Fuzzing

A Useful Approach to Finding Bugs

Table Of Contents

0x00000000(Fuzzing)

No software is perfect. That is why programs have bugs. Bug finders, code auditors, security researchers: they all have the same goal. They want to exploit imperfections.

Fuzzing is art of filling input with specified or random data. The goal of fuzzing is to find bugs in software. Fuzzers are automated programs that “fuzz” target(s). Some bugs are exploitable so you can execute commands and/or shellcode on the target. Some bugs just make the program crash or fail to operate properly.

Program A says you can tell it your name.
B0b tells it his name is “b0b%n%n%n%n%n”.
Program A snprintf()'s “name” into its buffer without specifying a format string.
Program A crashes.

B0b laughs. B0b puts on his evil grin. He has work to do. If B0b is auditing enterprise software, he may be able to afford that new phone and unlimited txt soon. B0b could even take his woman to Europe with him. B0b is such a romantic.

If you understand what programming bugs are, what exploits do, and even how to find bugs, you should comprehend fuzzing. If not, take a page from OWASP:

“*Fuzz testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion. A fuzzer is a program which injects automatically semi-random data into a program/stack and detect bugs.*

The purpose of fuzzing relies on the assumption that there are bugs within every program, which are waiting to be discovered. Therefore, a systematical approach should find them sooner or later.” -- [*http://www.owasp.org/index.php/Fuzzing*](http://www.owasp.org/index.php/Fuzzing)

0x00000001(Guidelines)

There are guidelines that should be followed when fuzzing and writing fuzzers. These have not been vetted and are not standardized. Do not quote them to be. These are just some good rules when building fuzzers.

Make sure what ever you are fuzzing (protocol, file format, other input) is implemented as properly as you can. Specifications should be followed as closely as possible and aim for good compliance and perfection (but don't be disappointed if all is not achieved, you'll do fine). Some RFC's are good, and some aren't much help at all.

Make sure the code is portable. You want many people to be able to understand how to run and use the fuzzer in different environments and situations. Try to code for the best of all people while also making it easy to add new fuzzing techniques and objects.

Make the fuzzer as automated as possible. Good etiquette is to write your code so that once all the necessary options are set, the fuzzer is good to go and we can go to work without it needing any further attention. It is fantastic to come back and see the target box annihilated and your fuzzer explain to you what happened ;)

Consider this principle: If the program crashes, it fails our fuzzing test. If the program does what we tell it to do (not within the bounds of normal operations), it also fails. If not, it passes. We want it to fail :)

Those are just a few semi-common-knowledge guidelines that should be followed when fuzzing. Take them to the head.

0x00000002(Building A Fuzzer)

When writing a fuzzer, there are steps that need to be taken for proper implementation. You have to walk before you can run, etc, etc.

(0x00) **Choose your target and identify points of fuzzing.** Basically, you need to figure out exactly what program, server, or otherwise code your going to be fuzzing and the points of input.

Search for information about the target.
Find out what protocols it implements and look at code that uses its functions (servers, clients, exploits).
Look for exploits for previous (or current :) versions of the target.
Run the target on an available environment and learn about it. Play with it. Test it out.
Research the points of input it the target supports and its possibly fuzzing areas.

(0x01) **Prepare Fuzzing Strings with Specifications.** Define your fuzz strings and protocol (use interchangeably) specifications, then put them together in a buffer to use for fuzzing.

pop3[] = USER, PASS, RETR, DELE, UIDL;
fuzz[] = A x 500, A x 2000, A x 10000, %n%n%n%n%n, -1, 65536, {}[]/\/\, blah;

You want to put all commands, variables, or otherwise input vectors into your arsenal to be as effective in fuzzing all areas that possibly contain bugs. And you defiantly want your fuzz strings (or otherwise) to be as clever and robust as possible. Nobody wants to miss a bug because they left out a fuzz string that they didn't think merited implementation. Be creative but also be smart, you don't want to waste your time either.

(0x02) **Fuzz your target.** If you got your fuzzing data generated in relation to the specifications set forth by the target or the protocol the target supports or reads from, take down the house.

```
fuzzpop3 = pop3[1] + fuzz[1];
connect(target); send(fuzzpop3);
```

Simple and somewhat effective. Checking if the target is down (and also incursive error checking), timeout settings, debugging, and logging play a role in fuzzing your target. Information is a key component in fuzzing, and we want to know every little step our fuzzer makes and every limb our fuzzer breaks ;)

(0x03) **Check for exceptions and out-of-bounds breakage.** The reason we fuzz is to find bugs, so monitoring for exceptions is part of the job. Let the fuzzer run, but if and when the target dies, check to see what happened. Logging is very helpful here as well. Also check for signs of command execution or trails thereof when fuzzing with strings that try to break the program out of normal operations, or out-of-bounds breakage.

