Olympiads in Informatics, 2010, Vol. 4, 113–119
© 2010 *Institute of Mathematics and Informatics, Vilnius*

Validating the Security and Stability of the Grader for a Programming Contest System [*]

Tocho TOCHEV, Tsvetan BOGDANOV
Sofia University, Bulgaria
J. Bourchier 5, 1164 Sofia, Bulgaria
e-mail: tocho.tochev@gmail.com, tsvetan.bogdanov@gmail.com

Abstract. Automated judging systems have had the tough task of ensuring the normal proceeding of programming contests for a long time. There are numerous ways proposed to secure the execution of a contestant's solution and many more actual implementations. In this article we review some criteria which the core of such systems must meet in order to be considered stable and secure. We will focus only on the core, as the functions it performs are similar across the variety of existing systems and it is the part of the system designed specifically to withstand attacks. In fact these criteria were created with their respective test cases in order to verify the grader we used in IOI 2009.

Key words: contests system, judging system, sandbox security, grader security, grader testing, grader test harness, "black box" testing, IOI 2009.

1. Introduction

Competitions in Informatics are used to evaluate the algorithmic thinking and programming skills of their participants. The first competitions involved judges manually reading and verifying the source code that the competitors submitted to them. With the evolution of the contests and of the difficulty of the problems this task became extremely daunting and potentially inaccurate.

This is how a myriad of automated judging systems were born – Moe, PC^2, USACO's, Top Coder's, Spoj0, SMOC are only some of the examples.

Of course, this presented new ways for people to interfere with the normal workflow of the competition by exploiting security loopholes. For example, to be able to run their programs using more resources than they are allowed, gaining insight about the test data, gaining access to a solution from another person, hindering the participation of other competitors, etc. The motivation for such misbehavior can range from material prizes, such as money, to acceptance in a better university. Therefore, it is important to have the automated system as secure as possible.

Unfortunately, the current implementations are uniform neither in architecture, nor in their approach to performing the submit evaluation. In this article we are going to focus on validating the security and stability of the core part of a programming contest judging system (the 'grader' as we will call it). The term stability stands for verifying the

[*]This work was partly supported by the Foundation "America for Bulgaria" and the Scientific Research Fund of Sofia University under the contract No. 247/2010.

correctness of the normal submits, while security focuses on filtering out the malicious ones.

Similar studies have been made by Forišek (2006), where he gives a classification of a number of attacks against programming contest systems. On the other hand the criteria we review are based on their usage in functional tests for SMOC, the system we used for IOI 2009 and the Bulgarian competitions in informatics. The need to create such a list arose when we unified the way SMOC and Moe (Mareš, 2009) sandbox the contestant's program execution. Moreover, any significant refactoring on a grader should be followed by thorough examination of the resulting module. We have limited the scope of these criteria to the grader, and will not discuss any attacks related to the parts of the system visible to the contestants (such as securing workstations, traffic sniffing, activity auditing, hacking the web server serving the contest system, "Denial of Service", and others).

Unfortunately, there is no such thing as 100% guarantee that a software solution is stable and secure. However, there are certain actions that can be taken in order to raise the confidence in it. The most common are:

1. Peer reviews of the source code.
2. Real-world testing by interaction.
3. Automated testing.

The code reviews are important and can identify problems the other methods cannot. However, as we are only human, they are not very reliable. Real-world testing by interaction, such as online contests, is a nice way to test the overall system. Unfortunately, it does not cover all cases and organizing such an event is time consuming. In contrast automated testing is "cheap" (write the test once, run it after you have changed the code), has good coverage, and can prevent regression bugs. It also allows for a quick inspection of a new server setup (e.g., different OS).

We will focus only on automated testing, however, it is always preferable to use a combination of these methods for verifying a programming contest system.

2. Implementing a Test Harness Around the Grader

2.1. *Functions of the Grader*

As we already stated the grader is the core of a programming contest judging system. In different systems this component may be comprised of several subcomponents, and it may have different levels of coupling with the other parts of the system.

In order to create a test harness we first need to know the functions that the grader performs. We define them as:

1. Compiling the source code.
2. Running the resulting program on some input in "restricted" mode (also known as sandboxing).
3. Running some checks on the program's output (for instance check how compatible the program's and the judge's outputs are).

The requirements that the grader must meet include:

1. Enforcement of the programming task's resource limits (processor, memory, number of threads/processes spawned).
2. Enforcement of unaided program runs – preventing communication with outside world, usage of temporally files or reads of the judge's output files; limiting precomputing, etc.
3. Enforcement of the usage only of tools approved for the competition (for example, restrict unsanctioned libraries or calls to external programs).

As can be seen the implications of an incorrect or compromised grader can be severe – usage of additional resources, obtaining access to the judge's output files and simply printing them, sending vital information about the input files back to the contestant, using pre-computed values between runs, "blocking" the grader machine and therefore hindering the whole competition, or in other way altering the results of the competition.

