the jail [3], so no such process should ever be run inside the jail, and permissions should be dropped to an unprivileged user as soon as possible.

The `chroot` call provides significant isolation for TEST's processes, but it only restricts TEST's view of the filesystem. Resource limitations will need to be imposed as above to prevent DOS attacks or leaking test data across the network. To address this, FreeBSD (in addition to `chroot`) provides the more powerful `jail` system call. It creates a restricted filesystem view just as `chroot` does, but it also allows jails to be restricted in other ways. For example, the `jail` system call specifies the list of Internet addresses available to the process, or disable networking within the jail entirely. This is a much simpler way of restricting network access than using kernel-level packet filtering, as above. Additionally, jails all have a *jid*, or jail ID, which can be used to refer to jails for external control and configuration, so that users with the appropriate privileges can execute commands within the jail, kill a jail's processes, etc. The `jail` call provides many of the benefits of a fully-virtualized guest operating system, but without the need for virtualization overhead. However, it is currently limited to FreeBSD, so other systems will need to implement other kinds of process-level controls to achieve the same results.

In addition to their security benefits, jails provide a useful way to compile and execute student code in a different build environment than that running on the normal system, which can be useful if certain programming projects require access to, for example, a different version of the C standard library than that used on the build server itself. This is an easy way to control the toolchain available to student code without interfering with the build server's normal operation.

[6] It is possible for a process to discover it is running inside a filesystem jail by examining known inodes, but it should not be possible for a non-superuser process to escape.

3.2.2 Mandatory Access Control

An alternative process control mechanism is known as mandatory access control (MAC). Mandatory access controls differ from the usual Unix permissions (known as *discretionary* access controls) in that the permissions, once set by the superuser, are enforced by the kernel and cannot be changed at the discretion of the user. MAC frameworks also tend to be much more granular: they provide more types of permissions (user, domain and role versus user and group) and the permissions apply to more interactions than just read, write and execute. Under MAC, the two levels of permissions are modeled by a detailed matrix of rules specifying the access permissions for every process involved in the entire test harness. In that sense, the two permissions levels break down in to many more nuanced permission levels, tailored to the individual task of a process. MAC has been implemented by the SELinux and TrustedBSD projects[7].

A MAC framework consists of a list of rules that specify how subjects (processes) can interact with objects (files or other processes), and how (or if) a subject in a given domain is allowed to transition to another domain. This idea of allowing a process to transition between domains is a key feature of MAC, since it allows processes to seamlessly and securely acquire or relinquish permissions as the testing process goes on. To configure a MAC system, the superuser creates this detailed set of rules and puts the system in "permissive" mode, where illegal access are logged but not prevented; these logging messages help diagnose what permissions need to be added to the system to allow normal operation. Once the rules are finalized, the system is put in "enforce" mode, where access violations are both logged and prevented. From then on, *any* change to the rules requires superuser permissions and, generally, a reboot of the machine.

Mandatory access control systems are intended for hardened security applications, in fact, SELinux first existed as a series of patches to the Linux kernel contributed by the NSA. They are based on intense theoretical security research, and are targeted at experienced and knowledgeable system administrators. No other security measure described in this paper matches them in thoroughness, but the difficulty of administering such a system must be weighed against the potential threats that the system must counter.

3.3 Virtual Machines

At the extreme end of the isolation spectrum are virtual machines. These are similar in many respects to the process jails discussed above, but they go further in that the operating system itself, or even the hardware of the machine, is emulated, rather than just the filesystem. In their fullest expression, there is theoretically no way for any process at any privilege level to determine that it is running in a virtual environment; in reality of course virtual machines are as vulnerable to security bugs as any other software system. However,

[7]The TrustedBSD MAC framework has been implemented in OS X since Leopard (it is known as Seatbelt), but as of this writing the documentation is incomplete.

they do limit potential attack vectors, since any access to the outside host must first compromise the virtual machine.

For the purposes of this discussion, we will consider two different types of virtual machines: the Java Virtual Machine (JVM) and a general class of hosted operating systems.

3.3.1 The Java Virtual Machine

The Java runtime provides a virtual stack machine that executes Java bytecode, constraining Java processes inside a limited virtual environment. It is not a fully virtualized guest, since Java programs access the host filesystem, display, etc. However, the added layer of indirection allows for fine grained and effective security controls using an instance of the `SecurityManager` class. Under the Java model, the permission levels relate to trusted and untrusted code, where trusted code is given HARNESS-level permissions, even if called from untrusted, TEST-level code.The permissions that can be checked range from creating sockets to audio playback. The Java runtime has a set of default system permissions, but per-user permissions can be set by policy files in the user's home directory.

When code running under a `SecurityManager` attempts an action that requires some permission, every code block on the call stack is checked for the proper permissions. If some block is found that does not have the proper permissions, an exception is thrown and the action is denied. However, if the stack includes a `PrivilegedAction` block, and that block was run by a class with the proper permissions, the action is permitted, regardless of the permissions of other code on the stack. This allows trusted code to "take responsibility," in a sense, for privileged actions on behalf of untrusted code. This makes the Java model the most flexible of all the models discussed in this paper, since only it can specify permissions per-library, even within a single process. Trusted code sources are identified in policy files, either system-wide or per-user. They can also be configured programmatically at run time.

