Code Testing through Fault Injection

PETER GUTMANN

Peter Gutmann is a researcher in the Department of Computer Science at the University of Auckland working on design and analysis of cryptographic security architectures and security usability. He is the author of the open source cryptlib security toolkit, and an upcoming book on security engineering. In his spare time he pokes holes in whatever security systems and mechanisms catch his attention and grumbles about the lack of consideration of human factors in designing security systems. pgut001@cs.auckland.ac.nz

Several years ago a friend of mine did some robustness testing on a widely used OpenSource Software Library. He instrumented the `malloc()` call so that it would fail (return a NULL pointer/out-of-memory error) the first time that it was called. On the second program run it would fail the second time that it was called, on the next run the third time, and so on. Then he fired up a test suite wrapper for the library and ran it using the fault-inducing `malloc()`.

Luckily, he'd had the foresight to hard-limit the script he was using to stop after a thousand core dumps rather than running through the full test suite wrapper. The hard drive on his computer still hasn't forgiven him for the thrashing it got, though. So why did something as simple as a memory allocation failure cause such havoc?

Why

Most developers have heard that writing unit tests for their code is a Good Thing, and some of them even include the odd one to substantiate this. What these tests invariably do, though, is exercise the standard code paths, the ones that get taken in the presence of normal input and normal operations by other parts of the system. The code paths that handle exception conditions, for example, memory allocation failures, never get tested. It's exactly these conditions that the instrumented `malloc()` exercised, and as the results show, the performance of the never-tested code in these paths was pretty dire.

The instrumented `malloc()` is an example of a testing technique called fault injection, which tests how well (or, more typically, how poorly) code handles exception conditions. The most commonly encountered type of fault injection is fuzz testing or fuzzing, which throws random input at a program to see how it handles it. One of the first instances of fuzz testing looked at the reliability of UNIX utilities in the presence of unexpected input, finding that one-quarter to one-third of all utilities on every UNIX system that the evaluators could get their hands on would crash in the presence of random input [1]. Unfortunately, when the study was repeated five years later the same general level of faults was still evident [2]. While this shows admirable consistency, it's probably not the result that was desired.

Other studies have looked at the behavior of GUI rather than command-line applications in the presence of unexpected input. One such study examined 30 different Windows applications from a mix of commercial and non-commercial vendors. Of the programs tested, 21% crashed and 24% hung when sent random mouse and keyboard input, and every single application crashed or hung when sent random Windows event messages [3]. Before everyone rolls their eyes and mumbles things about Windows, a related study on the reliability of GUI applications under OS X found them to be even worse than the Windows ones [4].

A second type of fault injection involves inducing specific execution failures. One way of doing this is through instrumented system calls like the `malloc()` example given earlier. The late Evi Nemeth of the University of Colorado used to have her students link their programming assignments against a custom stdlib/libc in which certain function calls didn't always succeed unconditionally. Most developers know that you need to check whether a `read()`/`fread()` actually managed to read the data that it was supposed to, but how many check an `lseek()`/`fseek()`?

Think of this as a type of control-flow fuzzing rather than the standard data-based fuzzing. Using modified libraries that randomly report failures for system calls that could reasonably be expected to fail, typically implemented as wrappers for standard libraries, makes for a useful testing tool. Some work has already been done in this area, generally looking at ways of automating the creation of fault-injection wrappers for different libraries [5].

The other type of instrumentation that you can use for fault injection is to modify the code itself to inject failures, such as a bit being flipped in digitally signed data, so that the signature check on your SSL handshake fails and your application warns you that the data has been tampered with. If you're thinking "goto fail" or the GnuTLS equivalent at this point, then you'll understand why this type of testing is important.

Implementing this type of fault injection is a bit more laborious than straight fuzzing of either input data or system calls, which rely on the fact that if you make random changes and rerun the code under test often enough then you'll eventually trigger a fault condition.

Statistical fuzzing only works most of the time. If the input data is highly structured, using a tag/length/value or TLV encoding, for example, then any change in the tag or length will be quickly detected and rejected, at least by a properly implemented parser, and any change in the value is irrelevant. To fuzz data like this, you need somewhat exotic and protocol-specific smart fuzzers [6], but that's getting a bit beyond the scope of this article.

What

So if you're trying to catch "goto fail"-style problems, which sorts of faults do you inject and where do you inject them? If what you're implementing conforms to some standard or specification, then the process is, at least in theory, relatively straightforward: You look for any location in the specification where you're required to report an error (e.g., due to a signature check failure) and then inject a fault of that type. If the implementation doesn't detect and report an error, then there's a problem.

The reason why I've said that this works in theory is because most standards seem to focus excessively on the format of messages rather than their semantics. So a standard will describe in minute detail the layout of data elements down to the individual-bit level, but then neglect to mention that if some particular processing step fails you shouldn't continue. For example, here's what the specification for a PKI standard has to say about values that protect against replay attacks:

> The [values] protect the PKIMessage against replay attacks. The [value] will typically be 128 bits of (pseudo-) random data generated by the sender, whereas the recipient [value] is copied from the sender [value] of the previous message in the transaction. [7]

That's the entire description (or non-description) of the replay-protection mechanism. Note how the text carefully describes the size of the value and how it's copied around, but never says anything about checking it, or what to do if the check fails. It's possible to create a fully standards-compliant implementation that has no protection whatsoever against replay attacks because the spec never tells you to use the value to defend against these attacks. And if you're relying on using the specification to determine locations for fault injection, you have to infer what you're supposed to do from the comment that the values "protect the PKIMessage against replay attacks."