2.2. *Ensuring Stability*

However, before the grader is secured it must first properly perform its functions in an environment with no malicious programs. To verify that, we decided to create a simple task for each of the task types that we support. So we ended up with:

- Batch tasks – *aplusb*: Given 2 integers output their sum.
- Reactive tasks – *binsearch*: Given a range $[A, B]$ guess a number in that range, by asking "Is it x?", and receiving "up" and "down" hints.
- Output-only tasks – *output*: Give a file with a single line.

In order to ensure the stability we decided that we must cover at least the following cases:

- Correct solution.
  For each task type and for each programming language we have a correct solution. This assures us that all compilers are present and working, as well as that the whole process runs smoothly.
- Wrong answer solution.
  One wrong answer solution (for instance "off by one") per task type. As well as some solutions that are partially correct.
  For the output only task we had a solution which failed the format checker (which is responsible for accepting the output-only solution for judging).
- Time-limiting solution.
  For batch and reactive tasks we implemented a test solution that outputs the right answer and after that falls in an infinite loop. We did this in order to guarantee that even if the contestant's program is correct but fails to terminate we assign to it a time limit exceeded resolution.
  Other test solutions that need to be included for reactive tasks are deadlocking ones. Note that the deadlock might occur in both the contestant's judge's programs – an excessive read would cause the contestant to wait forever and not writing enough information might cause the judge's program to block.

It is important to note that writing a solution exceeding the allowed time limit might be tricky. For example, the following code will be optimized by the compiler "`for (long long i=0;i<(1«50);i++);`" to constant time.

- Memory limit exceeding solution.
  We have C solutions with both dynamically allocated memory, and with static allocation. The amount of memory allocated needs to be slightly over the actual memory limit. A correct solution which uses slightly less than the actual memory limit should also be used.
  If there is a separate stack limit it should also be checked. And if there is no such limit, this also needs to be verified (i.e., static allocation should be done in the stack).
- Runtime error.
  There should be several sub-tests for runtime errors. A simple division by zero can be used to achieve a 'Floating point exception'. There also needs to be a solution that has a non-zero exit code as that can also signify runtime error. Furthermore, it is nice to have a simple invalid memory reference solution as some graders may filter these separately.
- Use of library functions.
  For every allowed language there need to be sample solutions that include permitted and restricted libraries. For example, for C++ there should be checks for `stl` (e.g., `vector`), `boost` and possibly `tr1`.

2.3. *Ensuring Security*

As we have stated the grader has three main functions – compilation, program sandboxing, and output evaluation – and each of them can be attacked.

2.3.1. *Attacks During Compilation*

Attacks aimed at compilation are very dangerous as they can make the grading system unresponsive or expose the judge's solution.

- Excessive submit size.
  The simplest "attack" is just pre-computing the answers for every possible test case, and inserting them in what becomes a 50MB source file submitted to the system. Therefore, the submit file size should be limited. Note that the actual limit may vary from task to task, since some tasks may allow more precomputing than others.
- Referencing forbidden files.
  Another attack is including files in the compilation that are not supposed to be included, such as "`#include<boost/lambda/lambda.hpp>`", or the more aggressive and dangerous "`#include"../solutions/judgestask1.cpp"`".
- "Denial of Service" and compile-time exploitation.
  Failing to impose reasonable limits (both time and memory) on the compiler can lead to pretty nasty attacks, such as "`#include"/dev/zero"`" (running "forever" while consuming more and more memory). A lack of compilation limit can

also be leveraged by the competitor to gather useful data using "Template metaprogramming" (e.g., compute the first N factorials or prime numbers, etc.). It is also possible for the contestant to cause "Denial of Service" by consuming too much memory or CPU during compilation.

2.3.2. *Attacks During Sandboxing*

As mentioned graders can use different ways of sandboxing – syscall interception (Mareš, 2007), virtualization, linux security modules (Merry, 2009), java virtual machine security profiles, and others. Depending on the type of sandboxing there are specific ways to try an attack.

However, the general things that we do not want contestants to do while sandboxed are:

