Olympiads in Informatics, 2009, Vol. 3, 67–73

Using a Linux Security Module for Contest Security

Bruce MERRY
ARM Ltd
110 Fulbourn Road, Cambridge, CB1 9NJ, United Kingdom
e-mail: bmerry@gmail.com

Abstract. The goal of a programming contest grading system is to take unknown code and execute it on test data. Since the code is frequently buggy and potentially malicious, it is necessary to run the code in a restricted environment to prevent it from damaging the grading system, bypassing resource constraints, or stealing information in order to obtain a better score.

We present some background on methods to construct such a restricted environment. We then describe how the South African Computer Olympiad has used a Linux Security Module to implement a restricted environment, as well as the limitations of our solution.

Key words: linux security module, programming contest, sandboxing.

1. Introduction

The South African Computer Olympiad (SACO) is an annual programming contest, whose final round is modelled on the International Olympiad in Informatics (IOI). In particular, contestants submit solutions in source form, in a variety of languages, to an online submission system. The online submission system provides a variable amount of feedback immediately to the contestant (typically, the results of some sample case). Final scores are only made available after the contest, by running each solution on a variety of test cases.

While we are not aware of any contestants having attempted to cheat by submitting malicious solutions, we must nevertheless protect the integrity of the grading system by running them in a *sandbox*, or locked-down environment. In some ways, this is easier than sandboxing a general application, since solutions are only intended to have a limited interaction with the execution environment (read a file, do some computation and write a file), and so it is possible to use a more tightly locked environment than would be possible for running applications that had legitimate needs to access the display, keyboard, network and so on. However, we must also enforce the rules of the competition (such as execution time limits), so some additional work is required.

The following section lays out our requirements in more detail. Section 3 discusses a number of approaches to sandboxing. In Section 4 we describe which approach we chose and how we implemented it. Conclusions are presented in Section 5.

2. Requirements

2.1. *Security Requirements*

Usually, sandboxing is used to prevent applications from either damaging the host environment or leaking information about it (for example, by stealing passwords or credit-card numbers from a browser). For a programming contest, there are some other limitations that must be taken into account. Below is a list of some of the goals we aimed to meet:

1. Resources (particularly, CPU time and memory) must be restricted, to prevent a malicious (or more likely, incorrect) solution from blocking the whole system.
2. Resource constraints must be accurate. For example, if a program has a one-second time limit, it is acceptable for it to run for 1.5 seconds and terminate normally, as long as it is possible to determine that the time limit was in fact breached. It is also important that programs are not able to make it appear that they used fewer resources than they actually did.
3. Programs must be limited to a single thread. This prevents programs from taking advantage of multi-core systems to get more computation done in the time available, as well as simplifying a number of other implementation details.
4. Programs must not be permitted to spawn other processes. This prevents, for example, a problem involving mathematical computation from launching an external program like `bc` or `Octave` to do the computation.
5. Programs must not be able to communicate with the outside world (for example, through TCP/IP sockets). This prevents the solution from offloading processing onto a separate, possibly faster system, as well as ensuring secrecy of test data.
6. A single run of a program must not be able to communicate with any other run (for example, by leaving a file with precomputed primes in the filesystem, or using inter-process communication).

2.2. *Other Requirements*

While security is obviously the primary goal, some potential solutions may be deemed unusable for other reasons. Other requirements include

1. The setup time must be minimal. We do not have a large cluster of machines for evaluation, so throughput is a concern.
2. It must not significantly impact performance. This is so that CPU time limits are not affected by the security mechanism.
3. It must allow common operations by the standard libraries of the compilers used (GCC for C and C++, FreePascal, Python and Java). Some of these libraries do I/O using system calls that can also be used in ways that violate the security policies above, so it is unacceptable to block these system calls unconditionally.

3. Background

3.1. *System Call Interception*

In Linux and indeed most desktop/server operating systems, applications do not have direct access to devices such as ethernet ports or disk drives, and can only access memory that is allocated to them. In order to interact with devices or other applications, it is necessary to use *system calls*. A system call is similar to a function call, but transfers control to the operating system kernel which undertakes these actions on behalf of the user.

One approach to restricting the actions that an untrusted application can take is to intercept these system calls. Linux provides the *ptrace* system call, which allows one process to be notified about any system calls made by another. The controlling process can then override or suppress system calls that the untrusted application should not be allowed to make according to the security policy.

This interception process adds a small amount of overhead, because for each system call there is an additional context switch from the kernel to the process that makes the decisions and back again.

System call interception is also prone to security holes if not implemented correctly, mostly due to race conditions (Watson, 2008). This is because the values passed to the kernel can be modified by another thread between the time they are checked by the interceptor and the time the kernel sees them. However, this is less of a concern for us, since we do not need to support multi-threaded processes.

