The state of OSS-Fuzz

Assessing statefulness in OSS-Fuzz: performance and results

Lars Esselink

Supervisors:
Erik Poll
Bram Westerbaan

Second readers:
Cristian Daniele
Seyed Behnam Andarzian

Faculty of Science
Computing Science
Radboud University
June 26, 2024

Abstract

OSS-Fuzz is a project by Google that fuzzes open-source software on their data centers, to improve the reliability of open-source software. The underlying fuzzers of OSS-Fuzz are libFuzzer, AFL++, and Honggfuzz. We expected OSS-Fuzz to be ineffective because these fuzzers are not designed to deal with stateful software. Using 'stateless' fuzzers on stateful software might be faster in executions but it might also be worse at finding bugs because stateless fuzzers that use fuzzing targets effectively circumvent the state transitions and thus might not reach bugs that are unreachable. That is why we want to explore OSS-Fuzz to see if it gives results in stateful software effectively or is wasting resources by fuzzing stateful software, with a stateless fuzzer. We try to answer this question by comparing bugs between a stateful fuzzer and OSS-Fuzz, using open-source software that has known vulnerabilities.

We picked two open-source projects, ProFTPD, and open62541, which have versions with known vulnerabilities, and we fuzzed them with AFLNet, a stateful fuzzer, and OSS-Fuzz to see if OSS-Fuzz is wasting resources. Experimenting with ProFTPD showed us that AFLNet did find more bugs, but that was not because OSS-Fuzz was not good enough, but because the fuzzing targets created by the developers of ProFTPD were not good enough. AFLNet did not find any bugs in open62541 that we could reproduce. OSS-Fuzz, however, did find bugs. We do now know if this bug could be found by AFLNet because the malicious input would have been caught elsewhere in the program, which prevents the input from reaching the part that would have crashed (which might not happen when fuzzing using OSS-Fuzz).

We cannot conclude that OSS-Fuzz is wasting resources, but can conclude more research is needed. To get a more exact answer, we would require more time and more case studies to see if the results we found are flukes or a repeating pattern. It is not fair to compare AFLNet and OSS-Fuzz because OSS-Fuzz is heavily dependent on good fuzzing targets. When looking at ProFTPD we found that the lack of good fuzzing targets lowered the amount of bugs found while open62541 had good fuzzing targets and outperformed AFLNet.

OSS-Fuzz might find fewer bugs than AFLNet, but it does substantially more executions than AFLNet (30,000 vs 5) because AFLNet fuzzes the socket entry point which in turn uses the whole code while OSS-Fuzz only fuzzes parts of the code. This might make OSS-Fuzz better at finding bugs, but whenever a bug is not reachable by OSS-Fuzz (because the fuzzing targets do not cover enough code) they can be faster but they will never find the bug.

Contents

1 Introduction

When we refer to stateless software, we mean software that does not maintain or track any state information. Let us take for example an implementation of FFmpeg which is stateless software that converts audio/video files to other types. It does not keep track of any states. But we also have stateful software, which is the opposite of stateless software. Stateful software does keep track of states, for example, an implementation of FTP (file transfer protocol). FTP has to be in specific states to do different commands. To download or upload files, you will need to be authenticated and thus be in the authenticated state.

Fuzzing is a popular technique used for testing software[14]. The advantage of fuzzing is that is almost fully automated. For normal test cases, you will need to write tests for each new code you write, while fuzzing only requires a one-time code creation (for example a fuzzing target) and thus saves time when trying to test your software. Another advantage is that it might catch edge cases that normal test cases would not catch. The downside of fuzzing is that you will need more computing power than normal test cases.

As long as you have computing power, you can test your software for every edge case and increase the reliability of your software. This is where Google OSS-Fuzz[8] comes in. Google created the OSS-Fuzz project to help improve open-source software. It takes open-source projects and fuzzes them using their data centers. Interestingly, OSS-Fuzz does not use any stateful fuzzer but fuzzes stateful software, and thus might not be as good at finding bugs as AFLNet at fuzzing stateful software.

In this thesis, we want to answer the following research question: is OSS-Fuzz wasting resources by fuzzing stateful software projects with stateless fuzzers? We investigate the matter by looking at a specific version of two stateful software projects with a known bug and fuzzing it with OSS-Fuzz's stateless and stateful fuzzers. If a stateful fuzzer can but OSS-Fuzz cannot find these bugs, it indicates that OSS-Fuzz is wasting resources. This is why we want to test it in an OSS-Fuzz environment to get the closest possible simulation of how OSS-Fuzz is doing it.

In Chapter 2, we discuss what fuzzing exactly is and the terms we are not on one line with. In Chapter 3, we look at related research. In Chapter 4, we discuss the method we use to get the results. Chapters 5 and 6 explain the results of fuzzing. In Chapter 7 we write down what we stumbled upon but was not in the scope of our research and can be researched in the future. In Chapter 8 we discuss the findings of the whole research, what we found, and explain the conclusions. In the last Chapter 8 we have a glossary for terms we use in this research.

2 Fuzzing

In this chapter, we discuss what fuzzing exactly is and the terms we are not on one line with. We also discuss the different fuzzing aspects we are not familiar with. Some abbreviations are not explained here but are written out in Chapter 8.

Fuzzing is a method to find vulnerabilities in software. This is done by sending many, (semi) automatically generated inputs to the software and seeing if it crashes or hangs. The mutation of messages is often guided based on multiple factors, like code coverage or the number of crashes. Fuzzers can be categorized into stateful and stateless groups, based on if they are aware of the states inside the software.

2.1 Stateless software & fuzzers

Software that does not have any internal states is called stateless, an example of stateless software is a browser that implements HTTP. The HTTP protocol does not keep track of any states and is therefore stateless. The fuzzers created for stateless software do not consider the states the software is in when making decisions.

2.1.1 AFL++

AFL++[5], the daughter of AFL, is a coverage-guided fuzzer. The coverage guide works by looking at the coverage of the current mutation in the input and decides based on the coverage what mutation should be done next. The fuzzer also listens to the feedback, which is the part where it makes mutations based on the information it gets from the instrumentation (like crashes/hangs). The coverage guide and the feedback guide both get their information from the instrumentation, but the feedback guide looks at the program metrics (like memory usage), crashes, and hangs. You normally use standard command line input (stdin for short) to fuzz using AFL++, but in this research, we use a special C library to get it working on the fuzzing targets of libFuzzer.

2.1.2 Honggfuzz and libFuzzer

Honggfuzz and libFuzzer[13] are coverage-guided fuzzers. They use so-called fuzzing targets, specific functions especially made for fuzzers inside the SUT, to execute specific functions in code (as if those functions are called by the program itself). They then check the code coverage and generate more mutations, maximizing code coverage. The difference between Hongfuzz and libFuzzer is minimal and both use the same principle but in different coding.

2.1.3 Reproducibility

Fuzzing has a problem when it comes to reproducibility (also known as flakiness), especially fuzzing targets. Let us take a look at the simple state model in Figure 1. Since fuzzing targets focus on specific parts of the code, the errors found work only in a specific state of the program. For example, when there is a bug found in fuzzing target 1, which operates in state 4, we can only reproduce that error when using that fuzzing target. When we want to reproduce the error on the original program, we need to construct a message that takes us from state 1 to state 4, using actions a, b, and c. This construction would be feasibly impossible to do and thus the error found with the fuzzing target would be impossible to reproduce outside the fuzzing target.

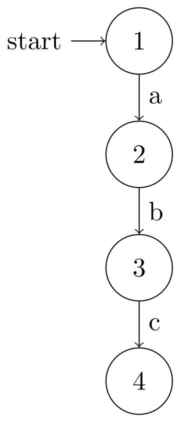


Figure 1: Simple state model

2.2 Stateful fuzzers

Software is generally stateful, meaning that during the execution of software, it can be in different states. Take the server side of FTP (file transfer protocol) for example, it has an idle state in which it can receive connections, but it also has a connected state where a client is connected and can send commands to the server. Stateless fuzzers do have some guiding mechanisms, like code coverage or instrumentation feedback, but none take states into account. With the example of FTP, a stateless fuzzer would fuzz FTP, but would not find the other state (It would be able to find other states when run for infinity). This is why we have stateful fuzzing. Stateful fuzzers also have state feedback that is taken into account when mutating input.

2.2.1 AFLNet

AFLNet[11] is a stateful fuzzer used to fuzz network protocols. It uses state feedback and measures code coverage. The state feedback works by receiving messages from the SUT and learning how the state model of the SUT is built. This feedback is then used to create a mutation of the input to trigger state transitions or stay in a specific state. This state feedback is also used to increase code coverage. It is used to fuzz stateful software (explained in Section 2.2) but can also be used to fuzz stateless software (explained in Section 2.1) but it would run slower than stateless fuzzers.

2.2.2 Socket vs Target

One of the differences between AFLNet and OSS-Fuzz is the difference between socket vs target fuzzing (visually explained in Figure 2). Socket fuzzing, the method of AFLNet, fuzzes the program interacting with it via a socket. The program is fuzzed using normal operations and tries to find it as if were a normal user. Fuzzing target, the method of libFuzzer, AFL++ (in OSS-Fuzz), and Honggfuzz, use fuzzing targets that focus on pieces of code within the program. It does not fuzz the program as a whole but works more like unit testing, where different sections of code are tested. Both have pros and cons (as discussed in 2.1.3).

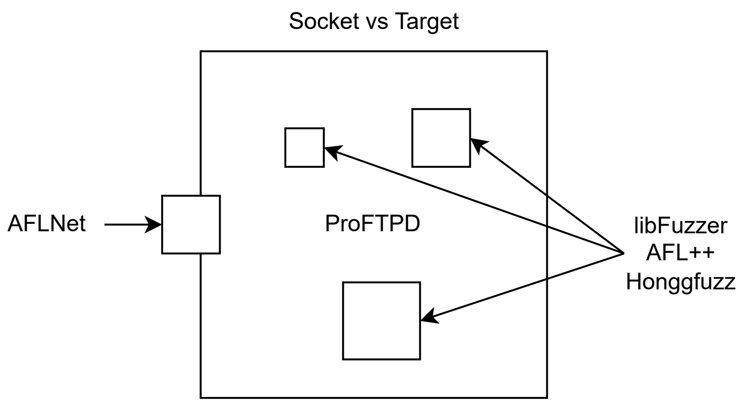


Figure 2: The difference between socket and target fuzzing

2.3 OSS-Fuzz

OSS-Fuzz[8] is a project by Google to fuzz open-source software using their data centers. The fuzzers included are libFuzzer[13], AFL++[5] and Honggfuzz[7]. None of these fuzzers specifically support stateful software. Google does not state why it chose to not include stateful fuzzers, but one assumption would be that using stateless fuzzers is less work for the OSS-Fuzz team. Stateless fuzzers only require fuzzing targets, while stateful fuzzers require a method of extracting the state, to use state feedback.