One potential downside to this technique is that it obviously requires the untrusted code to be executing on the JVM. However, that restriction only means that student code must be compiled into Java bytecode, and Java interpreters or compilers are available for a wide array of languages in many different paradigms. A larger concern is that of compatibility. If some library used by the test harness (e.g. a code coverage library) is not written to use `PrivilegedAction` blocks as described above, then it is relying on executing privileged code without a `SecurityManager` in place, and may well fail when one is enforced. Additionally, interpreters for non-Java JVM languages may not take the security framework into account, and they might similarly fail. While Java's `SecurityManager` provides an excellent way to control test execution, it would be important to do sample testing runs before committing, to make sure all the necessary libraries and interpreters work as expected.

The Java security framework provides a simple, cross-platform and granular way to restrict untrusted code, and it requires no superuser permissions to configure. Certainly Java-based projects should consider it, even in addition to the other methods described here. It is also important to note that the Java security framework does not suffice to protect against DOS attacks through infinite loops, nor does it provide protection during compilation. It must therefore be part of a system that involves other protections discussed in this paper, rather than the sole security mechanism.

3.3.2 Hosted Operating Systems

The term "virtual machine" usually refers to these, where an entire operating system (or the entire machine itself) is hosted as a guest application. Examples of hosted operating systems include systems such as Parallels, VMWare, VirtualBox, etc. Similarly to process jails, the virtual machine model implements the permission hierarchy by excluding protected data from the unprivileged environment entirely. This has become popular, to the extent that most modern processors include hardware support for virtualization. Virtual machines in this sense provide an extremely attractive solution for isolating untrusted code, since even if the code manages to compromise the guest system to the extent of gaining root privileges, as long as the virtual machine environment is secure, it cannot attack this host system. This is of course an ideal situation, and there are documented vulnerabilities in virtual machine software. There is also the drawback that virtualization comes with a certain performance overhead, which hardware support alleviates but does not remove.

Setting up a virtual machine is similar to a process jail (see above) in that the administrator must choose precisely what programs and configuration should comprise the guest environment. Additionally, the virtual machine must be granted a subset of the system's resources, such as disk space, memory, networking, etc.; this implicitly provides the needed resource limitations. This is significantly more difficult than setting up a process jail, since it is essentially configuring an entire operating system, which then must be captured in an image that can be mounted by the virtual machine software. This process varies greatly from application to application, and will not be discussed here.

Once the virtual machine is running, there must be some way for TEST (inside the guest) and HARNESS (running on the host) to communicate, since HARNESS will be responsible for fetching work (i.e. source code to compile and test) from the network and handing it off to TEST for compilation and execution. There are two main ways of doing this: either TEST and HARNESS can communicate via the virtual network, or they can share a directory on this host, mounted on the guest through some application-specific means. The most important property that this shared folder must have is that no user (not even the superuser) in the guest environment can use it to navigate above it in the host's filesystem.

Regardless of how TEST and HARNESS communicate between guest and host, there must be a way of starting TEST's processes from outside the virtual machine. A simple way to do this is to have TEST run as a daemon in the virtual machine, which is started on boot. Alternatively, many virutal machine applications allow a user outside the machine to initiate commands that will run inside the machine, allowing HARNESS to activate the testing process remotely. This external control exemplifies what is perhaps the biggest advantage of a virtual machine: disposability. One of the arguments for isolating the build servers from the submit server was that it allows servers to be taken down, updated and restarted at any time, without regard for it adversely affecting the user-facing systems. Confining test execution within virtual machines on the build servers themselves has a similar advantage. Upgrading the test environment is as simple as deploying a new virtual machine image to a build server, and dealing with a misbehaving test run is as simple as terminating and restarting a virtual machine. In the extreme, test environment hygiene can be ensured by restarting a testing virtual machine from a clean image for every single test run, although that would incur a certain performance overhead that may not be desirable. A compromise could be made between performance and hygiene by simply restarting test environments on some arbitrary schedule; this would also ensure that tests run in the most up-to-date virtual machine image on the system.

3.4 Summary

User-level, process-level and virtual-machine controls all provide unique ways to isolate the compilation and execution of untrusted code. Some approaches, like build server isolation, have distinct advantages regardless of the wider environment of the system being secured, but the extent of the security measures required depends greatly on the intended audience of the submission system and the possibility of institutional reprisal given abuse of the system. While no discussion of security practices could hope to be entirely complete, the models discussed above represent the three most straightforward methods for securely compiling and executing untrusted student code. Table 1 provides a breakdown of how each model handles the different requirements presented in the introduction.

3.5 Recommendations

Each model discussed above has its own strengths and weaknesses, and it may seem difficult to choose among them. However, there are strong reasons to prefer some models over others, so here we present our recommendations based on the research above.

