January 19, 2018

FUZZING TCP SERVERS

—

Intro

The architectures of software fuzzing setups that authors of security fuzzing tools had originally implemented were fairly simple. In early days of security fuzzing (before 2010) the vast majority of fuzzing engines were writing mangled content to disk files, and then instructed fuzzed binaries where to find them:

```
$ honggfuzz -f INPUT_DIRECTORY -- /usr/bin/djpeg ___FILE___
```

Here, the placeholder (___FILE___ for honggfuzz, @@ for AFL) was supposed to be replaced by a name of a file holding the actual fuzzing content, and a fuzzed binary was re-executed continuously over supplied set of input files. When such fuzzed binary (`djpeg` here) crashed, the input file that was believed to have caused this crash was copied into a *crashdir* directory under some new and fancy name, e.g.:

```
SIGSEGV.PC.af0ef0.STACK.c7dc5c2c7.CODE.1.ADDR.0x8.INSTR.mov____0x8(%rcx),%rdi.fuz
```

or

```
crash-50b12d37d6968a2cd9eb3665d158d9a2fb1f6e28
```

With continuous advancement of fuzzing techniques, even more sophisticated ways of exposing code-blocks - approximating roughly the software attack surface - to *fuzzy* data were developed. Inputs, which were supposed to be parsed by a given set of APIs, were now sent to the fuzz-tested process, and from there those tested APIs were invoked in an endless loop.

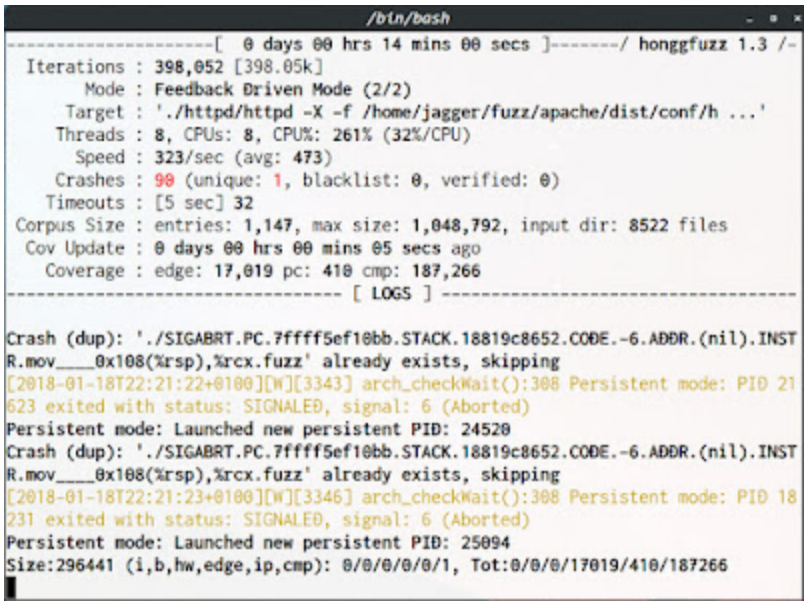
```
for (;;) {
  const uint8_t* buf;
  size_t len;

  GetDataFromTheInputPreparationStage(&buf, &len);

  /* Decode JPEG image into a bitmap */
  jpeg_mem_src(&cinfo, buf, len);
  jpeg_read_header(&cinfo, TRUE);
  jpeg_start_decompress(&cinfo);
  ...
}
```

The obvious benefit of such approach was the increased speed of execution; the chunk of CPU time previously spent on re-executing a binary from scratch (via `fork/execve` call sequences) and running expensive `ld.so` and library *init* routines was greatly reduced. With this new mode - dubbed *persistent fuzzing* - the CPU time was allocated mostly to executing target functions located within tested APIs.

The libFuzzer engine executed the fuzzing step using the so-called in-process mode, while honggfuzz and AFL engines opted into using a separate supervisor process (for one or another reason; discussion of which lies outside of the scope of this article) and were sending data to the fuzzed binary via some form of IPC (file-descriptors/paths, sockets or even via shared memory). But overall, this was not so conceptually different from how in-process fuzzing engines were performing their tasks.

All those approaches worked quite well, and over course of the last years various fuzzing engines helped to uncover hundreds, if not thousands already: logical bugs, crashes, data leaks, undefined behavior code constructs, DoSes/deadlocks and resource exhaustion cases. Many of those bugs finally turned out to be serious security vulnerabilities.