2.4 ASan and UBsan

ASan[9] (Address sanitizer) and UBsan[6] (Undefined Behavior sanitizer) are compiler flags to add checks in the compiled code for memory violations and undefined actions. ASan checks for common memory errors such as using memory after freeing it, memory leaks, etc. For example, whenever we use the function memcpy in the C language, we copy a block of memory to another block of memory. But this operation does not check bounds, and thus whenever we go over bounds it is not detected. When run without ASan memory faults might go unnoticed since it might overflow in memory that is not used, but with ASan this is immediately noticed and the ASan crashes the program. UBsan checks at run time for undefined behavior, for example whenever we use a non-static variable in the C language before it is declared, will give undefined behavior because of the nature of the C language. Just like memory violation, this might go unnoticed as the effect is not catastrophic, but with UBsan this will be immediately noticed and UBsan will crash the program. ASan and UBsan, in our case, will help with finding bugs more easily since every memory violation and undefined behavior will immediately show up and count as a crash and thus be detected by the fuzzer.

3 Related Work

In Section 3.1, we present the results of Yu Ding and Le Goues[3], which classifies the most bugs found in OSS-Fuzz.

3.1 Bugs found by OSS-Fuzz

OSS-Fuzz had its first launch in 2016. As of August 2023, it has fuzzed 1000 projects, found 36000 bugs, and identified over 10000 vulnerabilities [8]. Ding et al.[3] explored all the bugs and performed an empirical study. They analyzed over 4 years of data and 23,907 bugs. These bugs were categorized under 16 categories found in Figure 3. They were also categorized into fixed and unfixed and flakiness (reproducibility, whether a bug is reproducible or not). The biggest category is timeout. But they did not categorize based on stateless or stateful. This is what we planned on doing, albeit not that many projects, to see the difference between stateless and stateful fuzzers.

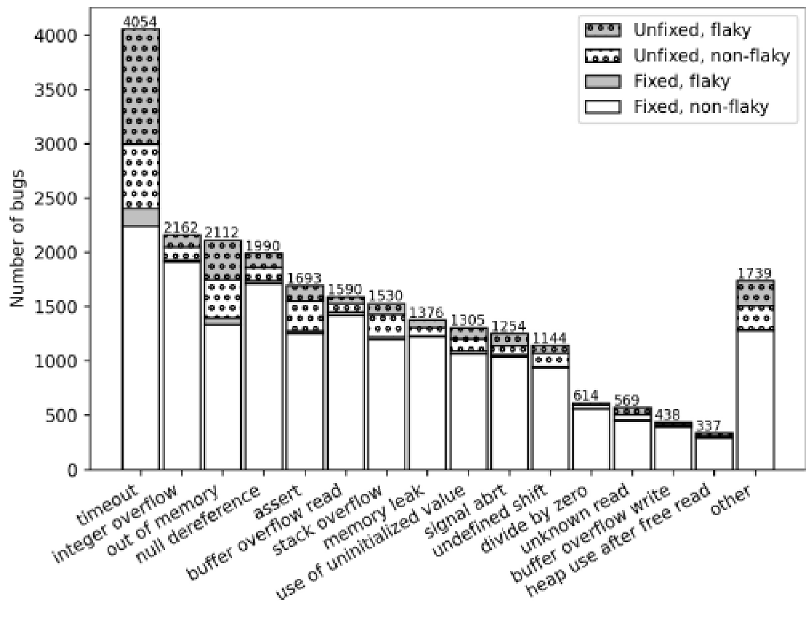


Figure 3: bugs found in OSS-Fuzz categorized in 15 categories[3]

4 Method

This chapter discusses an overview of the different steps to take to find out if OSS-Fuzz is wasting its resources with 2 case studies.

4.1 Project selection

First, we need to select a project to test our hypothesis on. We look at the list of already fuzzed projects[1] and manually filter it on if the project uses sockets. This list now contains projects that have sockets but are not necessarily stateful, for example, the project ffmpeg which does not have any states, and the socket use is optional. So now we manually apply a second filter, which filters on protocols that were specifically made for networking (as they are almost always stateful) and we get the following list as shown in Table 1. From this list, we can pick a suitable candidate. We chose ProFTPD and open62541 because both protocols (OPCUA and FTPD) already had an implementation in AFLNet which was easier for us to use and thus we did not have to create it ourselves.

bitcoin-core	libvnc
bluez	mosquitto
dropbear	msquic
fast-dds	open5gs
freeradius	open62541 (opcUA)
freerdp	opendnp3
gnutls	openssl
hiredis	openweave
irssi	pidgin
libgit2	pupnp
libmodbus	qpid-proton
librdkafka	resiprocate
libssh	wolfssl
libssh2	wpantund
libtorrent	proftpd

Table 1: The list of projects after we applied 2 filters.

4.2 Seed creation

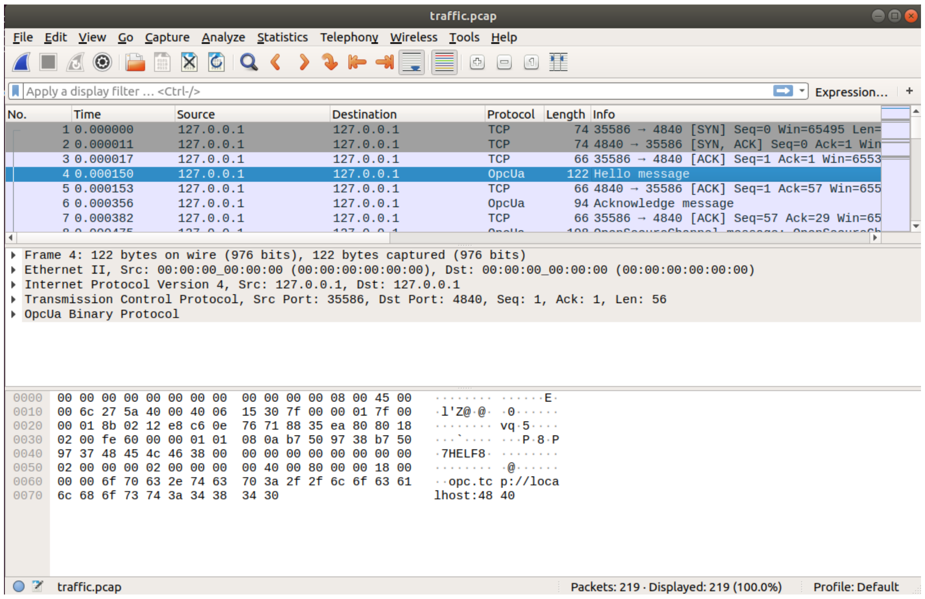
We are going to need seeds for fuzzers to work. This is not required but it might be better to give a starting point instead of randomly starting. These seeds are input strings that are used by the fuzzers and fed into the SUT. These seeds are also mutated to test different inputs. OSS-Fuzz does not require us to provide the seeds, because they are either provided by the project owner or not needed. For AFLNet however, we need to make the seed ourselves (as OSS-Fuzz uses seeds in another format). This is done using the normal operation of a SUT and tcpdump. Tcpdump records all traffic within the specified parameters, which lets us use that same traffic as input seed for AFLNet. For example, let us take

[1] https://github.com/google/oss-fuzz/tree/master/projects

open62541. First, we need to use the SUT to create traffic, such that tcpdump can capture it. In this example, we use:

```
tcpdump -w traffic.pcap -i lo port 4840
```

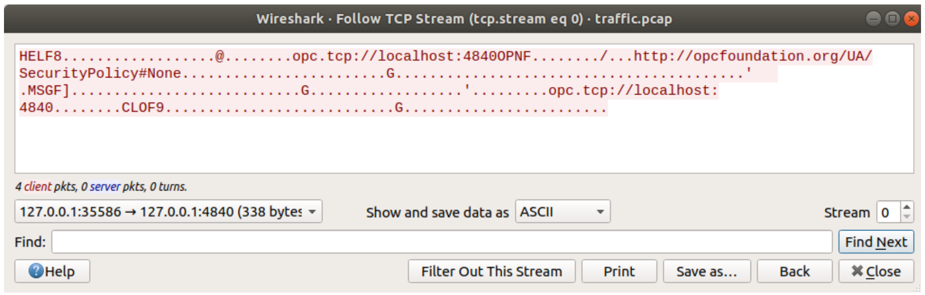
This starts tcpdump and tells it to write to the file called "traffic.pcap" and listen specifically to the loopback network interface on port 4840 (the port used by open62541 tools). After which you perform the normal operation of the SUT and close tcpdump. Now we have a file called "traffic.pcap" which holds the traffic of the program. We can read this with Wireshark, which we can find in Figure 4.


traffic.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…
+

```
No.   Time       Source      Destination  Protocol Length Info
    1 0.000000   127.0.0.1   127.0.0.1    TCP        74 35586 → 4840 [SYN] Seq=0 Win=65495 Len=
    2 0.000011   127.0.0.1   127.0.0.1    TCP        74 4840 → 35586 [SYN, ACK] Seq=0 Ack=1 Win
    3 0.000017   127.0.0.1   127.0.0.1    TCP        66 35586 → 4840 [ACK] Seq=1 Ack=1 Win=6553
    4 0.000150   127.0.0.1   127.0.0.1    OpcUa     122 Hello message
    5 0.000153   127.0.0.1   127.0.0.1    TCP        66 4840 → 35586 [ACK] Seq=1 Ack=57 Win=655
    6 0.000356   127.0.0.1   127.0.0.1    OpcUa      94 Acknowledge message
    7 0.000382   127.0.0.1   127.0.0.1    TCP        66 35586 → 4840 [ACK] Seq=57 Ack=29 Win=65

▸ Frame 4: 122 bytes on wire (976 bits), 122 bytes captured (976 bits)
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 35586, Dst Port: 4840, Seq: 1, Ack: 1, Len: 56
▸ OpcUa Binary Protocol

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 6c 27 5a 40 00 40 06  15 30 7f 00 00 01 7f 00   ·l'Z@·@· ·0······
0020  00 01 8b 02 12 e8 c6 0e  76 71 88 35 ea 80 80 18   ········ vq·5····
0030  02 00 fe 60 00 00 01 01  08 0a b7 50 97 38 b7 50   ···`···· ···P·8·P
0040  97 37 48 45 4c 46 38 00  00 00 00 00 00 00 00 00   ·7HELF8· ········
0050  02 00 00 00 02 00 00 00  00 40 00 80 00 00 18 00   ········ ·@······
0060  00 00 6f 70 63 2e 74 63  70 3a 2f 2f 6c 6f 63 61   ··opc.tc p://loca
0070  6c 68 6f 73 74 3a 34 38  34 30                     lhost:48 40

traffic.pcap                              Packets: 219 · Displayed: 219 (100.0%)    Profile: Default
```

Figure 4: The content of traffic.pcap using Wireshark

In Figure 4 you see network packets. From this, we can view the TCP stream (traffic of a session) and view the traffic between the client and server of open62541. We only care about the traffic between the client and server (as seen in Figure 5), since that is what AFLNet is going to use for fuzzing (and not the server-to-client traffic). From this, we can save it as raw bytes (otherwise it will be sent as ASCII, which the server cannot interpret) and use it as a seed.



