CERT/CC Blog

Vulnerability Insights

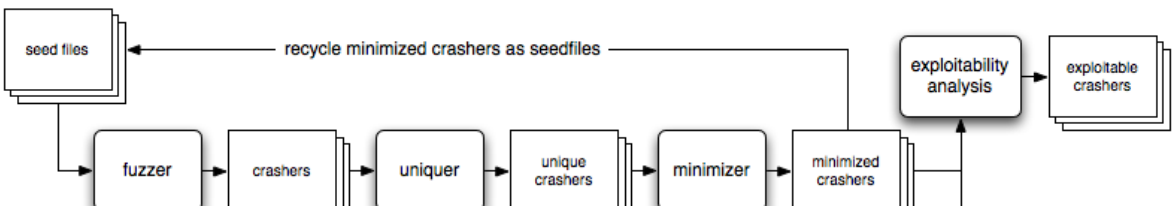
Attaching the Rocket to the Chainsaw - Behind the Scenes of BFF and FOE's Crash Recycler

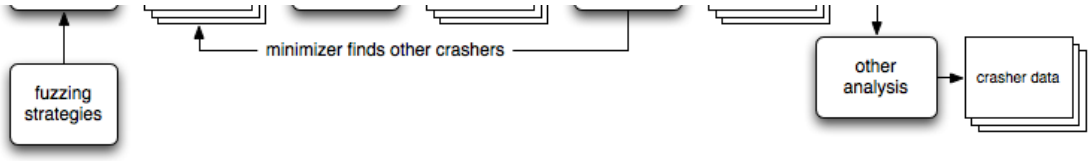
POSTED ON SEPTEMBER 30, 2013 BY ALLEN HOUSEHOLDER [/AUTHOR/ALLEN-HOUSEHOLDER] IN RESEARCH [HTTPS://INSIGHTS.SEI.CMU.EDU/CERT/RESEARCH/]

Hi folks, Allen Householder here. As Will Dormann's earlier post [http://www.cert.org/blogs/certcc/2013/09/one_weird_trick_for_finding_mo.html] mentioned, we have recently released the CERT Basic Fuzzing Framework (BFF) v2.7 [http://www.cert.org/download/bff/] and the CERT Failure Observation Engine (FOE) v2.1 [http://www.cert.org/download/foe/]. To me, one of the most interesting additions was the crash recycling feature. In this post, I will take a closer look at this feature and explain why I think it's so interesting.

(Hat tips to John Regehr for prompting the discussion [http://blog.regehr.org/archives/1042] and reddit user evilcazz for inspiring [http://www.reddit.com/r/netsec/comments/1mz3e9/one_weird_trick_to_finding_more_crashes_cert/cce6tjf] the title of this post.)

The idea behind crash recycling has been on our to-do list for a couple of years now. In fact, one of the earliest architecture sketches I drew for BFF 2.0 included it. Other features took precedent in earlier releases though, so it was just an arrow on a diagram up until a few months ago.





Meanwhile, in related research I had come across Adaptive Random Testing [http://www.utdallas.edu/%7Eewong/SYSM-6310/03-Lecture/02-ART-paper-01.pdf] (ART) by Chen, Leung, and Mak. ART attempts to "distribute test cases more evenly within the input space." This prompted me to think about what mutational file format fuzzers like BFF and FOE are doing in terms of exploring the input space of a program.

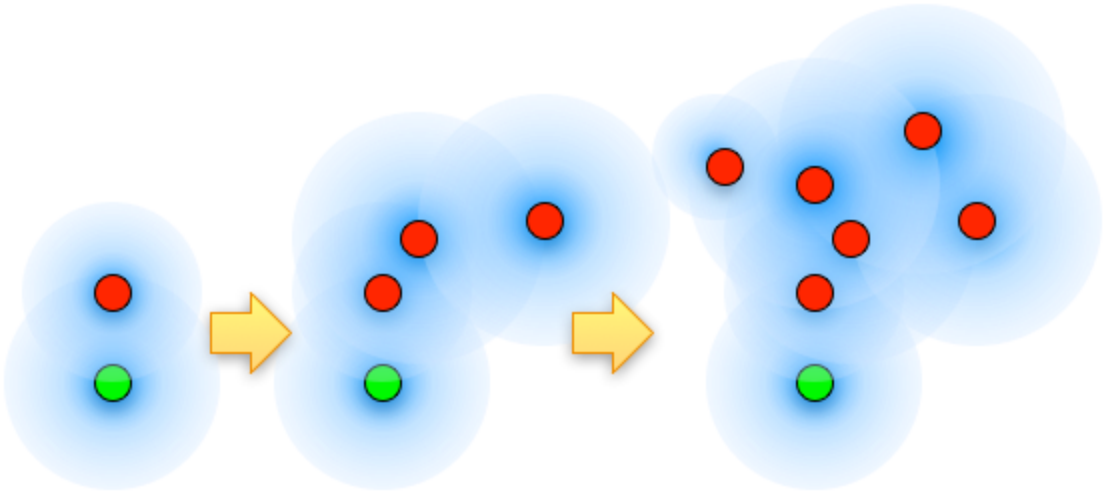
I tend to think about these things in a visual way, so before I go further, it might help to set the scene. A simple way of thinking about a file as input to a program is as a string of length N bytes. Simple bytewise mutational fuzzing like that implemented by FOE's ByteMut fuzzer takes a seed file and mutates some number of bytes M, which by definition is less than or equal to N. If you're familiar with information theory, you might notice that M represents the bytewise Hamming Distance [http://en.wikipedia.org/wiki/Hamming_distance] between the seed file we started with and the fuzzed file.