In the aftermath of systematic use of those tools by many sec-researchers around the world, each feedback-driven evolutionary fuzzer is now able to impress its existing and potential users with long list of software (and in some cases, hardware) problems uncovered with its help. Here're the trophy lists for honggfuzz, AFL and for libFuzzer.



TCP servers

Over time the process of fuzzing of well defined APIs, which were accepting pointers to data buffers and indications of their length, became much easier-to-implement and reasonably popular among software programmers. It has even become something of a convention, that authors of larger software packages started preparing fuzzing stubs (for lack of a better name) and distributed them along the main source code. For example, this OpenSSL directory contains a nice set a fairly comprehensive fuzzing stubs for a sizable chunk of functionality exposed to potential attackers by the OpenSSL code.

Unfortunately, it wasn't really the case for other type of software packages that were in bad need of security-testing. This type of software didn't take inputs as memory buffers or cmd-line arguments pointing to local disk files, but it was accepting network connections and interacting with its users over TCP/UDP/SCTP/etc. streams.

Even today, readily available examples of fuzzing setups for Apache's HTTPD, OpenSSH or for ISC BIND are fairly uncommon. For instance, this publicly available set of OpenSSH fuzzers tests only a small subset of interesting functionality exposed by this huge and important software package.

When taking widely-used TCP servers into consideration, as far as I know, there exist only quite *hacky* (nonetheless, working) setups prepared by author of this article (for Apache HTTPD and for ISC BIND) tailored for use with honggfuzz [the Apache HTTPD fuzzing setup was later adopted for use with AFL by Javier Jiménez].

Having said that, it's quite possible that alternative, sometimes non-public fuzzing setups continuously

looking for bugs in TCP servers exist and are in active use.

Honggfuzz NetDriver

But, if we really wanted to implement a fairly robust fuzzing setup for TCP servers, what would we really need in order to achieve this?

- A piece of code which can integrate with existing TCP servers, e.g. a static/dynamic library which can be linked together with Apache's HTTP (i.e. its `httpd` binary),
- A technical method for converting inputs from memory arrays and lenght identifiers into TCP streams,
- A way to indicate to TCP servers, that they shouldn't expect any more data over the TCP stream if there's none left in the fuzzing input buffer; otherwise TCP servers could hang forever, or terminate connections after not-so-good-from-fuzzing-perspective timeouts (say, more than 10 ms.),
- Some solution to the problem, that if many identical TCP servers ran at once on the same system, with same configs (or, with default configs), then they would also bind to numerically identical TCP ports. Of course, this only matters if it's our intention to fuzz multiple instances of the same TCP server at once (what is easily doable with honggfuzz).

The Honggfuzz NetDriver, introduced recently into the Honggfuzz's code-base, tries to deliver just that.

It is compilable to a static library [`libfhnetdriver/libhfnetdriver.a`]. Typing ...

```
make libhfnetdriver/libhfnetdriver.a
```

... inside the honggfuzz's source code directory will make it ready to use. If you are interested in implementation details, the source code for the driver can be found here.

The driver inserts itself as a target (provides symbols) for the interface implemented originally by the libFuzzer's authors:

```
int LLVMFuzzerTestOneInput(const uint8_t* buf, size_t len);
```

This choice makes it possible to use the driver not only with honggfuzz, but also with AFL and, obviously, with libFuzzer. The only piece of modification the Apache HTTPD project will need from now on boils down to the following diff:

```
--- a/server/main.c
+++ b/server/main.c
@@ -484,8 +484,11 @@ static void usage(process_rec *process)
destroy_and_exit_process(process, 1);
}

-int main(int argc, const char * const argv[])
-{
+#ifdef HFND_FUZZING_ENTRY_FUNCTION
+ HFND_FUZZING_ENTRY_FUNCTION(int argc, const char *const *argv) {
+#else
+ int main(int argc, const char *const *argv) {
+#endif
char c;
int showcompile = 0, showdirectives = 0;
const char *confname = SERVER_CONFIG_FILE;
```

After the patch has been applied, the **`main()`** function residing inside Apache HTTPD's `server/main.c` will be replaced with a custom macro (expanded by `hfuzz-cc/*` compilers). This expanded macro has a special meaning to the Honggfuzz NetDriver - it will recognize it as a location of the original entry point to the Apache HTTPD server code.