Wireshark · Follow TCP Stream (tcp.stream eq 0) · traffic.pcap

```
HELF8.................@........opc.tcp://localhost:4840OPNF......../...http://opcfoundation.org/UA/
SecurityPolicy#None.......................G.......................................'
.MSGF].........................G.................'.........opc.tcp://localhost:
4840........CLOF9...........................G.......................
```

4 client pkts, 0 server pkts, 0 turns.

127.0.0.1:35586 → 127.0.0.1:4840 (338 bytes

Show and save data as ASCII

Stream 0

Figure 5: The specific TCP stream of a session between client and server

4.3 Fuzzing instructions

After we have the input seeds for AFLNet (and the supplied seeds for OSS-Fuzz), we can proceed to fuzz. AFLNet and OSS-Fuzz work a little bit differently. We use AFLNet in our environment, we do not run it in a pre-supplied way, while we run OSS-Fuzz using their helper script. This script does the setup for the SUT and the fuzzers for us using only commands. We chose to do this because we want to use OSS-Fuzz as close to their environment as possible. Doing it this way makes it easier for us to make conclusions as to why they chose some options (as discussed in Chapter 1) and makes it also easier to see if OSS-Fuzz is wasting resources.

We also need a SUT to test the difference. We pick a stateful SUT from the list of already fuzzed projects by OSS-Fuzz. Once we find a project, we search if the project has a version that has a known bug that can be found by fuzzers. If we find a project that is fuzzed by OSS-Fuzz and has a version with a known bug, we can start fuzzing that project version with AFLNet and OSS-Fuzz.

4.3.1 AFLNet instructions

First, we start with AFLNet. For AFLNet we run it using:

```
afl-fuzz -i /path/to/seed/ -o /path/to/output -N
{tcp/udp}://{IP}/{PORT} -P {PROTOCOL} -q 3 -s 3 -E -K -R
-m none {path/to/executable}
```

You will need the supply the command with the following: the path to the seed, the path to where you want to save the output, does the SUT uses TCP or UDP, The IP and PORT of the SUT, the protocol of the SUT, and the path to the executable. The other parameters can be changed but do not make a big difference. Once it is fuzzing the program, we choose a time frame in which we fuzz the SUTs. We chose 24 hours as this is a reasonable amount of time to find a crash/hang, after which we manually stopped AFLNet. The results are stored in the output directory we defined.

4.3.2 OSS-Fuzz instructions

For OSS-Fuzz, we need to use the helper script. This script helps us build the SUT, build the fuzzers and run the fuzzers. First, building the SUT:

```
python3 /path/to/helper.py build_image {PROJECT}
```

The building of the SUT is reasonably straightforward, you just need to supply the project and it will build the SUT. After building the SUT, we need to build the fuzzer:

```
python3 /path/to/helper.py build_fuzzers --engine
{libfuzzer,afl,honggfuzz} {PROJECT}
```

To build the fuzzers you need to supply the fuzzer you want to run and the project you want to fuzz. Once finished, we execute the last step, running the fuzzer:

```
python3 /path/to/helper.py run_fuzzer --corpus-dir
/path/to/output/ --engine {libfuzzer,afl,honggfuzz}
{PROJECT} {FUZZER TARGET}
```

To run the fuzzer you need to supply the script with an output directory, the fuzzer you want to fuzz with, the project, and one of the pre-supplied fuzzing targets. After executing this it will start fuzzing. AFL++ and Honggfuzz will run until stopped, while libFuzzer will stop once it finds a hang/crash. So we run AFL++ and Honggfuzz also 24 hours, just like AFLNet. We can then find the results in the output directory we defined.

4.4 Comparing results found by fuzzing

When comparing results we do not have a global method, since we do not yet know what we get for results. For a single bug found by a single fuzzer, we cannot compare it to anything so we do not have a specific method we use. When multiple fuzzers find a bug, we can use debug to find where the crash/hang came from.

Using a debugger like GDB, we can inspect the execution step by step to find out where the program crashes. GDB works by setting so-called breakpoints in the execution of the program. These breakpoints cause the execution of the code to pause at the breakpoint. For example, if you set a breakpoint at a function, the debugger loads the functions onto the stack but does not yet execute it and pauses the execution until further notice. Let us take for example the source code for the MDNS fuzzing target (found in Figure 19 on page 41).

1. First we analyze the source code and set breakpoints at important functions, in this case at message_parse, mdnsd_new and mdnsd_in.

2. After the breakpoints are set, we run the program with an input. The debugger stops at the first breakpoint (or crashes if there is a crash before the breakpoint). We then choose to "continue" which continues to the next crash/breakpoint, "step" which takes one step in execution, or "finish" which finishes the function it is currently at.

3. Repeat steps one and two until you find a crash. After which you can dive deeper into a function, set breakpoints at more functions, and analyze the results until you find the exact line of code that crashes the program

5 Fuzzing ProFTPD

For our experiment, we use a version of each SUT with a vulnerability that can be found using fuzzing. We compare AFLNet and OSS-Fuzz (with the underlying fuzzers) results.

For the first case, we use ProFTPD. ProFTPD is an open-source FTP implementation. FTP is used to transfer files from the client to the server and vice versa. ProFTPD is a highly configurable and secure implementation of FTP. One of the versions of ProFTPD[2] has an out-of-bounds vulnerability which can be found using AFLNet (commit 5e57d41)[2], using ASan and UBSan. So we are going to test if for this version OSS-Fuzz can find the same crashes and hangs as AFLNet. In Section 5.1 we discuss the experiments we ran on ProFTPD using AFLNet and analyze those results. In Section 5.2 we talk about the experiments on ProFTPD using OSS-Fuzz and analyze the results. In Section 5.3 we take the results from Section 5.1 and 5.2, and we compare them to see if there is a difference in bugs found and execution speed.

5.1 Fuzzing ProFTPD with AFLNet

Run 1 We first compile ProFTPD without ASan and UBSan. We then tried to fuzz commit 5e57d41, but we did not get any results. As you can see in Table 7 on page 32, there are no crashes or hangs found, and it is very slow. The low amount of executions is because AFLNet fuzzes the program as a whole using the socket. When the whole program needs to execute to fuzz a small input it results in few executions done per minute. This is different from running fuzzing targets, which only run the targeted code and not the whole program. The lack of crashes or hangs might be because the original issue was found with ASan and UBSan enabled.

Run 2 We then run AFLNet with ASan and UBSan but AFLNet complained about memory usage when using ASan in combination with a 64-bit OS. It turns out that you need special arguments to run AFLNet with ASan, otherwise on a 64-bit OS you might use too much memory, namely the **m** flag. We managed to compile it with the instructions found in Figure B.1, which immediately gives us more results. After running AFLNet for 47 hours, we got 17 hangs where the last hang was found 24 hours before ending the fuzzer, meaning that it did not find anything new during the last 24 hours of fuzzing. Once we find a specific crash or hang, we can replay (using afl-replay) the exact test sequence used before it crashed or hung. We used afl-replay for our results and it immediately recognized the 'flaw' that caused these hangs. Namely, the ProFTPD configuration allowed for 3 login attempts and after that, it stopped the program. Which AFLNet classified as hang since the ProFTPD stopped. Thus all these hangs were false positives.

[2]https://github.com/ProFTPD/ProFTPD/issues/1683

Run 3 After changing the ProFTPD configuration to have unlimited login attempts, we ran it again, but this time for 24 hours since the last run did not have any new paths in the last 24 hours. The results (found in Table 2) showed 29 hangs. Unfortunately, the out-of-bound crash was not found.

ProFTPD	
Last new path	3 min
Last unique hang	2 hrs, 21 min
unique hangs	29
Last unique crash	None
unique crashes	0
Total execs	389k
Total paths	2808

Table 2: The third run of ProFTPD using AFLNet, using ASan and UBSan, with the correct configuration, extracted from table 8 on page 32.

Table 2 contains all the information of AFLNet. We can see how long it ran and how long it took until it found a new path/crash/hang. There are also statistics about the fuzzing process, for example, how many cycles it has done, the total paths found and the amount of (unique) crashes and hangs.

Analyzing run 3 We use the tool that AFLNet has provided (called aflnet-replay) to replay the packet that causes the crash/hang. We start the example server, in this case, we start ProFTPD. Then we use aflnet-replay on all 29 of the unique replay packets. From the 29 hangs, 28 gave a segmentation fault on the aflnet-replay side and the last one did not give us an error. We focus on the 27th packet that caused a segmentation fault when using aflnet-replay. We do not get any information from the client or the server side. Using tcpdump, we can see if there is traffic at all. Traffic is sent, but only one way before crashing. The server side sends the welcome message to the client but nothing is sent by the client. It must be a problem with the first line of the input since nothing is sent. The first few lines are displayed in Figure 6.

```
00000000  0e 00 00 00 55 53 45 52  20 70 72 6f 66 74 70 64  |....USER proftpd|
00000010  0d 0a 0e 00 00 00 50 41  53 53 20 70 72 6f 66 74  |......PASS proft|
00000020  70 64 0d 0a 06 00 00 00  51 55 49 54 0d 0a 0e 00  |pd......QUIT....|
00000030  00 00 52 4e 46 52 20 74  65 73 74 2e 74 78 0d 0a  |..RNFR test.tx..|
00000040  0e 00 00 00 50 41 53 53  20 70 72 6f 66 74 70 64  |....PASS proftpd|
00000050  0d 0a 0e 00 00 00 53 4d  4e 54 0d 70 72 6f 66 74  |......SMNT.proft|
00000060  70 64 0d 0a 0e 00 00 00  50 41 53 53 20 70 72 6f  |pd......PASS pro|
00000070  66 74 70 64 0d 0a 08 00  00 00 54 59 50 45 20 41  |ftpd......TYPE A|
```

Figure 6: The hex dump of the first few lines of the input that hang ProFTPD

The hex dump in Figure 6 does not say much and does not seem to be the problem of ProFTPD but is more likely Ubuntu or aflnet-replay. We do not know why these segmentation faults are happening and to find out why, we would need to debug the aflnet-replay tool which is out of the scope of this research. We mark it as non-reproducible.

Github input We also tested ProFTPD using the input of the original GitHub issue[3], which caused a heap buffer overflow (found in Appendix C.1). The problem was caused by sending 3722 '\' and ProFTPD having no sanitation rule to sanitize that many wrong characters and thus overflowing in memory. Our AFLNet run, however, did not give us the same result, probably because fuzzing is nondeterministic meaning that it will not come up with the same mutations in the seed every time.

