OpenSIPS security audit

results and next steps

Introduction

A security audit

- we focused on what we had in scope
- resulted in 13 security-relevant findings (i.e. vulnerabilities)
- wrote a number of tools in the process

The real value of our work

- immediate value: can be derived from the findings
- long term value: future manual tests and security testing automation
- long term aspect needs ongoing development but gives real ROI

Agenda: what's done

- describe what we did, our methodology
- delve into the work that is especially relevant to you
  - work done with libfuzzer, AFL and friends
  - code coverage and patterns
  - the framework that we built

Agenda: what's left

- there are parts that we did not:
  - manage to fully cover
  - cover at all
- we give you a clear picture of our coverage
- what we'd like to see in the remaining areas

Agenda: what's next

- our ideas of how to move on
- take your input

What's done

Methodology

the how and why

Setup

Target version: 3.2.2, commit `f380dc59eb79ec2153d5b0d6e8374c0877fca525`

Created multiple Docker images with the following configurations:

- Vanilla build (`make all`)
- Compiled with AddressSanitizer (`-fsanitize=address`)
- With optimization turned off (`DEFS+= -DCC_O0`)

Example:

```
server_type="vanilla" ./run.sh corefunctions/add_local_rport
```

Blackbox fuzzing

- made use of SIPVicious PRO fuzzing tool
- automated the testing procedure

Blackbox fuzzing: configuration example

```
route {
    if (is_method("REGISTER")) {
        if (!www_authorize("PRIVATE_IP", "subscriber")) {
                www_challenge("PRIVATE_IP");
        } else {
            send_reply(200,"OK");
        }
    }
    exit;
}
```

Blackbox fuzzing: core functions

- **Core functions**: `add_local_rport`, `append_branch`, `force_rport`, `setdsturi`, `sethost`, `sethostport`, `setport`, `seturi`, `setuser`, `setuserpass`, `strip_tail`
- **auth**: `www_authorize`, `www_challenge`

Blackbox fuzzing: sipmsgops functions

- **sipmsgops**: `add_body_part, append_hf, append_time, append_urihf, codec_delete_except_re, codec_delete_re, codec_delete, codec_exists_re, codec_exists, codec_move_down_re, codec_move_down, codec_move_up_re, codec_move_up, has_body_part, has_totag, insert_hf, is_audio_on_hold, is_method, is_privacy, is_uri_user_e164, list_hdr_add_option, list_hdr_has_option, list_hdr_remove_option, remove_body_part, remove_hf_glob, remove_hf_re, remove_hf, ruri_add_param, ruri_del_param, ruri_has_param, ruri_tel2sip, sipmsg_validate, stream_delete, stream_exists`

Blackbox fuzzing: sl & tm functions

- **sl**: `reply_error, send_reply`
- **tm**: `t_add_cancel_reason, t_add_hdrs, t_anycast_replicate, t_check_status, t_check_trans, t_flush_flags, t_inject_branches, t_local_replied, t_new_request, t_newtran, t_on_branch, t_on_failure, t_relay, t_replicate, t_reply_with_body, t_reply, r_wait_for_newbranches, t_wait_no_more_branches, t_was_cancelled`

Blackbox fuzzing: topology hiding

- **topology hiding**: `topology_hiding`, `topology_hiding_match`

Coverage-guided fuzzing

- Based on LLVM12 and AFL
- Built a framework based on our toolset
- Tool can be extended to include new tools
- Code can be ported to `oss-fuzz`

Coverage-guided fuzzing: function selection

- Easy functions
  - `int parse_msg(char* buf, unsigned int len, struct sip_msg* msg)`
- Complex functions
  - `auth_result_t pre_auth(struct sip_msg* _m, str* _realm, hdr_types_t _hftype, struct hdr_field** _h)`
- Hard to test (but not impossible)
  - `static int w_t_cancel_branch(struct sip_msg *msg, void *sflags)`

Coverage-guided fuzzing: coverage analysis

- Manual review of coverage reports generated by llvm tools
- Built tools to analyze and group coverage reports
- Built tools that helped in identifying uncovered critical parts of the code
- When missing coverage was observed, new initial corpus payloads and dictionary entries were added

Coverage-guided fuzzing: why does it take so long?