If we restrict ourselves to considering only the possible fuzzed files reachable by bytewise fuzzing from a starting input file of length N, we are in effect talking about an N-dimensional Hamming Space. But because file formats require some degree of structure for the program to recognize the input as sufficiently valid to process, larger Hamming Distances are typically less likely to produce useful results since they mangle the file too much.

For this reason, BFF and FOE include a feature we named rangefinder [http://www.cert.org/blogs/certcc/2011/02/cert_basic_fuzzing_framework_b.html], which in our current model essentially describes a probability function on the Hamming Radius around a given seed file. Our rangefinder empirically estimates an optimal Hamming Radius for each seed file based on the number of crashes found, but in general it prefers to explore Hamming Radii in the vicinity of the seed files. The image at left shows this conceptually.

BFF 2.6 and FOE 2.0.1 and earlier were basically just exploring the Hamming Radius in the immediate vicinity of their seed files. Thus, all the crashing test cases they could find would be relatively near the seed files in the Hamming space.

But then we wondered about something. What if we started exploring the input space around the crashing test cases we found by fuzzing the crashers? Well, it turns out that it works. Really well.



For example, the following table depicts a comparison of the results of eight FOE instances running for approximately a day, fuzzing the version of ImageMagick that we include with the installer for demonstration purposes. Four machines had crash recycling enabled, four did not. The numbers in the table show the number of unique crashes found across all four machines in each cohort.

	Without Crash Recycling	**With Crash Recycling**
Exploitable	233	2,203
Probably Exploitable	1	0
Probably Not Exploitable	47	51
Unknown	648	718
Total	929	2,972

But are these really unique bugs? Probably not. FOE uses Microsoft's !exploitable Crash Analyzer **[http://msecdbg.codeplex.com/]** to produce unique bug IDs. !exploitable generates a major and a minor hash for each crash based on a fuzzy stack backtrace hash. The counts above reflect the concatenated major.minor hash tallies. However, even when we only consider the major hash of the first exception encountered (which may be more closely connected to the underlying bug), we still have 61 unique hashes with crash recycling and 49 without. The table also provides anecdotal evidence that fuzzing crashers may increase the yield of exploitable test cases.

So Why Does This Work?

We don't really know for sure yet, it's an area of active research. But here's my take on it so far:

Selecting seed input is important to mutational fuzzing since you want to maximize your exploration of the

program's behaviors -- this is similar to what the ART process is trying to achieve. The key realization we made was that crashing test cases crash because they're inherently increasing code coverage of the program in some way. The fuzzed file zigs where the seed file zags, and so the fuzzed file behaves differently. But this is exactly the kind of criteria that would lead you to accept a new seed file: it exhibits behavior you hadn't previously seen in the program.

So recycling fuzzed crashers into the seed file set means you're adding files that increase code coverage. And increased code coverage opens up the doors to new branches, which can enable you to increase coverage even more.

The apparent increase in exploitability is even less well understood. It seems plausible that given an underlying bug that can trigger a range of different behaviors, your first encounter of the bug may not trigger the most exploitable behavior. So fuzzing the crashing test case is essentially exploring the behavior space nearby and is thus likely to land on one of the more exploitable examples of the bug.

Crash recycling may also exacerbate the limitations of using stack backtrace hashing to determine crash uniqueness. We have already encountered one such limit (stack corruption) which resulted in the addition of a PIN trace tool to BFF 2.7 [http://www.cert.org/blogs/certcc/2013/09/one_weird_trick_for_finding_mo.html]. It's possible that stack backtraces are insufficiently correlated with underlying bugs to use them for fine-grained screening of a large volume of test cases, even though we still think they make a good coarse-grained sieve for the potentially huge volume of test cases you'd have to deal with otherwise.

I'll yield the last word to Will Dormann, quoting his comment [http://blog.regehr.org/archives/1042#comment-12400] on John Regehr's blog (slightly edited for format/relevance):

> *However, there are important things to consider: (1) If I have 10, 100, 1,000,000 crashing test cases that turn out to be the same underlying bug, does it really matter if one of them just happens to be have the "home run" crash properties that demonstrate exploitability? Or if as a developer, I fix that one bug and the 1,000,000 cases now all go away when I do my verify run? (2) There really may be actual new underlying bugs uncovered by crasher recycling and fuzzing that quasi-valid file.*
>
> *But both points indicate the need for further investigation and testing in both of those areas. In the case of (1), it demonstrates perhaps the need for better uniqueness determination and also the need for the ability to handle huge piles of crashers that may or may not be from unique underlying bugs. In the case of (2), it demonstrates the need to perform tests to show whether or not any of the uniquely crashing test cases that were derived from fuzzing quasi-valid (crashing) files are indeed new underlying bugs.*
>
> *Both of these are definitely on our radar...and there's a reason that the feature is turned off by default. Neither security researchers nor software vendors seem to be prepared to handle that sort of volume.*

About the Author

Allen Householder

Contact Allen Householder [https://www.sei.cmu.edu/contact.cfm]
Visit the SEI Digital Library for other publications by Allen [https://resources.sei.cmu.edu/library/author.cfm?authorID=4483]
View other blog posts by Allen Householder [/author/allen-householder]