5.2 Fuzzing ProFTPD with OSS-Fuzz

We run the same experiments with OSS-Fuzz. We take the vulnerable version and run it with OSS-Fuzz. OSS-Fuzz does not use the whole program as SUT but uses fuzzing targets instead. Which focuses on a specific part of the program (explained in Section 2.2.2).

5.2.1 LibFuzzer

We fuzzed ProFTPd with libFuzzer (description found in Section 2.1.2) using the fuzzing targets in OSS-Fuzz. This is a relatively straightforward process. OSS-Fuzz contains a script to start the project to start fuzzing, which needs information about the fuzzer engine and fuzzing target. To submit a project to OSS-Fuzz its necessary to provide fuzzing targets for libFuzzer to use. We use these fuzzing targets to test how well libFuzzer can find bugs. We used the instructions in Appendix B.2.1 to run libFuzzer with ASan. Unfortunately, the OSS-Fuzz helper script does not allow multiple sanitizers at the same time, as we did with AFLNet. Because of this, we ran libFuzzer with address sensitization for 48 hours. Unfortunately, we did not find crashes or hangs. A closer look at the fuzzing target reveals that the target is not covering enough code. As you can see in Table 3, the coverage of the fuzzing target is high, while the coverage of the SUT is low. This means that the fuzzing target only focuses a small part of the code, even though the fuzzing target is fully (functionally) executed. The low code coverage on the SUT is a bad sign. The bug in AFLNet is impossible to find by libFuzzer because of the low code coverage and the fact that the bug we found was not inside the code reached by this fuzzing target. We can create a fuzzing target ourselves, but that defeats the purpose of testing the current OSS-Fuzz environment since we want to know if (without our help) libFuzzer can find the same bugs as AFLNet.

[3]https://github.com/ProFTPD/ProFTPD/issues/1683

PATH	**LINE COVERAGE**	**FUNCTION COVERAGE**	**REGION COVERAGE**
proftpd/	1.26% (740/58550)	2.88% (55/1908)	1.30% (567/43665)
fuzzer.c	88.89% (16/18)	100.00% (1/1)	83.33% (5/6)
TOTALS	**1.29% (756/58568)**	**2.93% (56/1909)**	**1.31% (572/43671)**

Table 3: The code coverage, provided by the OSS-Fuzz helper script, on the ProFTPD fuzzing target

5.2.2 AFL++

AFL++ is simpler than libFuzzer. It does not require any fuzzing targets, just an input seed to start fuzzing. It also accepts a dictionary file that specifies the correct input AFL++ can use. The installation of ProFTPD is the same as AFLNet found in Appendix B.1. The only difference is that it uses other instrumentation tools to compile the binaries and after installation, it is run with another input.

Run 1 The first run gave us no results. As you can see in Table 9 (on page 32), it says "odd, check syntax" after the last new path find. It means that AFL++ cannot find any path, this is a clear sign the fuzzer is not working correctly. It turns out to be because AFL++ does not support sockets in its normal mode, but AFL++ works with STDIN or AFL fuzzing targets. What we tried to do, however, was to run AFL++ directly on the executable. AFL++ did not do any fuzzing because the executable only used the socket entry point. OSS-Fuzz has a slightly different approach. OSS-Fuzz does not use STDIN mode or AFL fuzzing targets. What OSS-Fuzz does, is use the targets from libFuzzer and create some glue between the clang instrumentation and AFL++, using a C library, to get it to work with the non-AFL instrumentation[4].

Run 2 We can fuzz ProFTPD with AFL++ using the OSS-Fuzz helper script (instructions found in Appendix B.3). This time AFL++ worked as intended, but did not give us any errors or hangs (as found in Appendix A.1). It did find enough paths but it did not find any bugs. This is as expected since libFuzzer did not find anything, mainly because of the complexity of the fuzzing target (as explained in Section 5.2.1).

5.2.3 Honggfuzz

Just like AFL++ and libFuzzer, we compile Honggfuzz using the instructions for OSS-Fuzz (found in Appendix B.4) and let it run for 24 hours. Honggfuzz also did not give us any crashes or hangs (results found in Appendix A.3) for the same reason the other fuzzers did not find anything, namely a fuzzing target with low code coverage.

[4]https://github.com/google/oss-fuzz/issues/2094

5.3 Comparing AFLNet and OSS-Fuzz over ProFTPD

When we look at ProFTPD, we can see a clear difference between AFLNet and OSS-Fuzz. Although we do not have a clear answer yet to the question "Is OSS-Fuzz wasting resources", we can already see improvements when using AFLNet.

- AFLNet found a buffer overflow bug while OSS-Fuzz did not find any crashes or hangs.
- While fuzzing ProFTPD using AFLNet (using self-made seeds), we did not find any reproducible crashes or hangs. However, when we used the specific seed reported in a GitHub issue using AFLNet, we got ProFTPD to crash giving us a heap buffer overflow as a result.
- The fuzzing targets in ProFTPD do not provide good code coverage. There is only one fuzzing target which does not cover enough code to even find the heap buffer overflow found by AFLNet. This means that if OSS-Fuzz ran it, they would not find the same crash AFLNet found.

6 Fuzzing open62541

Open62541[12] is an open-source implementation of the protocol OPC Unified Architecture (OPC UA for short), which is used for data exchange between sensors and applications. Open62541 is a good candidate because its stateful and fuzzed by OSS-Fuzz. Fijneman[4] tested the execution speed difference between boofuzz[5] and AFLNet on open62541. Although our research's main topic (comparing fuzzers) is the same, we will only use AFLNet. Fijneman's results are summarized in Figure 7. As we can see there are 68 crashes and 14 hangs. We can use these results for a baseline of what we are supposed to find with AFLNet on open62541. For open62541 we will use commit 46d0395, just like Fijneman.

		Crashes		**Hangs**			
Implementation	**Lang.**	Total	Reproducible	Total	Reproducible	**Exec/s**	**Runtime**
legacy OPC UA ANSI-C Stack	C	8	8	0	0	$\pm 20 \cdot 8$	48 hours
open62541	C	68	0	14	0	$\pm 8 \cdot 8$	24 hours
FreeOpcUa	C++	43	43	9	0	$\pm 8 \cdot 8$	24 hours

Figure 7: The result of fuzzing OPC UA implementations in AFLNet by Fijneman[4]

In Section 6.1, we discuss the results of fuzzing open62541 in AFLNet. In Section 6.2 we discuss the the results of fuzzing open62541 with OSS-Fuzz. In the last Section, 6.3, we compare the results of AFLNet and OSS-Fuzz and see if it made any difference.

6.1 Fuzzing open62541 with AFLNet

We use the test application created by the developers of open62541, server_ctt.c, and client.c . We modify the SUT as described in [4], such that server_ctt and client can be used without authentication and be used by AFLNet. In detail, we:

1. removed **maxSecureChannels**, **maxSession** and **shutdownDelay** from server_ctt.c . This causes the server_ctt to be faster inside AFLNet.
2. changed the variable **enableAnon** to **true** in server_ctt.c so that we can use it without authentication.
3. changed **UA_Client_connectUsername** to **UA_Client_connect** inside client.c. This causes the client to connect to server_ctt anonymously.

[5]https://github.com/jtpereyda/boofuzz

After this, we created a seed file for AFLNet to start fuzzing with. This can be done by using the instructions found in Section 4.2. After which, we can start fuzzing using the following line:

```
afl-fuzz -i seed/ -o 1run -N tcp://127.0.0.1/4840 -P OPCUA
-q 3 -s 3 -E -K -R -m none ./server_ctt
```

Run 1 Running it with the instructions above, we did not get any problems (results found in Table 12 on page 33). There are 15 unique hangs with a good amount of executions.

Analyzing run 1 However, after doing a test run, we stumbled upon a fault in our modifications of the SUT. AFLNet only reached 1 state, which can be found in the ipsm.dot file inside the output folder. In this file, AFLNet keeps track of the states it has reached. After talking to Fijneman[4] about the modifications of the SUT, he made a new improved version (since it was already known to Fijneman[4]) which handled the state transition correctly on the new version, but AFLNet could only reach 2 states (found in Figure 8), which is still incorrect since FTP has 4 states. The problem turns out to be the seed namely, this input only reaches 2 states and it would be more difficult for AFLNet to find other state transitions. As we needed to create a seed for AFLNet using tcpdump, we saved it in the wrong format (ASCII instead of raw). Saving in the right format, got us the correct state model (found in Figure 8).

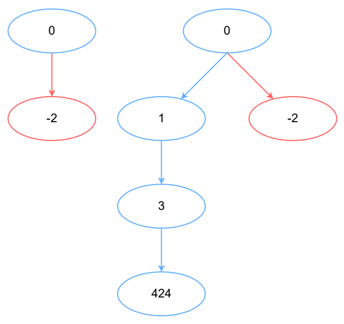


Figure 8: The incorrect state model (left) and the correct state model (right) of open62541 found by AFLNet

Run 2 AFLNet found 43 unique hangs with 143 total paths (found in Table 13, in page 34). This is still not the same as the crashes found by Fijneman[4]. But may give us some information about faulty coding.

Analyzing run 2 As already mentioned, AFLNet found 43 hangs. We used aflnet-replay (as we did in Section 5.1). If we test all the hang packets, they all give a segmentation fault on the aflnet-replay side except the last hang. The last 2 give an error on the server side, BadInterError and BadSecureChannelTokenUnknown. They are not bugs, since the bad inputs are caught by the coding and are rejected.

Run 3 We found another fault in our seed creation. When we create a seed for AFLNet to use (described in Section 4.2), we use tcpdump and Wireshark to create a seed from the TCP stream. We made the mistake of only looking at one TCP stream (as explained in Section 4.2) instead of all 4 TCP streams. Thus, this time we create 4 seeds from all the TCP streams and run AFLNet again. As we can see in Table 14 on page 34, There are no crashes or hangs found, enough paths found, and a good amount of total executions. Run 3 resulted in fewer hangs than run 2. This could be because fuzzing is a non-deterministic way of testing, meaning that it is not guaranteed to find the same bugs as the previous run with the same input.

6.2 Fuzzing open62541 with OSS-Fuzz

6.2.1 LibFuzzer

Just like ProFTPD, we fuzz open62541 using libFuzzer with the provided fuzzing targets. The open62541 developers provided multiple fuzzing targets, as shown in Table 4. We use these targets to test whether the libFuzzer can find the same crashes as AFLNet.

base 64 decode
base 64 encode
binary decode
binary message
JSON decode
JSON decode encode
MDNS message
MDNS xht
src ua util
TCP message

Table 4: The list of fuzzing targets created by open62541 team

Run 1 First we build the open62541 image, build the libFuzzer image and fuzzing targets for open62541, and then run it using the OSS-Fuzz helper script (instructions found in Appendix B.2.1) on each of the fuzzing target in the Table 4. The results can be found in Table 5.