- Be able to read/write files/directories.
  There needs to be a test for at least some of the important files that should be forbidden – the judge's solution, additional input files, output files, checkers and the grader itself. Some attention should also be paid when using syscall interception and killing the program when it seems to access 'unneeded' files as we found out that the current standard libraries of the programming languages seem to access some non-obvious 'files' (for instance, `qsort` can require access to `/proc/meminfo`).
  Also the policy which is taken towards the people who try to open files should not always be immediate ban, as we have witnessed lots of submits with forgotten "`freopen("mytest.txt", "r", stdin);`" statements.
- Be able to open sockets/access the network.
  It is imperative that the solutions cannot open sockets either as a server, or as a client. A break in this policy might allow the submit to mimic some important part of the system or send out vital information about the tests. Furthermore, this restriction does not only cover internet sockets but also any other protocol used by the contest system (e.g., unix domain sockets).
- Be able to spawn multiple threads.
  We usually want to check that the competitors cannot create multi-threaded applications as this might give them an unfair advantage. However, there might be competitions where this is actually encouraged.
- Be able to spawn multiple processes.
  The simplest test case here would be just using "`fork()`". Although, having multiple processes poses higher security threat than multiple threads, not every time that a contestant does an exec call, he should be banned. For example, some people under Windows use "`system("pause");`" for debugging and forget to comment it out before submitting.
- Be able to raise their privileges, or be able to break the sandbox.
  This is the class of attacks that are most sandbox-specific. The test cases can include buffer overflow attacks, `setuid` calls, breaking out of `chroot`, exploiting vulnerabilities in the system call wrappers, and others.

Although, it would be interesting to make a collection of such programs, it may be the case that they have to be tuned a little bit for the different sandbox classes.

- Protecting the judge's module in reactive tasks.
  In reactive tasks the judge's module must be protected against any attempts on its integrity. It is important to note that even a well sandboxed contestant's program can be successful in an attack if the judge's module is not written properly (e.g., allows buffer overflows from its input). Such an attack is difficult to perform and close to impossible unless the contestant is provided with the module's source code. Unfortunately, it is also hard to create a unified test case.
- Denial of Service during sandboxing
  There are a virtually infinite number of attacks in this class. However, there are several easy-to-test examples. First, the memory restriction model needs to disallow allocating any memory above the set limit. Any loophole (e.g., if the memory used is checked in regular intervals) might lead to exhaustion of the system resources (CPU or memory) and therefore non-responsiveness of the evaluating machine.
  Another check to consider would be outputting too much information (either on `stdout`, or on `stderr`). Although, the competitors might do this unintentionally, it can also lead to abnormal grader behavior. It can also lead to waste of disk space which will manifest itself later and should generally be avoided.

2.3.3. *Exploits During Checking*
The attacks that fall into this category are the result only of the output that the competitor's program has made. Mostly these exploits are not intentional, and are results from lack of foresight from the judges. Take for instance the segment "`while(i!=-1) {scanf("%d", &i); ...}`" in the judge's checker. It will lead to an infinite loop if the contestant never outputs -1. Therefore some precautions should be taken in terms of limiting the resources available to the checkers. Obviously, there are no checks that can cover large portions of the errors that can be done in checkers. However, a test that can be made with a custom checker that hangs and thus verifies the grader fallback mechanisms, in case of malfunctioning checker, are working (e.g., this is reported to the judges).

3. Conclusion and Further Work

It would be great if there was a complete checklist that would cover all possible vulnerabilities, however, such a thing cannot exist. What we have shown in this article is what we think are the most important tests cases. Every system for judging programming contests should be able to pass them in order for it to be considered at least somehow stable and secure.

Having such a checklist proved very useful for us during the preparation for hosting IOI 2009. We needed to organize a series of competitions, and each one of them introduced new challenges for the grading system – various operating systems, hardware configurations, as well as minor changes in the judging criteria. As part of the setup for each of these events we needed to verify the functionality of the grader. Thanks to the test suite we painlessly exposed several issues and gained confidence in any modifications that we have made.

Furthermore, some of the test cases can be reused for verifying other functional areas. Such an example was the protocol change we made between the grader and the dispatching module. Moreover, we used the test solutions as model contestants' submissions to create load tests. However, notice that a comprehensive load test also includes solutions specific to the competition. Of course, for any large competition (such as the IOI) the host also needs to run a series of other tests.

For the benefit of anyone interested, we provide our tests as part of the SMOC's source code[1]. However, it seems to be a good idea to setup "a universal grader test repository", open to everyone, accepting proposals from contest system maintainers, in order not to duplicate the effort that goes into testing a grader.

Perhaps this work can be extended and can lead to a number of stability and security checks that every IOI host system can be verified against.

References

Forišek, M. (2006). Security of programming contest systems. In: *Information Technologies at School*, 553–563.

Mareš, M, (2007). Perspectives on grading systems. *Olympiads in Informatics*, 1, 124–130.

Mareš, M. (2009). Moe – design of a modular grading system. *Olympiads in Informatics*, 3, 60–66.

Merry, B. (2009). Using a Linux security module for contest security. *Olympiads in Informatics*, 3, 67–73.

T. Tochev is a master student in artificial intelligence at Sofia University. Former competitor himself, he helped with the technical organization of many Bulgarian competitions in informatics and some international ones – JBOI 2008, BOI 2008, and IOI 2009.

T. Bogdanov is currently completing his masters in software engineering at Sofia University. Involved with grading systems since the Balkan Olympiad in informatics in 2004, he has helped with grading many of the national high school competitions in informatics and IOI 2009.

[1]`https://svn.openfmi.net/pcms/trunk/test/grader/.`