- Selecting the correct corpus and dictionary
- Fix bugs in the fuzzer itself, get better by time while we get accustomed to the codebase
- Fuzzing itself takes time (and lots of CPU power)
- Analysis of the coverage reports is time consuming

What happened after a crash was discovered?

- Reproduce the issue in the vanilla build of OpenSIPS
- Understand why the crash happened and debug in `gdb`
- Draft a report that can be sent to OpenSIPS developers over Slack
- Assess the criticality of the crash

What was covered?

- parser (black-box, coverage-guided)
  - `parser/digest`
  - `parser/contact`
  - `parser/sdp`
- auth module (black-box, coverage-guided)
- tm module (mostly black-box, less coverage-guided)
- sl module (mostly black-box, less coverage-guided)
- dialog module (mostly black-box, less coverage-guided)
- rr module (black-box, coverage-guided)
- topology hiding (black-box, coverage-guided)
- OpenSIPS management interface (coverage-guided)

Misc.

- Usage of SQL, code review and debugging
- Manual code review
- Look at diff of commits which had security implications

findings

the vulnerabilities

```
- Segmentation fault due to invalid Content-Length (black-box)
- Crash when specially crafted REGISTER message is challenged for authentication (black-box)
- Buffer overflow in sipmsgops SDP line deletion function leads to DoS or undefined behaviour (black-box)
- Buffer overflow in the SIP parser leads to DoS or undefined behaviour (coverage-guided)
- Buffer overflow in the SDP attribute parser leads to DoS or undefined behaviour (coverage-guided)
- Buffer overflow in SDP rtpmap parser leads to DoS or undefined behaviour (coverage-guided)
- Buffer overflow in SDP fmtp parser leads to DoS or undefined behaviour (coverage-guided)
- Off-by-one error in the To SIP header parameter parser leads to a crash (coverage-guided)
- Segmentation fault in SIP response building function (coverage-guided)
- Segmentation fault in MD5 calculator when generating the To tag suffix leads to DoS (coverage-guided)
- Potential crash in tm module's t_reply_matching (coverage-guided)
- Heap-buffer-overflow in function `parse_hname2` (coverage-guided)
- Segmentation fault in rewrite_ruri when processing a malicious SIP message (coverage-guided)
- Memory leak in `parse_mi_request` might lead to Denial of Service (coverage-guided)
- Buffer over-read in function `stream_process` leads to DoS (coverage-guided)
```

the bugs

- Buggy code mostly around 12-17 years old
- Most of the vulnerabilities were memory bugs

the "obvious" findings

buggy code:

```
while(*start != '\n')
    start--;
```

fix:

```
while(*start != '\n' && start > stream->body.s)
    start--;
```

the weird findings

```
const str *emsg;
emsg = &str_init(MESSAGE_400);
```

leads to:

```
(gdb) bt
#0  0x000055b8afa1892b in memcpy (__len=<optimized out>, __src=<optimized out>, __dest=0x7fc08e8457f4)
    at /usr/include/x86_64-linux-gnu/bits/string_fortified.h:34
#1  build_res_buf_from_sip_req (code=code@entry=400, text=text@entry=0x7fff3d795630, new_tag=new_tag@entry=0x7fc08c5fc6e0 <sl_tag>,
    msg=msg@entry=0x7fc08e844160, returned_len=returned_len@entry=0x7fff3d7954c8, bmark=bmark@entry=0x7fff3d7954d0)
    at msg_translator.c:2569
```

Are these the only bugs present in the code?

Obviously, **no**!

How can we find more bugs?

