How to FAIL at Fuzzing, Prospector

From: Ben Nagy <ben () iagu net>
Date: Wed, 15 Dec 2010 14:50:39 +0545

So, I don't have a blog. I know it's probably wrong to use this list as a substitute for one, but I am no stranger to wrong.

Or indeed failure.

You can also skip ahead to the end to read about the runtracing / code coverage / fuzzing template selection tools we released at Ruxcon - as I promised you may now all have your turn to pick holes in our implementation and generally poke fun.

At Kiwicon 4 "New Zealand's best (only) hacker con", some beers and I enjoyed ourselves immensely while presenting a talk entitled "How to FAIL at Fuzzing". That presentation won me a newly minted award for "Most Offensive Speaker", and the slide deck I used will never again see the light of day. However, for those interested, I summarise it below. Whilst I will attempt to retain some of the, ahem, 'earthy' tone of the presentation I feel it wiser to omit the bulk of the swearing, all of the slander, the lesbian pornography, and some of the sarcasm. To underline the fact that I speak of failure from deep experience, I illustrated each of the fails with at least one story from my own work, but I'm skipping most of those here to save space.

#1 - Masturbation

In around 2006 I presented some of my early fuzzing code at Ruxcon, fresh faced and, frankly, hopelessly naive. In that presentation, mainly concerning my mutation primitives, binary structure lib and general rubyness, I recall saying "delivery and instrumentation are up to you", which I still rate as one of the all time most stupid things I have said on stage.

Fuzzing without proper instrumentation is, effectively, masturbation. If you don't have a solid plan for how you're going to instrument your target, then you fail. If you release a tool which just spits out test cases, you fail. If you're waiting for a target to pop Dr Watson and then manually trying to save the crash dump... well, you might not actually fail, but you still suck.

#2 - Autofellation

Fuzzers don't get marks for being awesome, using tricky computer science, or doing anything at all that is smart, automatic, extensible, adaptive, neural, genetic or written in Haskell. They get lots of marks for finding bugs. Take a look at yourself in the mirror before you start adding stuff that sounds cool without a concrete concept of how it will find bugs, and how it will do so in a manner that is measurably better than the more primitive approaches (which, annoyingly, have a solid track record). On the other hand, SAGE rocks, and uses extremely deep voodoo; I'm not being a Luddite here as a moral position - if your cool stuff actually works then you win. If it gets you conference presentations, finds bugs in vuln_example.o and vanishes into the pages of history then you fail.

Fails #3 though #6 are subtitled "Hax0r Fail" (should probably be pH41L or something, actually). The fundamental message is that I feel like "I'm a hacker, I don't need to write good code" is an attitude that's far more common in security tools than it should be. If we're going to be arrogant enough to mock all of academia for being "out of touch with real problems", let's, at the absolute LEAST, make sure our own stuff even runs before we release it. Hell, why not go nuts and make sure it does what we say?

I draw the line at good documentation though. That's going too far.

#3 - Reliability

Trying to break someone else's code with your own, broken, code is not productive. Instrumenting another binary that is (hopefully) about to fail in unexpected ways requires forethought, planning, testing and care. Investing none of those four things may not result in a fail, but the odds are certainly not on your side. Missing crashes because your instrumentation was unreliable sucks. Restarting week-long fuzz runs from scratch because your framework just crapped its pants sucks. In short, code that sucks, sucks.

#4 - Corruption

So corruption is the whole idea right? How could corruption lead to fail? Simply put, unless your tests are exactly as corrupt as you expect then you are knocking on Mr Fail's door and he isn't buying cookies. Assertions are not for 'real developers' nor are they a 'waste of cycles'. Imagine millions of tiny fuzztests, swimming upstream through the code paths, their little tails wiggling, earnestly searching for the exploitable ovum... but you forgot to add a check to make sure that your corruptor didn't change the structure size, and it turns out your corruptor code is buggy. Bam! Cervical Length Check! ... aaand you fail.