After linking code of a TCP server (i.e. `httpd` here) with the `libhfnetdriver.a` (this step is performed automatically by the `hfuzz-cc/*` compiler wrappers) our chosen fuzzer engine (honggfuzz, libFuzzer or AFL) will run its own **`main()`** function first, and then it will run TCP server code inside a separate thread.

The aforementioned problem of signaling end-of-input to TCP servers (i.e. no-more-data-in-the-fuzzing-input-buffer state) can be solved by sending the TCP FIN packet along the established TCP stream. With modern operating systems, this can be done using the shutdown(sock, SHUT_WR) syscall.

The last requirement on our list of problems to solve, is the ability to start multiple TCP servers using same TCP ports. Unfortunately, the fairly new `setsockopt(SO_REUSEPORT)` option won't be of use here, as mangled inputs would be delivered to random instances of fuzzed processes. But, there's one good alternative here ...

... this *marvelous* solution to our problem are the Linux network namespaces. As their name suggests, they will work with Linux OS only, so users of other operating systems are unfortunately out of luck here. Running each new TCP server in a separate net namespace allows it to bind to any TCP port it wishes so, as each of those servers will see its own and brand new loopback interface. The NetDriver utilizes Linux namespace support code which can be found inside the Honggfuz's libhfcommon library.

Once the TCP server, now running inside its own software thread, is up and accepting new TCP connections, our net driver will:

1. be called as the **`LLVMFuzzerTestOneInput()`** interface by the input preparation stage of your chosen fuzzer,
2. **`connect()`** to the TCP server,
3. **`send()`** input (buffer) over established TCP connection,
4. use **`shutdown(sock, SHUT_WR)`** to indicate to the TCP server that there's no more data that can be delivered,
5. wait with **`recv()`** for the data that the TCP server wants us to receive, up to the point when it closes its side of the TCP connection,
6. **`close()`** the client's TCP connection endpoint,
7. jump to 1. and repeat.

The following set of commands should help you to start fuzzing your first project with the NetDriver (at least, for Apache HTTPD).

```
$ ( cd httpd && ./hfuzz.compile_and_install.asan.sh )
$ honggfuzz -v -Q -f IN/ -w ./httpd.wordlist -- ./httpd/httpd -X -f /home/jagger/
...
...
Honggfuzz Net Driver (pid=21726): Waiting for the TCP server process to start acc
Honggfuzz Net Driver (pid=21726): The TCP server process is ready to accept conne
Size:9378 (i,b,hw,edge,ip,cmp): 0/0/0/4643/110/56364, Tot:0/0/0/4643/110/56364
Size:40712 (i,b,hw,edge,ip,cmp): 0/0/0/1971/104/21274, Tot:0/0/0/6614/214/77638
...
```

A custom compilation script, together with all necessary patches, configs and initial corpus files for the Apache HTTPD server can be found inside this honggfuzz directory.

I suppose that it should be a relatively easy task to integrate the NetDriver with both libFuzzer and with AFL fuzzing setups.

Closing thoughts

One could argue that testing small, well-defined sub-components (APIs) of huge TCP servers is the best way to go here. Unfortunately covering with fuzzing routines every parsing library and routine exposed to external users inside behemoths of a size of Apache HTTPD or ISC BIND would be a long and tiresome task. What is more, there's no guarantee of achieving 100% code coverage here, as identifying every piece of code which can touch/parse user-supplied HTTP headers, POST content, HTTP/2 streams, or data passed to CGI scripts can easily take weeks, if not months.

Even if I personally support the very idea of testing small and well-specified APIs first, the fact that that those fuzzing setups for TCP servers don't exist yet, or are fairly incomplete in terms of covered attack surface, makes such end-to-end fuzz-testing quite valuable tool. It has already proven useful, yielding interesting results in a form of discovered, reported and fixed security vulnerabilities. You can read more about them here: Apache HTTPD [1, 2, 3], OpenSSH [1].

If you have any thoughts or comments about this article, you can reach me at robert@swiecki.net or via my Twitter account: @robertswiecki.

Robert Swiecki

VISIT PROFILE

Archive

Report Abuse