- Keep the fuzzers running for a longer time (oss-fuzz)
- More fuzzers should be included by the OpenSIPS dev team (using our fuzzing framework as a starting point)
- Modify the code to become more *fuzzable*

libfuzzer

libfuzzer tools provided

- Based on llvm/clang v12 (available on Github)
- Built as a Docker image
- Provides generic scripts to build, test and run the fuzzer

AddressSanitizer and coverage mapping

- built using using AddressSanitizer (`-fsanitize=fuzzer,address`)
- profiling is performed by building an instrumented binary (`-fprofile-instr-generate`)
- the fuzzer generates coverage mapping which is useful for code coverage analysis (`-fcoverage-mapping`)

static modules

The code is modified using a script named `prebuild.sh` to allow for modules to be statically compiled in the fuzzer binary. We did not opt to use directives such as `STATIC_TM` since not all modules supported this feature.

suggestion: make all modules capable of being statically compiled

structure of a fuzzer

```
int initialized;

static char buf[BUF_SIZE + 1];
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)
{
    //
}
```

structure of a fuzzer

```
int initialized;

static char buf[BUF_SIZE + 1];
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)
{
  if (Size > BUF_SIZE)
  {
    return 0;
  }

  memcpy(buf, Data, Size);
  buf[Size] = 0;
}
```

structure of a fuzzer

```
int initialized;

static char buf[BUF_SIZE + 1];
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)
{
  if (Size > BUF_SIZE)
  {
    return 0;
  }

  memcpy(buf, Data, Size);
  buf[Size] = 0;

  if (initialized == 0)
  {
    initfuzzer();
    initialized = 1;
  }
}
```

structure of a fuzzer

```
int initialized;

static char buf[BUF_SIZE + 1];
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)
{
  if (Size > BUF_SIZE)
  {
    return 0;
  }

  memcpy(buf, Data, Size);
  buf[Size] = 0;

  if (initialized == 0)
  {
    initfuzzer();
    initialized = 1;
  }

  struct sip_msg *req;
  req = (struct sip_msg *)pkg_malloc(sizeof(struct sip_msg));
  if (req == NULL) return 0;
  memset(req, 0, sizeof(struct sip_msg));

  req->buf = buf;
  req->len = Size;
  req->rcv.src_ip.af = AF_INET;
  req->rcv.dst_ip.af = AF_INET;

  if (parse_msg(buf, Size, req) == 0)
  {
    if ((req->via1==0) || (req->via1->error!=PARSE_OK)){
        goto end;
    }

    // call function under test
  }

end:
  free_sip_msg(req);
  pkg_free(req);
  return 0;
}
```

the framework explained: the Dockerfile

```
FROM ghcr.io/enablesecurity/fuzzing-images/libfuzzer:latest

RUN apt-get update
RUN apt-get install -y build-essential git pkg-config libssl-dev flex bison \
    libsqlite3-dev libsctp-dev libradcli-dev libhiredis-dev unixodbc-dev \
    libconfuse-dev libmysqlclient-dev libexpat1-dev libxml2-dev libpq-dev \
    zlib1g-dev libperl-dev libsnmp-dev libdb-dev libldap2-dev \
    libcurl4-gnutls-dev libgeoip-dev libpcre3-dev libmemcached-dev \
    libmicrohttpd-dev librabbitmq-dev liblua5.1-0-dev libncurses5-dev \
    libjson-c-dev uuid-dev python-dev libmaxminddb-dev \
    vim net-tools psmisc python3
WORKDIR /
RUN git clone https://github.com/OpenSIPS/opensips
WORKDIR /opensips
RUN git checkout f380dc59eb79ec2153d5b0d6e8374c0877fca525

RUN CC=clang \
    CFLAGS="-g -O0 -fsanitize=fuzzer-no-link,address -fprofile-instr-generate -fcoverage-mapping" \
    LDFLAGS="-g -O0 -fsanitize=fuzzer-no-link,address -fprofile-instr-generate -fcoverage-mapping" \
    make

ENTRYPOINT [ "/fuzzer/fuzz.sh" ]
```

the framework explained: the image `buildimage.sh` script

```
#!/bin/bash

docker build -t opensips-pentest/libfuzzer .
```

the framework explained: the run script

Once the image has been created, the only script that should be used is the `./run.sh` script. This script can be used to build, test and debug fuzzers.