Fuzzing open62541 results					
	crash	hang	execs / s	time ran	reproducible
Base64 decode			4096	24h	
Base64 encode			4096	24h	
Binary decode	x		0	$\leq 30s$	yes
Binary message	x		0	$\leq 10s$	no
JSON decode		x	4096	$\leq 5m$	no
JSON decode encode		x	4096	$\leq 5m$	no
MDNS message	x		0	$\leq 10s$	yes
MDNS xht			44k	24h	
src UA utitl			4096	24h	
TCP message			4	24h	

Table 5: The results of the different libFuzzer fuzzing targets

Analyzing libFuzzer results If we look at Table 5, we can see that 3 crashes were found by libFuzzer. Most of them were found right after startup, which explains why the run time was less than a few seconds. Let us first look at the crash (found in Figure 14, on page 39) of MDNS message. A heap buffer overflow caused the error. Rerunning the same tests, results in the same error but with different addresses. This rules out a coincidence. There also were 2 targets that did not respond (hang). The JSON decode and the JSON decode encode target both had a time out of more than 25 seconds, probably because the fuzzing target no longer responded to new inputs.

Binary decode First we take a look at the crash of the binary decode target (found in Figure 18 on page 40), which gives us an error because of an unknown address. When we want to debug and try to find out where it crashes, we use GDB (more information about the debugging process is found in Section 4.4). So we run GDB on binary decode with the seed that crashes the program. We set some breakpoints to see where it crashes. This gives us, that it crashes on the last assert (also seen in the crash itself). The assert in question:

```
UA_assert(ret == UA_STATUSCODE_GOOD);
```

Unfortunately, the debugger did not provide us with any information about local variables. This was because the dwarfdata (which is debugger data), is corrupted. Namely, if we start the debugger, we get "Cannot handle DW_FORM_unknown". This fails because the variable 'ret' is not the same as 0x00000000.

Binary message The binary message crash (found in Figure 17 on page 40) was caused by a heap buffer overflow. It went 4 bytes over bounds. However, when we try to reproduce the crash using the seed directly on the seed without a fuzzer, we cannot reproduce the crash because it does not crash. Which means this crash is not reproducible.

JSON decode The JSON decode hang (found in Figure 15 on page 39), was caused by an unknown bug. When we run it in GDB, and set some breakpoints around functions inside the fuzzing target, we find that it hangs around the execution of the "UA_decodeJson" function. Whenever we try to dive deeper, we cannot get an answer, because it hangs immediately after going to the function somewhere else in the code. Analyzing this further, we found that after setting the memory limit for the program to unlimited, it succeeded. Thus, the timeout is the problem of not having enough memory given to the fuzzing target. Although the hang is reproducible, it turns out it is not a hang because it did not get enough memory.

JSON decode encode The JSON decode encode hang (found in Figure 16 on page 40), was caused by the same bug as the JSON decode fuzzing target. The JSON fuzzing target did not have enough memory, which was fixed by giving the fuzzing target more memory.

MDNS message The MDNS message crash (found in Figure 14 on page 39), was also caused by heap buffer overflow. It went immediately out of bounds with less than a byte. When we run GDB with the fuzzing target, we set a breakpoint at specific open62541 functions. And when we run the GDB with the seed that crashes the target, it crashes on the execution of the function "message_parse". When we dive deeper into the message parse function we see that it crashes on the execution of:

```
if (!_rrparse(m, m->an, m->ancount, &buf, m->_bufEnd))
    return false;
```

This function parses the RRS part of the MDNS message. When we dive even deeper, we find that it does a memcpy, which writes something in memory but does not check bounds and thus overflows in the allocated memory. There is a comment in the code that states "check if the message has the correct size, i.e. the count matches the number of bytes" (found in 22 on page 44), which the code below should check the size and stop our crash from happening, but does not.

6.2.2 AFL++

As discussed in Section 5.2.2, OSS-Fuzz uses the fuzzing targets from libFuzzer to fuzz using AFL++, using a C library created by the OSS-Fuzz team. So we run AFL++ on the existing fuzzing targets with the helper script (described in Appendix B.3).

Run 1 As shown in Table 6,AFL++ found crashes in the same piece of code but fewer crashes/hangs than libFuzzer. Although AFL++ found a hang in Base64 and MDNS xht, taking a quick peek at the results, it seems that the PC where these fuzzers were run on, had a hang itself since the Base64 encode and the MDNS xht hang happened at the same time. The Binary decode and the MDNS message crashes were no coincidence, they were also found by libFuzzer.

Fuzzing open62541 results				
	crash	hang	execs / s	time ran
Base64 decode			56k	24h
Base64 encode		1	30k	24h
Binary decode	97		10k	24h
Binary message			5	24h
JSON decode			30k	24h
JSON decode encode			30k	24h
MDNS message	15		30k	24h
MDNS xht		1	30k	24h
src UA utitl			30k	24h
TCP message			5	24h

Table 6: The results of the different libFuzzer fuzzing targets using AFL++

Binary decode As shown in Table 6, there are 61 crashes. However, when we tried to analyze the crashes, we found out that the OSS-Fuzz helper script did not save the seeds that caused the crashes. So we had to rerun AFL++ on the MDNS fuzzing target, but on the second run, it found 97 crashes which is more crashes. When we try to recreate the crashes we get the same error we found using libFuzzer in Section 6.2.1, binary decode.

MDNS message As we can see in Table 6, there are 15 crashes. After running the target seeds with the input seeds that caused these crashes, we can see they all cause a buffer overflow (as we have seen in Section 6.2.1, MDNS message). When we use the debugger to find what causes the crash and try the same method we used in Section 6.2.1 (with the same breakpoints), we can see that it crashes in the same location as found with libFuzzer on the MDNS message fuzzing target.

6.2.3 Honggfuzz

Upon trying to fuzz the fuzzing targets with Honggfuzz, we found out that the developers of open62541 do not specify that they want to be fuzzed with Honggfuzz, as every project that is fuzzed by OSS-Fuzz needs to provide specific information, for example, what fuzzer to use. Knowing that they did not specify Honggfuzz to be used, we will not try to get Honggfuzz working on their fuzzing targets (as we also found out that Honggfuzz does not work on open62541).

6.3 Comparing AFLNet and OSS-Fuzz over open62541

Observing the bugs found in open62541 (just like in Chapter 5) we noticed a clear difference in the bugs found. This time it is in favor of OSS-Fuzz. The results of open62541 make answering the question “Is OSS-Fuzz wasting resources” more difficult, since this experiment gave opposite results than in Chapter 5.

- OSS-Fuzz found a buffer overflow bug while AFLNet did not find any crashes or hangs.
- Unlike ProFTPD, the fuzzing targets were good enough to find a bug in programming. AFLNet however, did not find anything using the socket. This could probably be because the bad input would be filtered out when reaching the part that crashes on OSS-Fuzz.
- The underlying fuzzers of OSS-Fuzz did not have many differences since AFL++ and libFuzzer found the same bug in the same place.

As we discussed in Section 2.1.3, it is not easy to reproduce fuzzing target crashes on the original program (generally speaking) because it is nearly impossible to reproduce a trace with the part that caused the fuzzing target to crash (as discussed in Section 2.1.3). That is also why it is not easy to compare these results as we cannot test the bad inputs of OSS-Fuzz on the program as a whole but we can test the bad inputs of AFLNet. This was no problem when using ProFTPD as SUT because the fuzzing target in ProFTPD did not find any crashes or hangs. This would have been a problem if the fuzzing target did find crashes or hangs.

7 Future work

7.1 More stateful fuzzers

In the future, it might be an idea to test different stateful fuzzers, like SGFuzzer[1], to see if there is a better fuzzer than AFLNet, that would work better for OSS-Fuzz in both bug finding and getting it to work with OSS-Fuzz. This is because AFLNet requires the developer to create a packet parser in code into AFLNet and there might be an even better stateful fuzzer at finding bugs.

7.2 More case studies

It is also an idea to test using more case studies because 2 case studies might not be enough to exactly tell the difference between AFLNet and OSS-Fuzz. It would also be good to test protocols with encryption. We did not test protocols with encryption because the nature of encryption makes it more difficult to fuzz those protocols than protocols without.

7.3 Better fuzzing targets

In this project by Google, called OSS-Fuzz-gen[10], Google uses LLMs (Large Language Models), like OpenAI ChatGPT or Google Gemini, to create fuzzing targets to increase the code coverage of the fuzzing for stateless and stateful projects. If you have better code coverage, you have more chance of finding faults in the code. OSS-Fuzz depends heavily on good fuzzing targets for all their fuzzers (libFuzzer, AFL++, Honggfuzz), this is why it would be an idea to try to use the LLM OSS-Fuzz-gen, to improve code coverage and thus increase OSS-Fuzz bug finding vs AFLNet.

8 Conclusion

Speed OSS-Fuzz does substantially more executions than AFLNet (30,000 vs 5) because AFLNet fuzzes the socket entry point which in turn uses a large part of the code while OSS-Fuzz only fuzzes smaller parts of the code at once. This might make OSS-Fuzz better at finding bugs, but whenever a bug is not reachable by OSS-Fuzz (because of state transitions) they can be faster but they will never find the bug.

ProFTPD AFLNet improved the bug finding on ProFTPD compared to OSS-Fuzz. Although OSS-Fuzz is not wasting a lot of resources since ProFTPD has only one fuzzing target, OSS-Fuzz might not even run the fuzzing target for long as it might notice that some statistics (like code coverage and crashes) are not changing, and thus terminating early. Points to note on ProFTPD (for more analysis of ProFTPD, see Section 5.3),

- ProFTPD has a lot to improve on when talking about the fuzzing targets. Since the fuzzing targets only reached 2.8% $\sim$ coverage (see analysis in Section 5.3), ProFTPD can add more targets like open62541 and improve the ones they have.
- AFLNet could improve OSS-Fuzz when fuzzing ProFTPD. Even though we saw that the bug finding is dependent on the creation of good fuzzing targets, when using AFLNet it would decrease the burden of the creation of good fuzzing targets to have results (as we see in Section 5.3). Implementing AFLNet does, however, require developers who request fuzzing by AFLNet, to provide an entry point (unified socket) and an implementation of the protocol into AFLNet.

Open62541 The results for open62541, however, show us something different. While we expected the same results for open62541 as ProFTPD, the results for open62541 are the opposite. Points to note on open62541 (for more analysis of open62541, see Section 6.3),

- Open62541 does not have a lot to improve on when talking about the fuzzing target, because while AFLNet did not find any reproducible bugs, OSS-Fuzz did find bugs.
- AFLNet would not improve OSS-Fuzz when fuzzing open62541, as open62541 had good enough fuzzing targets to find bugs.
- OSS-Fuzz did find crashes, more specifically AFL++ and libFuzzer did. Both fuzzers found the same bugs, which caused a buffer overflow. This bug was created by a memcpy function that did not check any bounds.
- OSS-Fuzz also found more unreachable bugs, than reachable bugs. Namely, when executing the normal path of the program, the bugs found by OSS-Fuzz might be not possible since the input might have already been filtered and checked.

