Fuzzing: Breaking software in an automated fashion

Ilja van Sprundel

December 8, 2005

1 Introduction

Fuzzing is the art of automatic bug finding. This is done by providing an application with semi-valid input. The input should in most cases be good enough so applications will assume it's valid input, but at the same time be broken enough so that parsing done on this input will fail. Such failing can lead to unexpected results such as crashes, information leaks, delays, etc.

It can be seen as part of quality assurance, although only with negative test cases. Fuzzing is mostly used to uncover security bugs, however, it can often also be used to spot bugs that aren't security critical but which can non-the-less improve robustness.

2 Types of fuzzers

While fuzzing is mostly done in an entirely automated fashion, it is also possible to perform semi-automated fuzzing. This usually involves making a small tool, doing one test run and then carefully examine the response. The benefit of semi-automated fuzzing is that very subtle bugs -that would otherwise not get noticed- can be found. If need be the code used for semi-automated fuzzing can be changed

Manual testing can sometimes also be though of as a type of fuzzing. In most cases it is the preparation needed to perform automated fuzzing. With manual tests it becomes obvious which parts of a program or a protocol are the most interesting for fuzzing. Sometimes critical bugs are even found during manual testing.

Obviously tools are needed to conduct fuzz testing. There are 2 types of fuzzing tools, standalone tools which were designed to fuzz a single program

or protocol and fuzzing frameworks. Fuzzing frameworks have an api that can be used to easily create broken data and implement specific protocols.

While most fuzzers are build to test networking protocols, it's possible to fuzz a whole lot more then just network protocols. Files, api's, arguments for a command line utility, standard input, signals, and many more can all be fuzzed. Any point where there is some communication with an application can potentially be fuzzed.

3 How fuzzing works

When building a fuzzing tool there are 2 common approaches. The first one is to randomly send some kind of data in an endless loop. this random fuzzing has the potential to uncover a lot of bugs but often misses quite a few because an application parsing the data might consider it to be invalid before it reached a faulty piece of code. In most cases this can be worked around by implementing adleast some intelligence into these kind of fuzzing tools.

The second one is where it has been carefully studied what will likely cause problems and iterate over all possible combinations thereof, it comes close to fault injection. This kind of fuzzing is usually finite. One problem here is that it is almost always impossible to generate all possible combinations that will trigger a bug and often some bugs get missed.

In reality random fuzzing usually finds the first couple of bugs faster. The second type of fuzzing is often more complete. However, sometimes random fuzzing will uncover bugs that would have never been found with the 2nd type of fuzzing, because it is never ending and uses a random source for most of it's input. It's not uncommon to still discover a bug with random fuzzing after several hours or several days.

The kind of intelligence that is put into a random fuzzer usually depends on the amount of effort that has been put into it. More intelligent fuzzing usually leads to more results, but requires more time developing the fuzzing tool.

4 Determining completeness and failing

Determining Completeness when fuzzing is usually very hard to do, more often then not when performing fuzzing some or all of the documentation is not available and most information has been gained through reverse engineering. Even when standards are available variations on the standards could have

been implemented and new (undocumented) features might be introduced.

The configuration of whatever application that gets tested plays a big part in completeness as well. What's not configured can't be tested.

While fuzzing it's important to determine when a program failed, which isn't always easy. failing usually can be determined when the program hangs, crashes (sigsegv), reboots or consumes huge amounts of memory. Crashes can be detected by attaching a debugger to the application being fuzzed. Hangs can possibly be detected by means of timing. Huge memory consumption can be detected with memory statistics tools. Keeping track of specific logfiles might also help to determine weather an application failed or not.

5 Key elements in fuzzing

Some interesting things to look at while developing a fuzzing tool are any kind of size field, strings, something that marks the beginning of a string or a binary piece of data and something that marks it's ending.

For size fields it's always a good idea to use values around the boundaries of the size field's type. For example, if a size is seen as an unsigned 32 bit value giving it a value of 0xffffffff might cause an integer overflow:

```
p = malloc(yourlength + 2);
strlcpy(p, yourstring, yourlength);
```

Negative values often lead to problems, in a lot of cases they get missed when bound checking is performed:

```
if (yourlen > MAX_BUFSIZE) return SOME_ERROR;
else memcpy(buf, yourdata, yourlen);
```

Negative values can also lead to underindexing after the same kind of flawed bounds checking.

Sometimes applications will assume that the length given is exactly that of a string passed to it and will happily copy that string into a buffer after a bounds check is passed:

```
read(fd, &yourlen, sizeof(yourlen));
if (yourlen > MAX_BUFSIZE) return SOME_ERROR;
else strcpy(buf, yourstring);
```

Using random numbers sometimes might trigger a bug, it's impossible to tell where the programmer of an application messed up until it's been tested:

```
#define SOME_MAX 4096
if (yourlen < 0 || yourlen > SOME_MAX) return SOME_ERROR;
p = malloc( ((yourlen * 2) * yourlen) * sizeof(very_large_struct));
for (i = 0; i < yourlen; i++) p[i]->elem = some_number;
```

String handling has caused many software bugs in the past and hence it would probably be benificial to take advantage of this. Obviously it's always a good idea to try very long strings, since those might cause trivial buffer overflows.

Including formatstrings such as "%n%n%n%n" in strings while fuzzing might also result in bugs being found.

Binary data inside strings sometimes leads to surprising bugs. A good source for binary data can be found in /dev/urandom:

```
a = malloc(strlen(b) +1);
while(*b != 'b' && *b) b++;
b++;
strcpy(a,b);
```

Using empty strings might also trigger some bugs. Sometimes, although very protocol specific, there are length fields inside strings. The previously mentioned interesting size field comments apply here as well.

Using sql statements in strings quite often leads to sql injection bugs, simularly putting shell escape codes in your strings might lead to code execution.

Pieces of data that mark some beginning or ending are usually good candidates for fuzzing as well (",],',NULL,...). Don't use them in some test runs, use them twice in other test runs, escape them, put data after them anyway, ...

All of the things described in this chapter are mostly used in data generation, Where everything gets generated by the fuzzing tool itself. Sometimes it's also useful to take a valid piece of data and then change it somehow, otherwise known as data mutation[1].

6 Annoyances while fuzzing

When fuzzing there are several things that make fuzzing harder then it looks. A very common problem is that of a bug behind a bug, where it is almost

[1] see http://ilja.netric.org/files/fuzzers/mangle.c for a simple example

impossible to trigger one bug because another one is in the way. Often the only solution is to fix that bug and start fuzzing again.

"Userfriendlyness" can tremendously slow down, or even halt automated fuzzing. One solution to this problem is to preload some library and get rid of whatever is slowing down fuzzing.

Slow programs are also annoying, often this is because the program is badly written, There is no way to fix this problem while fuzzing, but it is usually an indication that many things are wrong with that application and fuzzing it will likely turn out to be very sucessful.

Checksums, encryption and compressions are often annoying because they simply need to be implemented in the fuzzing tool and increase development time of a fuzzing tool significantly.

Memory leaks can also get in the way of fuzzing. Arguably this is a case of a bug behind a bug. The problem is that they might not get noticed until they slow down fuzzing.

Last but not least "undefined states" are very annoying. These undefined states are usually triggered by test run $x - n$ but only discovered in test run x. They are often very hard to track down.

7 Conclusion

This paper has covered all the essentials to automated bug finding. It has detailed how fuzzers work and how to build them yourself. It described the types of fuzzing and annoyances that come with automated bug finding. The thing that this paper has unfortunatly not delivered is the actual thrill of fuzzing, since it is something you have to experience and cannot just get from reading about it.