Example usage:

```
# Compile the fuzzer code and exit, used to make sure
# the code compiles while the test is being written
./run.sh parse_msg compile

# Run the fuzzer with the default settings
./run.sh parse_msg run

# Run the fuzzer for 1 round, 600 seconds, parallel=8
./run.sh parse_msg run 600 1 8
```

how do I add a new test?

The `libfuzzer` deliverables directory looks like this (showing only relevant files for clarity):

```
- Dockerfile
- buildimage.sh
- run.sh
- scripts/
    - patches/
    - Makefile
    - prebuild.sh
- tests/
- report/
```

New tests are added under the `tests/` directory

structure of a test

```
- main.c
- build.sh
- corpus/
  - payload1
  - payload2
  - ...
- dict/
  - dict.dict
```

`build.sh` of each test

This is the file that is called by the `./run.sh` to compile the fuzzer:

```
#!/bin/bash

/test/prebuild.sh

make -s -f /test/Makefile clean
make -s -f /test/Makefile all -j$(nproc)
```

what happens when the fuzzer is executed?

1. the `build.sh` script is called
2. if the binary has been created, corpus minimization is performed
3. run the fuzzer
4. if the binary crashes, or the maximum time has been exceeded, the coverage report is automatically generated and shared in the `report/` directory
5. The resulting corpus is compressed and kept for the next run of the fuzzer

what is the output of each test?

The output of each test are copied to the host running the Docker container in the `report/` directory. This is a sample for a test that found a crash:

```
- report/
    - sipmsgops-remove_body_part_f/
        - 2022-01-21_06-00-33/
            - crash-144564680b4e65d4c8ddecf5d7ef276ba7d343a8
            - crash-144564680b4e65d4c8ddecf5d7ef276ba7d343a8.bt
            - report/
                - nocoverage.txt
                - opensips/
                    - opensips source code coverage
                - test/
            - run.2022-01-21_06-00-36.functions-report.txt
            - run.2022-01-21_06-00-36.linebyline-report.html
            - run.2022-01-21_06-00-36.linebyline-report.txt
            - run.2022-01-21_06-00-36.sourcefiles-report.txt
```

more about coverage report

```
> tree report
report
├── nocoverage.txt
├── opensips
│   ├── async.c
│   ├── cachedb
│   ├── config.h
│   ├── context.c
│   ├── crc.c
│   ├── crc.h
│   ├── db
│   │   └── db.h
│   ├── dprint.h
│   ├── dset.c
│   ├── evi
│   │   ├── event_interface.c
│   │   ├── evi.h
│   │   └── evi_params.c
```

sample coverage report

```
      0 | char* get_hdr_field(char* buf, char* end, struct hdr_field* hdr)
   413M | {
      0 |
   413M |     char* tmp;
   413M |     char *match;
   413M |     struct via_body *vb;
   413M |     struct cseq_body* cseq_b;
   413M |     struct to_body* to_b;
   413M |     int integer;
      0 |
```

global coverage report

Since different fuzzers cover different sections of the code, it was useful to create a global line-by-line reporting tool, which parses all the reports generated by each fuzzer, and generates one consolidated report. This report is useful to observe which parts of the code have not been fuzzed, and would help the testers to create better tests and corpus.

making use of `weggli` and the coverage reports

https://github.com/googleprojectzero/weggli

```
# Look for code which calls q_memchr and has not been covered by one of the fuzzers
python3 weggli-coverage.py ../libfuzzer-docker/report '{q_memchr();}' \
    `pwd`/.work/opensips

# A more complex example which looks for code which performs loop operations on a
# pointer type variable
python3 weggli-coverage.py ../libfuzzer-docker/report '{_* $i; for($i=_;;){};}' \
    `pwd`/.work/opensips
```