AFLNet vs OSS-Fuzz We do not have a clear answer to the question "Is OSS-Fuzz wasting resources?". ProFTPD shows us that OSS-Fuzz is wasting resources since it is not fuzzing with many results. Open62541 however, shows us a different picture. AFL++ and libFuzzer found bugs in open62541 while AFLNet did not. AFLNet would be better at finding bugs against bad fuzzing targets. There would be no improvement when there are good fuzzing targets. Answering this question would require more time, more specifically a bigger sample rate.

Another way of looking at it would be that OSS-Fuzz is not wasting time but the developers are, since OSS-Fuzz depends heavily on good fuzzing targets. Creating good fuzzing targets would be wasting time instead of using AFLNet which only requires a one-time creation of a code parser inside AFLNet and takes less time.

OSS-Fuzz One of the assumptions we made is that the developers know the effect of the quality of their fuzzing targets on the quality of the results of OSS-Fuzz. This assumption might be wrong since ProFTPD has bad fuzzing targets and we do not know the exact reason why. The reason for bad fuzzing targets could also have been that the developers submitted their project for the money since OSS-Fuzz gives out money for submitting your project.

Recommendations Although there is no clear answer to the question "Is OSS-Fuzz wasting resources?", we do have a recommendation for developers who submit their projects. As OSS-Fuzz is heavily dependent on good fuzzing targets, making good fuzzing targets would increase the bug finding with OSS-Fuzz. This would also be a recommendation to OSS-Fuzz as they might want to recommend creating fuzzing targets with high code coverage to ensure good fuzzing targets.

Glossary

ASan	Address Sanitizer
UBSan	Undefined Behaviour Sanitizer
OPCUA	OPC Unified Architecture
FTP	File Transfer Protocol
SUT	System Under Test
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

References

[1] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, Boston, MA, August 2022. USENIX Association.

[2] Castaglia. Proftpd. `https://github.com/proftpd/proftpd`, 2014.

[3] Zhen Yu Ding and Claire Le Goues. An empirical study of OSS-Fuzz bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142, 2021.

[4] Mark Fijneman. *Fuzzing open source OPC UA implementations.* bachelor thesis, Radboud universiteit, Nijmegen, Gelderland, August 2023.

[5] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[6] Google. UndefinedBehaviorSanitizer. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`.

[7] Google. Honggfuzz. `https://github.com/google/honggfuzz`, 2015.

[8] Google. OSS-Fuzz. `https://github.com/google/oss-fuzz`, 2016.

[9] Google. AddressSanitizer. `https://github.com/google/sanitizers/wiki/AddressSanitizer`, 2019.

[10] Dongge Liu, Jonathan Metzman, and Oliver Chang. AI-powered fuzzing: Breaking the bug hunting barrier. `https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html`, Aug 2023.

[11] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465, 2020.

[12] Stefan Profanter, Ulius Pfrommer, and Noel Graf. open62541. `https://github.com/open62541/open62541`, 2013.

[13] LLVM project team. libFuzzer. `https://llvm.org/docs/LibFuzzer.html`.

[14] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley Professional, Boston, MA, 2007.

A Results

A.1 ProFTPD in AFLNet

The results of fuzzing ProFTPD commit 5e57d41 with AFLNet without any sanitation over multiple runs.

Run time	1 day, 20 hours, 2 minutes, 1 seconds
Last unique crash found	None
Last unique hang found	None
Total unique hangs	None
Total unique crashes	None
Total paths	800×6
Exec speed	~ 3 / sec

Table 7: The summary of the first run of AFLNet on ProFTPD, summing the statistics of 6 fuzzers

Run time	1 day, 6 hours, 42 minutes, 11 seconds
Last unique crash found	None
Last unique hang found	2 hours, 21 minutes, 3 seconds
Total unique hangs	29
Total unique crashes	None
Total paths	2808
Exec speed	1 / sec

Table 8: The summary of the third run of ProFTPD using AFLNet, using ASan and UBSan, with the correct configuration

A.2 ProFTPD in AFL++

The results of fuzzing ProFTPD commit 5e57d41 with AFL++ with sanitation over multiple runs.

Run time	0 days, 20 hours, 42 minutes, 7 seconds
Last new path found	None
Last unique crash found	None
Last unique hang found	None
Total unique hangs	0
Total unique crashes	None
Total execs	33.7 million
Exec speed	507 / sec

Table 9: The summary of the first run of AFL++ on ProFTPD

Run time	1 days, 0 hours, 22 minutes, 36 seconds
Last new path found	0 days, 3 hours, 54 minutes, 31 seconds
Last unique crash found	None
Last unique hang found	None
Total unique hangs	0
Total unique crashes	None
Total execs	4.32 billion
Exec speed	48.4k / sec

Table 10: The summary of the second run of AFL++ on ProFTPD

A.3 ProFTPD in Honggfuzz

The results of fuzzing ProFTPD commit 5e57d41 with AFL++ with sanitation over a single run.

Summary iterations	634679822
time	85000
speed	7466
crashes_count	0
timeout_count	0
new_units_added	661
slowest_unit_nm	714
guard_nb	19394
branch_coverage_percent	1
peak_rss_mb	0

Table 11: The results of ProFTPD in Honggfuzz

A.4 Open62541 in AFLNet

The results of fuzzing open62541 commit 5e57d41 with AFLNet with sanitation over multiple runs.

Run time	1 day, 0 hours, 21 minutes, 49 seconds
Last unique crash found	None
Last unique hang found	0 days, 17 hours, 53 minutes, 36 seconds
Total unique hangs	15
Total unique crashes	None
Total paths	32
Total execs	79.7k
Exec speed	~ 5 / sec

Table 12: The summary of the first run of open62541 in AFLnet

Run time	1 day, 0 hours, 36 minutes, 55 seconds
Last unique crash found	None
Last unique hang found	0 days, 17 hours, 53 minutes, 36 seconds
Total unique hangs	43
Total unique crashes	None
Total paths	32
Total execs	89.5k
Exec speed	~ 5 / sec

Table 13: The summary of the second run of open62541 in AFLNet, now with fixed AFLNet implementation

Run time	1 day, 0 hours, 33 minutes, 45 seconds
Last unique crash found	None
Last unique hang found	None
Total unique hangs	0
Total unique crashes	None
Total paths	219
Total execs	381k
Exec speed	~ 5 / sec

Table 14: The summary of the third run of open62541 in AFLNet, now with 4 seeds

A.5 Open62541 in AFL++

The results of fuzzing open62541 commit 5e57d41 with AFL++ with sanitation over a single run.

	base64 encode	base64 decode	binary decode	binary message	JSON decode
Run time	24 hours	24 hours	24 hours	24 hours	24 hours
Last unique crash	None	None	23 minutes	None	None
Last unique hang	1 hour	None	None	None	None
Total unique hangs	1	0	0	0	0
Total unique crashes	0	0	61	0	0
Total execs	1.61G	5.57G	698M	797K	2.83G
Exec speed	17.9k/sec	56.8k/sec	13.0k/sec	14.0k/sec	37.8k/sec

Table 15: The first part of the summary of the afl AFL++ results on open62541

	JSON encode	MDNS message	MDNS xht	src ua util	TCP message
Run time	24 hours	24 hours	24 hours	24 hours	24 hours
Last unique crash	None	11 hours	2 hours	None	None
Last unique hang	None	None	None	None	None
Total unique hangs	0	0	1	0	0
Total unique crashes	0	15	0	0	0
Total execs	2.72G	1.6G	234M	3.44G	349k
Exec speed	28.3k/sec	33.8k/sec	277/sec	45.2k/sec	8k/sec

Table 16: The second part of the summary of the afl AFL++ results on open62541

B Installation

The way the different fuzzers and SUTs are compiled and run.

B.1 ProFTPD

```
$ CFLAGS='-g -fsanitize=address,undefined'
    LDFLAGS='-fsanitize=address,undefined'
    CXXFLAGS='-g -fsanitize=address,undefined' CC=
    "afl-clang-fast" CXX="afl-clang-fast++" ./
    configure --enable-devel=nodaemon:nofork
$ make
$ make install
$ afl-fuzz -d -i in-ftp/ -o fourthrun/ -x /home/
    ubuntu/aflnet/tutorials/lightftp/ftp.dict -N
    tcp://127.0.0.1/2021 -R -E -q 3 -s 3 -K -D
    10000 -m none -P FTP proftpd -n -c /tmp/PFTEST
    /PFTEST.conf
```

Figure 9: Instructions to compile and fuzz ProFTPD

B.2 OSS-Fuzz

The instructions to run libFuzzer and AFL++ with the help of the OSS-Fuzz helper script.

B.2.1 Instructions for the OSS-Fuzz helper script

```
$ cd path/to/oss-fuzz
$ python3 infra/helper.py build_image
    $PROJECT_NAME
$ python3 infra/helper.py build_fuzzers --engine
    libfuzzer $PROJECT_NAME
$ python3 infra/helper.py run_fuzzer --corpus-dir
    =<path-to-temp-corpus-dir> --engine afl
    $PROJECT_NAME <fuzz_target>
```

Figure 10: Instructions to fuzz projects with libFuzzer using OSS-Fuzz helper scripts

B.3 AFL++ in OSS-Fuzz helper script

```
$ cd path/to/oss-fuzz
$ python3 infra/helper.py build_image
  $PROJECT_NAME
$ python3 infra/helper.py build_fuzzers --engine
  afl $PROJECT_NAME
$ python3 infra/helper.py run_fuzzer -corpus-dir
  =<path-to-temp-corpus-dir> --engine afl
  $PROJECT_NAME <fuzz_target>
```

Figure 11: Instructions to fuzz projects with AFL++ using OSS-Fuzz helper scripts

B.4 Hongfuzz in OSS-Fuzz helper script

```
$ cd path/to/oss-fuzz
$ python3 infra/helper.py build_image
  $PROJECT_NAME
$ python3 infra/helper.py build_fuzzers --engine
  hongfuzz $PROJECT_NAME
$ python3 infra/helper.py run_fuzzer -corpus-dir
  =<path-to-temp-corpus-dir> --engine hongfuzz
  $PROJECT_NAME <fuzz_target>
```