Another limitation of system calls is the sheer number of them: over 300 in current versions of Linux. Arguably, the system call interface is the wrong level of abstraction, because the same semantic operation (for example, extracting data from a file handle) can be achieved with many different system calls.

Examples of general-purpose, configurable system call interceptors include Systrace (Provos, 2003) and GSWTK (Fraser *et al.*, 1999).

3.2. *Linux Security Modules*

Linux provides an interface by which alternative security policies may be plugged into the kernel. Whenever the kernel is about to undertake some privileged action on behalf of the user, a hook in the current security module is used to determine whether it should be permitted. This is also used to implement the default security policy (for example, to allow the *root* user to access any file).

The interface for a security module is at a more appropriate level for our task: it deals with abstract actions, such as read from a file, rather than the specific system call used to achieve that action. It is also less vulnerable to the same race conditions as system call interceptors, because the security module accesses data within the kernel rather than userspace data that will later be copied into the kernel.

Unfortunately, the interface is frequently changed between kernel releases, and the documentation of the interface is often not updated to reflect the changes. This means

that a system based on the module cannot freely upgrade the kernel to take advantage of new features or security fixes to the kernel itself.

3.3. *Virtualisation*

Virtualisation allows one instance of an operating system (the guest) to run inside another (the host). When the untrusted program is run inside a guest operating system, it will be unable to access the resources of the host as it has no way to even address them (for normal uses of virtualisation, special configuration must be done to make host resources accessible from the guest).

This provides a high level of security, since rather than having to catch bad system calls, there is no way for a malicious application to even form a bad system call. And because any side effects of the program (such as leaving files in a temporary directory) are limited to the guest operating system, they can be wiped and a pristine guest operating system used for the next run.

The primary disadvantage of this approach is that the guest operating system would need to be booted for each run, adding significant overhead to the evaluation process.

4. Implementation

In the previous section, we listed three general approaches: system call interception, security module, and virtualisation. At the time we made the decision, we were not aware of the general-purpose system call interceptors, and writing one from scratch seemed a daunting task. Virtualisation was rejected because we wanted a light-weight setup that would not require us to maintain an operating system image separate from the primary operating system on the evaluation server. At the time, we were also only using a single machine for both the web front-end, compilation and evaluation, and we felt that booting a virtual operating system for each evaluation would be too expensive. We thus chose to use a Linux Security Module.

The Linux Security Module (LSM) framework provides a communication channel (`/proc/self/attr/exec`) for user processes to communicate with the security module. A wrapper program uses this channel to configure a restricted environment, then calls `exec` to launch the untrusted program. There are a number of commands that can be sent using this stream. There are some commands that are specific to Java (see 4.5); the remaining ones are listed in Table 1).

Once the process calls `exec`, the security restrictions come into place, and cannot be changed further. The `allow exec` command exists to allow further layers of wrappers around the actual program to execute (some interpreters, for example, are actually shell scripts which `exec` the “real” interpreter.

The majority of security module hooks are for more exotic functionality that a contest solution should have no need for, such as setting or querying scheduling policy, changing user, inter-process communication and so on. For each of these, we simply check whether we’ve flagged the task as restricted (which is quite easy, since the kernel task structure

Table 1
Commands that can be issued to the security module

Syntax	**Meaning**
`version` *major*`.`*minor*	Mark this process as restricted, and check for version mismatches
`allow threads` *n*	Set the number of threads the process may have at any time
`allow exec` *n*	Allow *n* calls to `exec`
`allow write`	Remove any extra restrictions on filesystem access
`allow write` *file*	Allow creation and write access to *file*

has a field available for the security module to store information), and if so, reject the call with the appropriate error code.

For calls related to creating or writing to files, we check both the global write enable, and the list of files marked as writable (for a contest problem, this would typically be just the output file). Note that there are multiple system calls to modify the data in a file (`write`, `fwritev`, `pwrite`, `truncate`, `ftruncate`, `sendfile`, ...), but they are all handled with the same security module hook – a distinct advantage over system call interception. In addition, the parameters to the hook use the kernel's internal representation of the filesystem, so there is no need to compare pathnames to determine whether are simply different ways to refer to the same file (due to symlinks, for example).

Initially we attempted to block all operations for which we did not explicitly see a need in a contest environment. However, we were surprised to discover the extent to which standard libraries depended on these calls for internal use. For example, we found that `glibc` preferred to use `mmap` for file I/O, and various files in `/etc` are consulted during library startup. In the end, we decided that it was easiest to rely on just the standard kernel security model for most read-only operations, and block or restrict only operations that had side-effects.