#5 - Doing It Wrong

The road to this fail is paved with phrases like '...for now' and '...as a proof of concept'. You want to add a database for some reason, so just to get it up and running, you think you may as well use sqlite to test, and then work out how to use postgresql later (me). You can't find a good dbgeng wrapper for Ruby, so you just take CDB and talk to it via IO using CreatePipe (also me). You don't want to work out how to use something like 0mq, so you write a quick protocol, for now, using JSON and Netstrings (me, again). If you like rewriting your own code, over and over, then this style of approach is great. If you like selling bugs it is not.

#6 - 0x41

This was actually an attempt to insert some content that might actually be useful, and I ran over some ideas for things to insert or modify that aren't 'A'. However instead of said useful content, I will relate an amusing anecdote. Metlstorm, of Kiwicon (and other) fame and I were sitting in the back row of a conference somewhere, watching an exploitation talk, when he came up with the idea of mapping ascii goatse at 0x41414141 (huh, why isn't that a read AV? Let me just... AUGH!). It is an awesome idea and I recommend all software vendors do so. For people who write fuzzers, I instead recommend that you read or think about more imaginative malicious input.

#7 - Overexcitement

You found a bug! Whatever you do, don't try and repro it, or test any input variants to see if the crash changes! Crack open your favourite debugger and spend hours trying to trace back to the root cause to see if you can get control of eax. Tell your friends as soon as possible that you have something that 'looks exploitable'. Drop it in some conference slides (me, yet again.) Post to dailydave, full-disclosure and bugtraq about your epic fuzzer which found this bug, but obfuscate the bug itself because it is clearly of earth shattering importance. For bonus points, obfuscate it badly so that others can find it independently. Misunderstand the impact of the bug, maybe call it a DoS when it's remote system. Well done. You have mastered Fail #7.

#8 - Wasting [expletive] Time.

Your fuzzer development mantra should begin with "While this fuzzer runs, I will...". Any fuzzer at all, no matter how primitive, has a better chance of finding a bug than an idle CPU core. If you're working on your Mark II fuzzer, codenamed "G0d of |3ugz" that's great, but do it while "crapfuzz.py" is running. Protip for young players - watching porn while a fuzzer is running in another window is billable hours.

That's all I had time for, although my own personal list is considerably longer. Anyway, if I've made you think twice about an impending failure situation then I've exposed myself to ridicule to good purpose. If I've merely made you laugh at the mental image of ascii goatse in an immdbg window then I can live with that as well.

On to (slightly) more serious matters.

I was uncharitably mean to some perfectly nice Finns a while ago, and at that time I promised that we (being The Grugq and I) would 'put up or shut up' to employ the vernacular. Behold! Metrics.

I traced 85021 MS Word files recently; it took a bit over a week, with one trace completing about every 9 seconds, on average. I used 64 cores for the tracing and another 4 for the control and analysis. The full trace output database is around 540MB (the traces are compressed, obviously). The iteratively reduced set, which was created as the tracing was in progress, contains 3229 files. Once the full tracing process was finished, a further 'greedy' reduction was applied, and the approximate minimum set is 2401 files. That set covers 496457 edges that are at least partially within mso.dll, wwlib.dll and winword.exe. Without reproducing the full graph, at 26k traces I already had 471217 edges in 1753 files.

For more information about the nuts and bolts, the 'wiggly curve' graph of the rolling average of edges added per trace, and a tedious discussion of the merits (or otherwise) of the greedy set cover algorithm, the materials are up at the Ruxcon website here: http://www.ruxcon.org.au/archive/2010-materials/.

To download Prospector, go here: http://www.coseinc.com/en/index.php?rt=downloads

You could also use the unstable code in the repos (URLs in the slides), but that would be unwise. We provide a precompiled Pintool for the code coverage, for those that are too lazy to get a Windows build environment working. It's not backdoored in any way. Who would even do such a thing? You should definitely use it, and get into the habit of running untrusted binaries from security researchers with questionable senses of humour.

Demonstrating my enthusiastic and ongoing commitment to Fail #5, I note that the documentation is poor and the dialog clicky clicky script in the instrumentation harness is more tightly coupled to Word than I'd like.

Man, that was long. Maybe CMU will give me a Masters...

Cheers,

ben