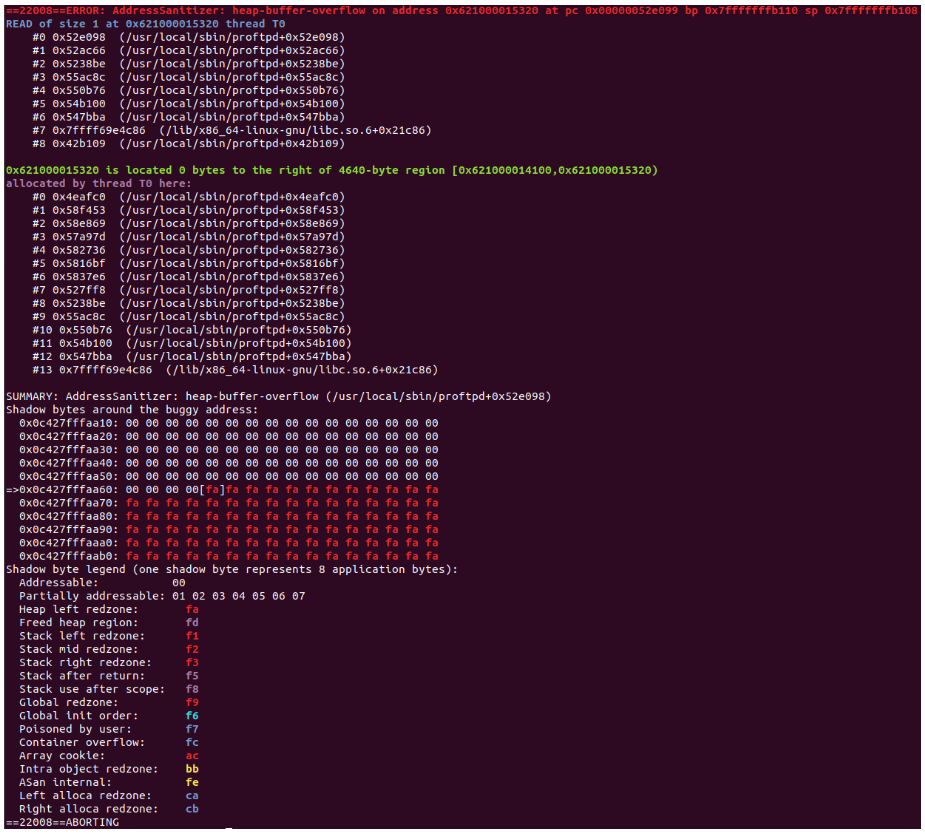
Figure 12: Instructions to fuzz projects with Hongfuzz using OSS-Fuzz helper scripts

C Bugs

The bugs we found during the fuzzing of software.

C.1 ProFTPD

The crash when trying to reproduce the crash from AFLNet on ProFTPD using the seeds from the original GitHub issue.

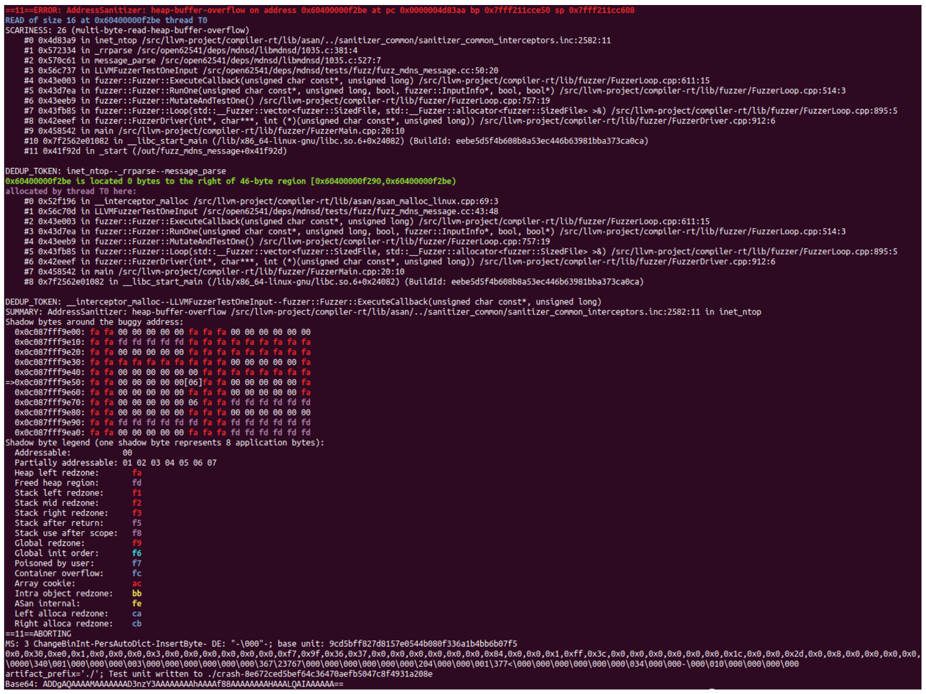


```
==22008==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x621000015320 at pc 0x00000052e099 bp 0x7fffffffb110 sp 0x7fffffffb108
READ of size 1 at 0x621000015320 thread T0
    #0 0x52e098  (/usr/local/sbin/proftpd+0x52e098)
    #1 0x52ac66  (/usr/local/sbin/proftpd+0x52ac66)
    #2 0x5238be  (/usr/local/sbin/proftpd+0x5238be)
    #3 0x55ac8c  (/usr/local/sbin/proftpd+0x55ac8c)
    #4 0x550b76  (/usr/local/sbin/proftpd+0x550b76)
    #5 0x54b100  (/usr/local/sbin/proftpd+0x54b100)
    #6 0x547bba  (/usr/local/sbin/proftpd+0x547bba)
    #7 0x7ffff69e4c86  (/lib/x86_64-linux-gnu/libc.so.6+0x21c86)
    #8 0x42b109  (/usr/local/sbin/proftpd+0x42b109)

0x621000015320 is located 0 bytes to the right of 4640-byte region [0x621000014100,0x621000015320)
allocated by thread T0 here:
    #0 0x4eafc0  (/usr/local/sbin/proftpd+0x4eafc0)
    #1 0x58f453  (/usr/local/sbin/proftpd+0x58f453)
    #2 0x58e869  (/usr/local/sbin/proftpd+0x58e869)
    #3 0x57a97d  (/usr/local/sbin/proftpd+0x57a97d)
    #4 0x582736  (/usr/local/sbin/proftpd+0x582736)
    #5 0x5816bf  (/usr/local/sbin/proftpd+0x5816bf)
    #6 0x5837e6  (/usr/local/sbin/proftpd+0x5837e6)
    #7 0x527ff8  (/usr/local/sbin/proftpd+0x527ff8)
    #8 0x5238be  (/usr/local/sbin/proftpd+0x5238be)
    #9 0x55ac8c  (/usr/local/sbin/proftpd+0x55ac8c)
    #10 0x550b76  (/usr/local/sbin/proftpd+0x550b76)
    #11 0x54b100  (/usr/local/sbin/proftpd+0x54b100)
    #12 0x547bba  (/usr/local/sbin/proftpd+0x547bba)
    #13 0x7ffff69e4c86  (/lib/x86_64-linux-gnu/libc.so.6+0x21c86)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/usr/local/sbin/proftpd+0x52e098)
Shadow bytes around the buggy address:
  0x0c427fffaa10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c427fffaa20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c427fffaa30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c427fffaa40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c427fffaa50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c427fffaa60: 00 00 00 00[fa]fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fffaa70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fffaa80: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fffaa90: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fffaaa0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fffaab0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==22008==ABORTING
```

Figure 13: Heap buffer overflow found by AFLNet

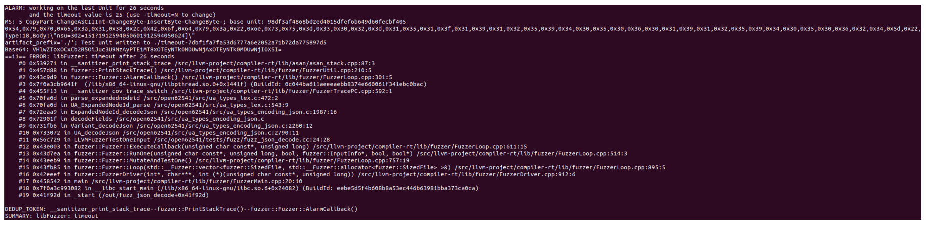
C.2 Open62541

These are the bugs found by fuzzing different targets of open62541 using libFuzzer. The fuzzing targets include MDNS message, JSON decode, JSON decode encode, binary message, binary decode.