The results of the tool looking for uncovered `q_memchr` calls would look like this:

```
opensips/mi/mi_trace.c:342->tok_end = q_memchr(bw_string, type_delim, len);
opensips/mi/mi_trace.c:377->tok_end = q_memchr( list.s, list_delim, list.len );
opensips/modules/dialog/dialog.c:1092->while( (p=q_memchr(s.s,DLG_SEPARATOR,s.len))!=NULL ) {
opensips/modules/dialog/dlg_hash.c:386->if ((p=q_memchr(ct->s, ':', ct->len))==NULL) {
opensips/modules/dialog/dlg_req_within.c:921->body->s = q_memchr(ct->s, ':', body_s.len - 7);
```

what were the stumbling blocks?

- which modules should be initialized first (solved by replicating what `main.c` was doing)
- fully understand how the TM, SL and Dialog create and handle transactions in isolation
- understand why certain crashes occurred

parser statistics

Filename	Regions	Missed Regions	Cover	Functions	Missed Functions	Executed	Lines	Missed Lines	Cover	Branches	Missed Branches	Cover
opensips/parser/contact/contact.c	428	211	50.70%	5	1	80.00%	207	23	88.89%	220	86	60.91%
opensips/parser/contact/parse_contact.c	227	136	40.09%	5	1	80.00%	119	28	76.47%	96	55	42.71%
opensips/parser/digest/digest.c	135	105	22.22%	7	2	71.43%	108	63	41.67%	86	70	18.60%
opensips/parser/digest/digest_parser.c	185	12	93.51%	9	0	100.00%	159	1	99.37%	106	6	94.34%
opensips/parser/digest/param_parser.c	193	14	92.75%	1	0	100.00%	34	0	100.00%	80	0	100.00%
opensips/parser/hf.c	92	55	40.22%	3	1	66.67%	146	42	71.23%	106	32	69.81%
opensips/parser/hf.h	27	27	0.00%	1	1	0.00%	32	32	0.00%	52	52	0.00%
opensips/parser/msg_parser.c	2272	1448	36.27%	15	10	33.33%	1081	532	50.79%	1242	805	35.19%
opensips/parser/msg_parser.h	70	35	50.00%	5	4	20.00%	103	75	27.18%	34	18	47.06%
opensips/parser/parse_allow.c	60	23	61.67%	1	0	100.00%	37	8	78.38%	26	14	46.15%
opensips/parser/parse_allow.h	7	0	100.00%	1	0	100.00%	6	0	100.00%	4	2	50.00%
opensips/parser/parse_authenticate.c	457	129	71.77%	7	0	100.00%	265	28	89.43%	260	76	70.77%
opensips/parser/parse_authenticate.h	1	0	100.00%	1	0	100.00%	4	0	100.00%	0	0	-
opensips/parser/parse_body.c	648	424	34.57%	12	6	50.00%	422	192	54.50%	328	218	33.54%
opensips/parser/parse_call_info.c	114	37	67.54%	3	0	100.00%	79	1	98.73%	50	19	62.00%