User permissions. The feature breakdown in table 1 raises the question of why anyone would choose any model other than user permissions, given that it can address all the security concerns laid out above, and

	CPU	Processes	Memory	Filesystem	Network	Access Control	Test Hygiene	Superuser Required
	Resource Limitation					Misc.		
User permissions	✓	✓	✓	✓	✓	✓	✓	✓
Process jails	×	×	×	×	?	✓	×	✓
MAC	?	?	?	?	?	✓	×	✓
Java VM	×	✓	✓	✓	✓	✓	×	×
Guest OS	✓	✓	✓	✓	✓	✓	✓	×

Table 1: Feature breakdown of the various models. A ✓ means that the model supports the restriction, ? means that it is implementation-dependent and × means that techniques from other models must be employed.

it has the simplest configuration of all the models discussed. The user permissions model unfortunately has a serious flaw: the code base that must be trusted is large and heterogenous. Every single program used during the execution of a test must be trusted and secure, because a single privilege-escalation vulnerability anywhere in the chain obviates all the protections provided by the model. So while this model is attractive in its simplicity, the size and complexity of the trusted code base makes it far from ideal. In that sense, all the other models improve on user permissions, and make this model appropriate only for systems with limited exposure.

Process jails. Process jails are an attractive model in that they seem to have a much smaller code based (only the kernel needs to be trusted to enforce the jail restrictions), but with a strong caveat that they are not intended to be security solutions (see [3]); they are intended only to provide limited process isolation. Specifically, privilege escalation attacks that grant root privileges can allow a process to escape the jail, meaning that jails suffer from the same unwieldy trusted code base that user permissions do. Given that setting up a jail is nearly as complicated as other solutions that provide real security guarantees, this model is also far from ideal.

Mandatory access controls. As mentioned above, MAC systems make the strongest security guarantees of any model discussed here. They truly limit the trusted code base to the kernel itself, and since they are designed to handle hardened, critical security applications, they can be relied upon to protect against all but the most brilliant attacks. Ironically, this hardening is their biggest downside for this application,

since the configuration of a MAC system is far and away more complicated than any other model, and requires constant administrator involvement to setup and maintain. The value of MAC as a security model must be weighed, not only against the potential for malicious behavior, but also against the consequences of a successful attack. It seems unlikely than any attack on an automated submission system could have repercussions that warrant such a step.

Virtual Machines. Here we use the term "virtual machine" to refer exclusively to hosted operating systems (VMWare, etc.). The Java Virtual Machine provides excellent security guarantees, but the requirement of running all projects on the JVM is unrealistic[8]. A hosted operating system is clearly the best option for most submission systems. Configuration is no more difficult than setting up the build server in the first place, and the trusted code base is limited to the virtual machine itself. This should be comforting, since virtualization technology is commonly used in industry, with many companies relying on the security guarantees of virtual machine software. Additionally, deploying updates to a virtual machine is simple (as was mentioned above), so they can be moved between computers easily, and different versions can exist side-by-side on the same build server, allowing projects to be compiled and tested in a customized environment. It is for this combination of simplicity and real security that makes a hosted operating system an ideal choice for automated submission systems.

4 Conclusion

Safely executing untrusted code is one of the most beguiling questions in computer security today. In educational settings, where students are submitting code that is to be automatically executed for testing, submission systems must compile and test untrusted code many times throughout the semester. While actual malicious intent may be unlikely, it is not impossible and the chance for harm due to poorly-written programs cannot be ignored. Unfortunately, research on automated evaluation tends to focus on features or outcomes, raising serious concerns about the security of extant submission systems, and once these systems are opened up to students from across the Internet, the threat of bad actors become a paramount concern. This issue has not been ignored by the security community at large, and there are many mechanisms available to counteract the threat that untrusted student code poses. The discussion above is of course only a partial list of the possible security models and techniques, but the proper use of those tools would represent a large step forward in the current state of the art of automatic submission and evaluation systems.

[8] Java-specific projects can benefit from the JVM security mechanisms in addition to the more general security model

References

[1] Bug 40357 - [4.5 regression] compiler hang for c++ code, . URL `http://gcc.gnu.org/bugzilla/show\_bug.cgi?id=40357`.

[2] Bug 100119 - fix bug 4421494 infinite loop while parsing double literal, . URL `https://bugs.openjdk.java.net/show\_bug.cgi?id=100119`.

[3] Jeremy Andrews. Abusing chroot. URL `http://kerneltrap.org/Linux/Abusing_chroot`.

[4] Michal Forišek. Security of programming contest systems. In *Informatics in Secondary Schools, Evolution and Perspectives*, 2006.

[5] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.

[6] Michael Luck and Mike Joy. A secure on-line submission system. *Softw. Pract. Exper.*, 29(8):721–740, July 1999.

[7] Roland H. Untch, Ralph Butler, and Chrisila C. Pettey. A small and secure submission system for unix systems. In *Proceedings of the 43rd annual Southeast regional conference - Volume 1*, ACM-SE 43, pages 341–344, New York, NY, USA, 2005. ACM.