Additional Guidelines and Suggestions for Project Milestone 1

CS161 Computer Security, Spring 2008

Some students may be a little vague on what to cover in the Milestone 1 submission for the course project, so we have written this document to provide some more specific guidance and ideas.

Design Considerations

First of all, each team should spend a little time getting some background on the PostScript language. We recommend you take a look at the *PostScript Language Tutorial and Cookbook* (a.k.a. the "blue book"), which can be obtained by searching online or from `http://inst.eecs.berkeley.edu/~cs161/sp08/Projects/psintro.pdf`. Don't worry about reading that document in detail! You should be fine just reading the first couple chapters and quickly skimming the rest, which should only take about 30 minutes or so. Also, the operator summary in the appendix may be a handy reference.

The main question to be addressed in your design document is how you will represent the syntax and (possibly to some extent) semantics of the PostScript language within your program. The most rudimentary approach would be to include a list of PostScript keywords / built in operators ("add", "def", "dup", "showpage", etc.), then construct a test document by just concatenating random sequences of these along with string and numeric literals. However, most such sequences will simply cause the PostScript interpreter to immediately exit with an error. For example, any sequence that begins with "add" or any other operator that takes arguments will cause a stack underflow as soon as the first token is read.[1] So in order to better test the

[1]Of course, it would be good for your fuzzer to be able to generate highly invalid

interpreter, you will probably want to at least take into account the number of arguments that each operator takes (pops off the operand stack) and the number of results it returns (pushes back on to the stack). Given that information, a random PostScript program could be constructed as an expression tree, then output in postfix order to form a document which is valid at least in terms of the operand stack. Since there are operators which take and return a variable number of arguments (e.g., "copy"), you may not always be able to ensure this sort of validity, but that's okay. You just want your fuzzer to be capable of eliciting a fairly wide range of behaviors from the interpreter rather than causing it to immediately exit with an error every time.

Another thing to consider is whether your fuzzer will give any particular operators or language constructs specific testing treatment. For example, the "get" operator indexes into arrays, so maybe you would want your fuzzer to test a range of negative or out of bounds values for the index. As another example, you might want to test the interpreter on various sorts of quoted string literals such as very long ones or ones with special characters. If you have any specific types of tests such as these in mind, they could be either incorporated into your primary method for generating a random PostScript program / document, or they could be separated out as distinct tests via the `SPEC` argument given to your fuzzer (see main project description).

When considering these issues with respect the specially modified `pstotext(1)` program which will be provided for testing and grading, don't pay too much attention to how that program works. In particular, don't worry about how the actual `pstotext` program is implemented; you might as well assume we implemented ours from scratch. Our only intention in selecting this program was to pick a program that reads PostScript, so just assume it uses a complete PostScript interpreter / renderer internally rather than using some sort of shortcut that just extracts text.

However you choose to design the high level approach to generating random PostScript documents, ideally your design will fit into a more general framework for specifying data formats to your fuzzer (for example, the probabilistic context free grammars mentioned in the main project description). While not a requirement, a more general system will be easier to adapt to other programs, which may allow you to gain extra credit without much additional work.

programs such as these, but if that is all it can do, its tests will not be very comprehensive.

Implementation Considerations

To get you thinking about your design in greater detail, we've included below a list of implementation-level questions you should consider addressing in your Milestone 1 submission (in addition to the high level design issues discussed so far). It's not necessary to answer every one of these questions in order to get full credit on your submission, but the more you address, the better the feedback we will be able to give you early in the project. Our goal is for the project to go smoothly and easily for each team, minimizing the work necessary to reach a satisfactory level of success!

- **What programming language(s) will you use to write your fuzzer?** Will there be any difficulties in compiling and running your code on `ilinux1.eecs.berkeley.edu`?

- **Are there any libraries or other external sources of code you plan to use?** If there is any library or existing source code that would reduce your work, we absolutely encourage you to use it. We're not trying to test your ability to write computer programs – the goal is to learn about fuzz testing.

- **If you are planning on using a library, is it available on the ilinux machines?** If not, will you be about to bundle it up in the tarball you eventually submit so your program works? It's fine to include a library or other external code in compiled form if that is the easiest way to get it to work. Your code, however, should be included as source.

- **How will you spawn the program being tested as a child process and check whether it has exited due to signal 11 or a timeout?** In Linux, this is typically done through a combination of the `fork(2)`, `execve(2)`, and `wait(2)` (with `WNOHANG` in order to allow for a timeout) system calls. In C/C++, they can be called directly; other languages often provide a higher level interface to this functionality.

- **How will your fuzzer generate pseudorandom numbers?** Just to keep things straight, note that there are two separate "random" things that your fuzzer will need to do. First, for each test run, it will need to pick a new pseudorandom number generator (PRNG) seed. This need not be reproducible, so you could just read however many bytes the seed needs from `/dev/urandom`.[2] More importantly, given a seed value, your fuzzer will have to use it to seed a PRNG, then use that PRNG for all the random selections made in the course of constructing a test input. Possibilities for this PRNG include using one in some sort of standard library, one which is a built in feature of the programming language you are using, one you downloaded source code for, or one you wrote yourself. Note that it doesn't have to be cryptographically secure, but you may want to consider how large a seed it can handle. A PRNG with a 32-bit seed will be limited to generating about four billion distinct test inputs.

- **Are there any particular programs other than the special testing program which you are considering trying your fuzzer on?** Running your fuzzer on other programs isn't a requirement of the project, but may be a good way to get extra credit. In particular, without any extra work, you should be able to try your fuzzer on any program that reads PostScript, since you will be implementing PostScript generation anyway.

[2]Just going through seeds sequentially would also work fine, you would just need to remember the ranges you've already tested between different runs of your program.

Template Milestone 1 Submission

Below we have provided a sample organization for your Milestone 1 submission, which should be one or two total pages in length. The timetable and division of labor are of course just examples.

CS161 Computer Security, Spring 2008
Project Milestone 1
Team: Alice, Bob, Eve, and Mallory

Design

Here you should describe the high level approach your fuzzer will take to generate random PostScript documents and other types of program test inputs (if applicable). Specifically, describe the space of the PostScript documents that your fuzzer is intended to explore and how it will sample from that space.

Implementation

Here you should discuss the lower level details of how you plan to implement your fuzzer, including choice of programming language, PRNG, etc.

Division of Labor and Timetable

Person	Tasks
Alice	Write up for Milestones 1 and 2, pre Milestone 2 implementation.
Bob	Final write up, pre Milestone 2 implementation.
Eve	Final write up, post Milestone 2 implementation.
Mallory	Post Milestone 2 implementation, testing of additional programs for extra credit.

Date	Goals
Apr 4	Repository, skeleton code, and build system set up.
Apr 18	Code for Milestone 2 complete.
Apr 25	First batch of bugs in testing program discovered.
May 2	Most bugs in testing program discovered, begin searching for extra credit bugs.
May 9	Code and final write up done.