Using fuzz string “fuzzer;touch /tmp/fuzzed;fuzzing123”
Program insecurely passes the string to a function that executes commands as part of an operation when processing input.
“ls -al /tmp”

```
-rw-r--r--  1 root root    0 2008-10-17 21:11 fuzzed
drwxrwxrwt  4 root root 8192 2008-10-17 06:25 .
drwxr-xr-x 21 root root 4096 2008-05-08 01:00 ..
srw-rw-rw-  1 root root    0 2008-10-10 16:06 .gdm_socket
drwxrwxrwt  2 root root 4096 2008-10-10 16:06 .ICE-unix
drwxrwxrwt  2 root root 4096 2008-10-10 16:07 .X11-unix
```

(0x04) **Explore Findings.** After you review the bugs, see what you can accomplish by leveraging them. Determine if you can only crash the program or if you can run execute arbitrary commands on the target. Valuable or not, these things are sometimes fun to investigate. There's not many feelings as good as when you've been auditing a program and see that semi-magical 0x41414141 instruction pointer value. I know that feeling.

Now, that being said, I challenge you to write your own fuzzer. I have written a few myself, the same way that I have just explained to you. Feel free to take my code, use it and abuse it (nicely). Write a fuzzer, and if you like, send me some exploits :)

ComRaider is an excellent ActiveX Fuzzer by David Zimmer.
http://labs.idefense.com/software/fuzzing.php#more_comraider
http://labs.idefense.com/files/labs/releases/COMRaider_Setup.exe

Browser Fuzzer fuzzes HTML and JavaScript, plus some of DOM.
http://jbrownsec.blogspot.com/2008/10/browser-fuzzer-released.html
http://packetstormsecurity.org/fuzzer/bf10BETA.tar.gz

Zeroday Fuzzer remotely fuzzes 9 different protocols and has two generic modes.
http://jbrownsec.blogspot.com/2008/10/zeroday-fuzzer-released.html
http://packetstormsecurity.org/fuzzer/zfz20BETA.tar.gz

FileFuzz is a Windows-based file format fuzzer.
http://labs.idefense.com/software/fuzzing.php#more_filefuzz
http://labs.idefense.com/files/labs/releases/FileFuzz.zip

Zzuf is a transparent application input fuzzer that corrupts user-contributed data.
http://caca.zoy.org/wiki/zzuf
http://caca.zoy.org/files/zzuf/zzuf-0.12.tar.gz

Of course we all know of SPIKE, notSPIKEfile, AxMan, JBroFuzz, fsfuzzer, on and on and on.

0x00000003(Finding Bugs)

Fuzzing is great and you love finding bugs. Well then, lets test out my browser fuzzer on a popular web browser, such as KDE's Konqueror (www.konqueror.org):

```
$ gdb /usr/bin/konqueror
GNU gdb 6.8
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu"...
(no debugging symbols found)
(gdb) r
Starting program: /usr/bin/konqueror
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
.....

Open a page with this code in Konqueror: <html> <font color="A" x 500000> </html>

konqueror: ../../src/xcb_lock.c:89: request_length: Assertion `vec[0].iov_len >= 4'
failed.

Program received signal SIGABRT, Aborted.
[Switching to Thread 0xb66856c0 (LWP 5926)]
0xb7f65410 in __kernel_vsyscall ()
(gdb) i r
eax            0x0      0
ecx            0x1726   5926
edx            0x6      6
ebx            0x1726   5926
esp            0xbfa0c0ec     0xbfa0c0ec
ebp            0xbfa0c108     0xbfa0c108
esi            0x1726   5926
edi            0xb7d4eff4     -1210781708
eip            0xb7f65410     0xb7f65410 <__kernel_vsyscall+16>
eflags         0x200202 [ IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Fuzzed. It does not look like we'll be able to execute commands or shellcode, but it did crash. It was fuzzed. This is just an one example of fuzzing.

0x00000004(Last Thoughts)

Fuzzing will not always yield bugs that will allow you to execute code. Fuzzing does not promise to make your dreams come true. Fuzzing is an art and a software programmer's nightmare. Fuzzing will open your eyes to see that it is no longer enough to know the code backwards and forward, inside and outside, layer by layer, line by line, bit by bit.

If you have any questions, comments, or concerns, feel free to contact me.

Thank you for reading.

Jeremy Brown
0xjbrown41@gmail.com
http://jbrownsec.blogspot.com