tm module statistics

Filename	Regions	Missed Regions	Cover	Functions	Missed Functions	Executed	Lines	Missed Lines	Cover	Branches	Missed Branches	Cover
opensips/modules/tm/async.c	332	332	0.00%	3	3	0.00%	219	219	0.00%	164	164	0.00%
opensips/modules/tm/callid.c	110	92	16.36%	4	3	25.00%	62	45	27.42%	50	42	16.00%
opensips/modules/tm/cluster.c	1245	1234	0.88%	12	10	16.67%	369	359	2.71%	516	513	0.58%
opensips/modules/tm/dlg.c	558	558	0.00%	20	20	0.00%	383	383	0.00%	312	312	0.00%
opensips/modules/tm/fix_lumps.h	58	56	3.45%	1	0	100.00%	42	36	14.29%	34	33	2.94%
opensips/modules/tm/h_table.c	248	198	20.16%	15	5	66.67%	227	123	45.81%	146	128	12.33%
opensips/modules/tm/lock.c	40	26	35.00%	8	4	50.00%	42	18	57.14%	14	10	28.57%
opensips/modules/tm/lock.h	4	0	100.00%	2	0	100.00%	20	12	40.00%	0	0	-
opensips/modules/tm/mi.c	608	608	0.00%	22	22	0.00%	488	488	0.00%	304	304	0.00%
opensips/modules/tm/sip_msg.c	1538	984	36.02%	6	2	66.67%	860	280	67.44%	982	575	41.45%
opensips/modules/tm/sip_msg.h	23	23	0.00%	1	1	0.00%	15	15	0.00%	14	14	0.00%
opensips/modules/tm/t_cancel.c	67	67	0.00%	4	4	0.00%	64	64	0.00%	34	34	0.00%
opensips/modules/tm/t_cancel.h	11	11	0.00%	2	2	0.00%	19	19	0.00%	6	6	0.00%
opensips/modules/tm/t_ctx.c	15	15	0.00%	9	9	0.00%	27	27	0.00%	0	0	-
opensips/modules/tm/t_dlg.c	2	2	0.00%	2	2	0.00%	7	7	0.00%	0	0	-

AFL/AFLnet/AFLplusplus

AFL/AFL++

```
int main()
{
  char *buffer = malloc(BUF_SIZE);
  int size = read(STDIN_FILENO, buffer, BUF_SIZE - 1);

  if (size > 0 && size < BUF_SIZE - 1)
  {

    buffer[size] = 0;
    fuzz(buffer, size);
  }
  free(buffer);
  return 0;
}
```

AFLnet

https://github.com/aflnet/aflnet

```
make all include_modules="db_mysql" -j8 CC="afl-clang-fast"
/aflnet/afl-fuzz -d -i /corpus -o out -m 200 -N udp://127.0.0.1/5060 -P SIP ./opensips -f /configs/opensips.cfg
```

What's left

DoS testing

What we have done so far

- Setup an OpenSIPS instance on a VPS
- Create up to 100 attack nodes targeting OpenSIPS
- Monitor the target system using a SIP ping tool, CPU and memory monitoring using `sar`
- The above is fully scripted and automated

What's left to be done

The actual tests against:

- Configurations that target specific functions
- Configurations that are production ready

What else is left?

Partially covered

- indialog fuzzing (only blackbox done)
- DDoS testing of various different configurations
- internal IDs
- topology hiding - needs tests related logic

Not covered

- TLS specific coverage
  - verify / authenticate client-certificates and related checks (looking for authentication bypass)
  - repeat tests with wolfSSL implementation
- Transport protocol testing

What's next

Status

Original plan (part 1)

- OpenSIPS parser including:
  - `parser`
  - `parser/digest`
  - `parser/contact`
  - `parser/sdp`
- OpenSIPS auth module
- OpenSIPS tm module
- OpenSIPS dialog module

Original plan (part 2)

- The following modules:
  - UDP
  - TCP
  - TLS
  - WS
- The OpenSIPS Management Interface
- Focus on exposed internal IDs
- Topology hiding

Do that in 23 days

We did around 45 days

(and loved it ... mostly)

Unfortunately ...

- we do not know how to continue
- ... without your help

What we do with our clients

- Pentesting - Done **for** You
- Consultancy - Done **with** You
- DIY (SIPVicious PRO)

Note: this is no sales pitch!

:-)

Done with You

- ad hoc training
- knowledge transfer
- helping testers (and developers) implement security testing
    - CI/CD pipelines for fuzzing
    - manual testing too

How could this be useful for both of us?

- we cannot do much more within the current agreement
- think that OpenSIPS security would benefit greatly from your work
- we would be happy to assist you and contribute

Examples: Done with You and our deliverables

- integrating with OSS-Fuzz
- keep automated tests up to date
- improve *fuzzability* of the whole project
- DDoS tests on systems setup / configured by you

Examples: Done with You and the remaining scope

- TLS specific tests
- blackbox fuzzing on systems setup / configured by you
- topology hiding

So, what is next?

- we are committed to this project
- let's discuss!