4.1. *Separate User Account*

Before the introduction of the kernel module, we used to have a weaker security system that merely executed programs under a different user ID, using `sudo` (Miller and Jepeway) to allow the user ID that runs the submission system to launch processes under this user ID. We decided to keep this model when adding the kernel module. While partly for defence-in-depth, this made it possible to allow the normal UNIX file access controls to govern read access without exposing the full set of test data, results etc. to submitted programs.

4.2. *Resource Limits*

The security module prevents multithreading and process execution, but CPU time and memory limits are enforced via the standard `setrlimit` system call. We use a wrapper program that forks, sets the limits and the security module settings, and finally launches

the program. It then waits for it to terminate and records the results (such as the exit code and execution time). `setrlimit` can only set a CPU time limit with one-second precision, so we round up the time limit for the purposes of `setrlimit` and check the actual time consumed after termination.

4.3. *Side Channels*

When the kernel module was initially introduced, jobs were run in parallel, starting as soon as they were submitted. We did not find a good way to prevent side channels between processes running concurrently. Although we are able to block the obvious routes such as sockets or files in a common area, there is a wealth of information available in `/proc` and our attempts to block access here led to instabilities in the kernel module itself. It is also known that shared caches between processors can be used to extract information by timing memory accesses, even if one of the processes does not intend to leak this information (Osvik *et al.*, 2006).

In our current system, jobs are queued and executed serially on each grading server, so side channels between running processes are not a concern. The most obvious side channel between processes that do not execute concurrently would be to leave a file in the filesystem. This is prevented by restricting the processes to write to only a specified list of files, all of which are purged after execution. There may still be side channels available (particularly related to uninitialised memory), but we believe that exploiting them reliably would be at least as much work as solving problems correctly in the first place.

4.4. *Alternate Root Directories*

In Linux (and other UNIX-like operating systems), it is possible to run a process in an environment where the root directory is actually only a subdirectory of the "real" root directory. This prevents the process from accessing any files other than those specifically placed in that subdirectory.

At present, we have not implemented this, largely due to the unwillingness to maintain separate copies of files in the alternative root filesystem. While we're not aware of any additional security we would gain from this, it would provide better defense-in-depth should any of the other security provisions fail (for example, should any of the contest test data become world-readable by accident).

4.5. *Java*

The Sun JVM (Java Virtual Machine) performs a lot of operations that would normally be blocked by the kernel module. While we initially tried to use the security module unchanged for Java, we found it impractical to let through the system calls that the JVM needed while simultaneously keeping the system secure for C and C++ programs. We have instead used the Java security manager to limit solutions to legal operations, and the command line option `-Xmx` to limit the maximum Java heap size.

The Java security manager uses a policy file which describes which privileged actions may be undertaken by which classes. The default configuration is quite permissive, allowing various operations not suitable for an olympiad (such as write access to any files, subject only to OS-level checks). We use the `-Djava.security.policy` command-line option to specify our own policy file. This custom policy limits all classes to just a list of explicitly allowed permissions – mostly querying of Java system properties, but also general read access, and write access to the output file. The file format permits a variable expansion syntax, so we are able to use a single file and provide the name of the output file on the command line for each evaluation.

5. Conclusions

When we started this project, we were under the impression that the Linux Security Module interface was a reasonably stable interface, suitable for third-party development of custom security modules. However, the interface changes with almost every kernel release, and maintainence has been more difficult than expected. Posts to the Linux Kernel Mailing List (Edge, 2007) suggest that in fact the interface is only intended for security modules maintained within the kernel tree, and as of Linux 2.6.24, it is no longer possible to build modules outside of the kernel tree. While it is a simple, low-overhead interface at a good abstraction level, we will have to consider whether other options will require less maintainence in the long term.

References

Edge, J. (2007). LSM: loadable or static? *Linux Weekly News*, 25 October.
`http://lwn.net/Articles/254982/`

Fraser, T., Badger, L. and Feldman, M. (1999). Hardening COTS software with generic software wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May, 1999.
`http://www.isso.sparta.com/opensource/wrappers/`

Miller, T.C., and Jepeway, C. *Sudo Manual*.
`http://www.sudo.ws/sudo/man/sudo.html`

Provos, N. (2003). Improving host security with system call policies. In *12th USENIX Security Symposium*.
`http://www.citi.umich.edu/u/provos/systrace/`

Osvik, D.A., Shamir, A. and Tromer, E. (2006). Cache attacks and countermeasures: the case of AES. In *Proc. RSA Conference Cryptographers Track (CT-RSA) 2006*. Springer, 1–20.

Watson, R.N.M. (2008). Exploiting concurrency vulnerabilities in system call wrappers. In *WOOT'07 First USENIX Workshop on Offensive Technologies*.

B. Merry took part in the IOI from 1996 to 2001, winning two gold medals. Since then he has been involved in numerous programming contests, as well as South Africa's IOI training program. He obtained his PhD in computer science from the University of Cape Town and is now a senior software engineer at ARM.