This problem is widespread among security standards. The OpenPGP specification, which devotes a full 15 pages to the minutiae of the formatting of signatures and signature data, completely omits what exception conditions need to be checked for when processing signatures or how to respond to them. The only comment in the standard that I could find was a statement that "if a subpacket is encountered that is marked critical but is unknown to the evaluating software, the evaluator SHOULD consider the signature to be in error" [8].

The standards that cover SSH are no better, with the sole check that's required being that "values of [Diffie-Hellman parameters] that are not in the range [Diffie-Hellman prime size] MUST NOT be sent or accepted by either side" [9].

As long as an implementation checks those parameters, it can ignore the signature validity check and be completely standards-compliant.

This lack of information unfortunately means that you'll need to go through and annotate the specification to indicate fault conditions that need to be checked at various locations. This can get somewhat tedious, because many specifications are presented more as a catalog of message types (one side sends message A with the following format, the other side responds with message B in the following format, and so on) than a description of the control flow of the protocol.

An additional complication arises because a particular type of failure, and again I'll use the "goto fail" signature-check flaw as the poster child, can have a number of different causes. In this case a signature check could fail because of any corruption/modification of the signed data, incorrect calculation of the hash value that's signed, corruption/modification of the signature value, and incorrect computation of the signature value. So a full-coverage test needs to inject each of these faults in order to verify that the signature-check code will catch all of the different error types.

When you're thinking about what sorts of faults to inject, you also have to know when to stop. For example, what if you're worried that the code that hashes the data to be signed can detect corruption at the start of the data but not at the end, or a high bit flipped but not a low bit? Eventually, you need to make some assumptions about the correct functioning of standard operations before you start developing an urge to inject faults down at the atomic level.

An alternative strategy that you can use to determine what sorts of faults to apply is to look through the code and make sure that you inject ones that exercise every error path. This isn't such a good approach, though, because it's not certain that the code that you're using as a template to generate your faults is actually checking for all of the error conditions that it's supposed to. This can arise either due to a coding error (the programmer intended to check for an error condition but forgot to add the necessary code, or added the code but got it wrong) or because of a design error (the programmer never even knew that she was supposed to be checking for an error condition). In either case, if the code isn't obviously checking for a fault, you don't know that you should be injecting one.

Figuring out where to inject faults, and what sorts of faults to inject, is by far the hardest part of the process. Once you've done that, you can then get down to implementing the fault injection.

How

Now that you've identified what sorts of faults you want to inject, how do you do it? The most straightforward, but probably also the ugliest, approach is to insert chunks of code inside `#ifdefs` that inject faults at appropriate locations. Eventually, you'll end up with the code encrusted in a mass of `#ifdefs` controlling conditional compilation, and you'll be hard-pressed to resist taking your former Mona Lisa, now turned into the equivalent of a spray-painted bathroom stall, outside and setting fire to it.

A less inelegant way to handle this is to hide the mess behind a macro, or whatever equivalent your programming language gives you. I use `#define INJECT_FAULT`, taking as argument two parameters, an enum that defines which fault to inject and the code to use to inject the fault. The macro invocation:

```
INJECT_FAULT( FAULT_SIGCHECK, FAULT_SIGCHECK_CODE );
```

expands to:

```
if( faultType == FAULT_SIGCHECK )
  {
  fault code defined in FAULT_SIGCHECK_CODE;
  }
```

where `faultType` is a global variable that's set to the appropriate fault to inject, and the fault code itself is just a macro-based paste of whatever you need to use to inject the fault. You'll still get a mass of random code to handle the fault injection, but now it's all squirreled away in a header file where you can't see it anymore, or at least where it isn't obviously plastered all over your Mona Lisa.

Finally, you need to exercise your newly added fault-injection capability. This is pretty straightforward: You run your normal test routines, but this time inject one of the faults that you've set up, for example with `setFault( FAULT_SIGCHECK )`. If your test routine still reports success (or if your code simply crashes), then you've got a problem that needs to be addressed. Do this for each fault in turn and make sure that the error is detected.

So that's how you can test your software using fault injection. It won't catch every problem, but it will help you avoid going to fail.

References

[1] Barton Miller, Lars Fredriksen, and Bryan So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12 (December 1990), p. 32.

[2] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," University of Wisconsin-Madison Computer Sciences Technical Report #1268, April 1995.

[3] Justin Forrester and Barton Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," *Proceedings of the 4th USENIX Windows Systems Symposium (WinSys '00)*, August 2000, p. 59.

[4] Barton Miller, Gregory Cooksey, and Fredrick Moore, "An Empirical Study of the Robustness of MacOS Applications Using Random Testing," *SIGOPS Operating Systems Review*, vol. 41, no. 1 (January 2007), p. 78.

[5] Paul D. Marinescu and George Candea, "LFI: A Practical and General Library-Level Fault Injector," *Proceedings of the Intl. Conference on Dependable Systems and Networks (DSN)*, June 2009: http://dslab.epfl.ch/pubs/lfi.pdf.

[6] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid and Vitaly Shmatikov, "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations," *Proceedings of the 2014 Symposium on Security and Privacy (S&P '14)*, May 2014, p. 114.

[7] Carlisle Adams, Stephen Farrell, Tomi Kause, and Tero Mononen, "Internet X. 509 Public Key Infrastructure Certificate Management Protocol (CMP)," RFC 4210, September 2005.

[8] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer, "OpenPGP Message Format," RFC 4880, November 2007.

[9] Tatu Ylonen and Chris Lonvick, "The Secure Shell (SSH) Transport Layer Protocol," RFC 4253, January 2006.