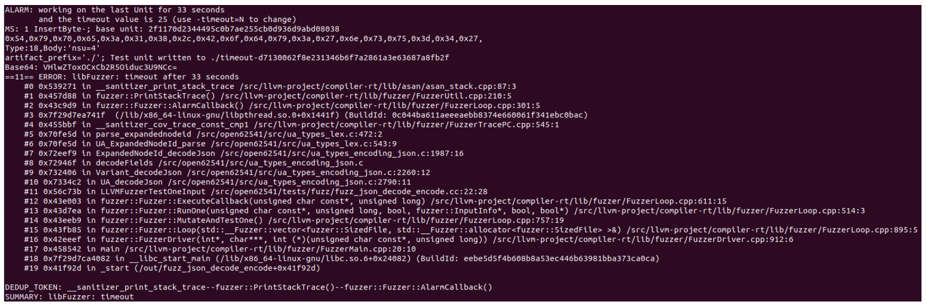
```
==11==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60400000f2be at pc 0x0000004d83aa bp 0x7fff211cce50 sp 0x7fff211cc608
READ of size 16 at 0x60400000f2be thread T0
SCARINESS: 26 (multi-byte-read-heap-buffer-overflow)
    #0 0x4d83a9 in inet_ntop /src/llvm-project/compiler-rt/lib/asan/../sanitizer_common/sanitizer_common_interceptors.inc:2582:11
    #1 0x572334 in _rrparse /src/open62541/deps/mdnsd/libmdnsd/1035.c:381:4
    #2 0x570c61 in message_parse /src/open62541/deps/mdnsd/libmdnsd/1035.c:527:7
    #3 0x56c737 in LLVMFuzzerTestOneInput /src/open62541/deps/mdnsd/tests/fuzz/fuzz_mdns_message.cc:50:20
    #4 0x43e003 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:611:15
    #5 0x43d7ea in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool, bool*) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:514:3
    #6 0x43eeb9 in fuzzer::Fuzzer::MutateAndTestOne() /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:757:19
    #7 0x43fb85 in fuzzer::Fuzzer::Loop(std::__Fuzzer::vector<fuzzer::SizedFile, std::__Fuzzer::allocator<fuzzer::SizedFile> >&) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:895:5
    #8 0x42eeef in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:912:6
    #9 0x458542 in main /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerMain.cpp:20:10
    #10 0x7f2562e01082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca)
    #11 0x41f92d in _start (/out/fuzz_mdns_message+0x41f92d)

DEDUP_TOKEN: inet_ntop--_rrparse--message_parse
0x60400000f2be is located 0 bytes to the right of 46-byte region [0x60400000f290,0x60400000f2be)
allocated by thread T0 here:
    #0 0x52f196 in __interceptor_malloc /src/llvm-project/compiler-rt/lib/asan/asan_malloc_linux.cpp:69:3
    #1 0x56c70d in LLVMFuzzerTestOneInput /src/open62541/deps/mdnsd/tests/fuzz/fuzz_mdns_message.cc:43:48
    #2 0x43e003 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:611:15
    #3 0x43d7ea in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool, bool*) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:514:3
    #4 0x43eeb9 in fuzzer::Fuzzer::MutateAndTestOne() /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:757:19
    #5 0x43fb85 in fuzzer::Fuzzer::Loop(std::__Fuzzer::vector<fuzzer::SizedFile, std::__Fuzzer::allocator<fuzzer::SizedFile> >&) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:895:5
    #6 0x42eeef in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:912:6
    #7 0x458542 in main /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerMain.cpp:20:10
    #8 0x7f2562e01082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca)

DEDUP_TOKEN: __interceptor_malloc--LLVMFuzzerTestOneInput--fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long)
SUMMARY: AddressSanitizer: heap-buffer-overflow /src/llvm-project/compiler-rt/lib/asan/../sanitizer_common/sanitizer_common_interceptors.inc:2582:11 in inet_ntop
Shadow bytes around the buggy address:
  0x0c087fff9e00: fa fa 00 00 00 00 00 fa fa fa 00 00 00 00 00 00
  0x0c087fff9e10: fa fa fd fd fd fd fd fa fa fa fa fa fa fa fa fa
  0x0c087fff9e20: fa fa 00 00 00 00 00 fa fa fa fa fa fa fa fa fa
  0x0c087fff9e30: fa fa fa fa fa fa fa fa fa fa 00 00 00 00 00 fa
  0x0c087fff9e40: fa fa 00 00 00 00 00 00 fa fa fa fa fa fa fa fa
=>0x0c087fff9e50: fa fa 00 00 00 00 00[06]fa fa 00 00 00 00 00 fa
  0x0c087fff9e60: fa fa 00 00 00 00 00 fa fa fa 00 00 00 00 00 fa
  0x0c087fff9e70: fa fa 00 00 00 00 00 06 fa fa fd fd fd fd fd fd
  0x0c087fff9e80: fa fa 00 00 00 00 00 fa fa fa 00 00 00 00 00 00
  0x0c087fff9e90: fa fa fd fd fd fd fd fd fa fa fd fd fd fd fd fd
  0x0c087fff9ea0: fa fa 00 00 00 00 00 fa fa fa fd fd fd fd fd fd
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==11==ABORTING
MS: 3 ChangeBinInt-PersAutoDict-InsertByte- DE: "-\000"-; base unit: 9cd5bff827d8157e0544b080f336a1b4bb6b07f5
0x0,0x30,0xe0,0x1,0x0,0x0,0x0,0x3,0x0,0x0,0x0,0x0,0x0,0x0,0xf7,0x9f,0x36,0x37,0x0,0x0,0x0,0x0,0x0,0x0,0x84,0x0,0x0,0x1,0xff,0x3c,0x0,0x0,0x0,0x0,0x0,0x0,0x1c,0x0,0x0,0x2d,0x0,0x8,0x0,0x0,0x0,0x0,
\0000\340\001\000\000\000\003\000\000\000\000\000\000\367\23767\000\000\000\000\000\000\204\000\000\001\377<\000\000\000\000\000\000\034\000\000-\000\010\000\000\000\000
artifact_prefix='./'; Test unit written to ./crash-8e672ced5bef64c36470aefb5047c8f4931a208e
Base64: ADDgAQAAAAMAAAAAAAD3nzY3AAAAAAAAhAAAAf88AAAAAAAAHAAALQAIAAAAAA==
```

Figure 14: Error found while fuzzing open62541 with libFuzzer and fuzzing target, MDNS message



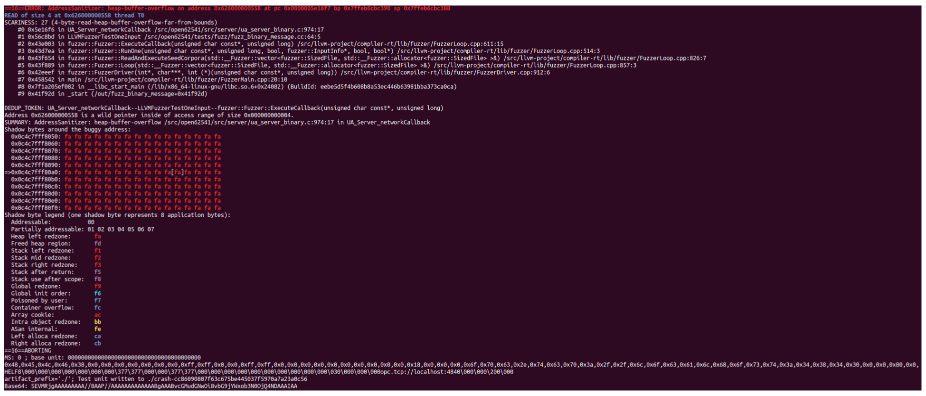
```
ALARM: working on the last Unit for 26 seconds
       and the timeout value is 25 (use -timeout=N to change)
MS: 5 CopyPart-ChangeASCIIInt-ChangeByte-InsertByte-ChangeByte-; base unit: 98df3af4868bd2ed4015dfef6b649d60fecbf405
0x54,0x79,0x70,0x65,0x3a,0x31,0x38,0x2c,0x42,0x6f,0x64,0x79,0x3a,0x22,0x6e,0x73,0x75,0x3d,0x33,0x30,0x32,0x3d,0x31,0x35,0x31,0x3f,0x31,0x39,0x31,0x32,0x35,0x39,0x34,0x30,0x35,0x30,0x36,0x30,0x31,0x39,0x31,0x32,0x35,0x39,0x34,0x30,0x35,0x30,0x36,0x32,0x34,0x5d,0x22,
Type:18,Body:\"nsu=302=151?1912594050601912594050624]\"
artifact_prefix='./'; Test unit written to ./timeout-7dbf1fa7fa53d6777a6e2052a71b72da775897d5
Base64: VHlwZToxOCxCb2R5OiJuc3U9MzAyPTE1MT8xOTEyNTk0MDUwNjAxOTEyNTk0MDUwNjI0XSI=
==11== ERROR: libFuzzer: timeout after 26 seconds
    #0 0x539271 in __sanitizer_print_stack_trace /src/llvm-project/compiler-rt/lib/asan/asan_stack.cpp:87:3
    #1 0x457d88 in fuzzer::PrintStackTrace() /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerUtil.cpp:210:5
    #2 0x43c9d9 in fuzzer::Fuzzer::AlarmCallback() /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:301:5
    #3 0x7f0a3cb9641f  (/lib/x86_64-linux-gnu/libpthread.so.0+0x1441f) (BuildId: 0c044ba611aeeeaebb8374e660061f341ebc0bac)
    #4 0x455f13 in __sanitizer_cov_trace_switch /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerTracePC.cpp:592:1
    #5 0x70fa0d in parse_expandednodeid /src/open62541/src/ua_types_lex.c:472:2
    #6 0x70fa0d in UA_ExpandedNodeId_parse /src/open62541/src/ua_types_lex.c:543:9
    #7 0x72eaa9 in ExpandedNodeId_decodeJson /src/open62541/src/ua_types_encoding_json.c:1987:16
    #8 0x72901f in decodeFields /src/open62541/src/ua_types_encoding_json.c
    #9 0x731fb6 in Variant_decodeJson /src/open62541/src/ua_types_encoding_json.c:2260:12
    #10 0x733072 in UA_decodeJson /src/open62541/src/ua_types_encoding_json.c:2790:11
    #11 0x56c729 in LLVMFuzzerTestOneInput /src/open62541/tests/fuzz/fuzz_json_decode.cc:24:28
    #12 0x43e003 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:611:15
    #13 0x43d7ea in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool, bool*) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:514:3
    #14 0x43eeb9 in fuzzer::Fuzzer::MutateAndTestOne() /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:757:19
    #15 0x43fb85 in fuzzer::Fuzzer::Loop(std::__Fuzzer::vector<fuzzer::SizedFile, std::__Fuzzer::allocator<fuzzer::SizedFile> >&) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:895:5
    #16 0x42eeef in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:912:6
    #17 0x458542 in main /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerMain.cpp:20:10
    #18 0x7f0a3c993082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca)
    #19 0x41f92d in _start (/out/fuzz_json_decode+0x41f92d)

DEDUP_TOKEN: __sanitizer_print_stack_trace--fuzzer::PrintStackTrace()--fuzzer::Fuzzer::AlarmCallback()
SUMMARY: libFuzzer: timeout
```

Figure 15: Error found while fuzzing open62541 with libFuzzer and fuzzing target, JSON decode



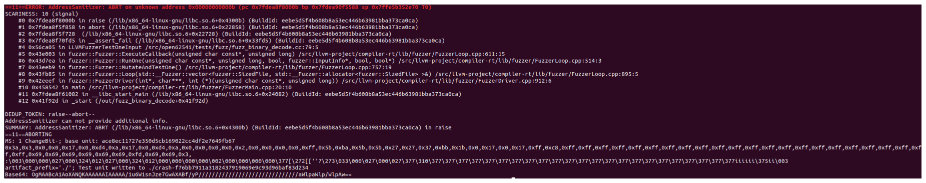
```
ALARM: working on the last Unit for 33 seconds
       and the timeout value is 25 (use -timeout=N to change)
MS: 1 InsertByte-; base unit: 2f1170d2344495c0b7ae255cb0d936d9abd08038
0x54,0x79,0x70,0x65,0x3a,0x31,0x38,0x2c,0x42,0x6f,0x64,0x79,0x3a,0x27,0x6e,0x73,0x75,0x3d,0x34,0x27,
Type:18,Body:'nsu=4'
artifact_prefix='./'; Test unit written to ./timeout-d7130062f8e231346b6f7a2861a3e63687a8fb2f
Base64: VHlwZToxOCxCb2R5Oiduc3U9NCc=
==11== ERROR: libFuzzer: timeout after 33 seconds
    #0 0x539271 in __sanitizer_print_stack_trace /src/llvm-project/compiler-rt/lib/asan/asan_stack.cpp:87:3
    #1 0x457d88 in fuzzer::PrintStackTrace() /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerUtil.cpp:210:5
    #2 0x43c9d9 in fuzzer::Fuzzer::AlarmCallback() /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:301:5
    #3 0x7f29d7ea741f  (/lib/x86_64-linux-gnu/libpthread.so.0+0x1441f) (BuildId: 0c044ba611aeeeaebb8374e660061f341ebc0bac)
    #4 0x455bbf in __sanitizer_cov_trace_const_cmp1 /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerTracePC.cpp:545:1
    #5 0x70fe5d in parse_expandednodeid /src/open62541/src/ua_types_lex.c:472:2
    #6 0x70fe5d in UA_ExpandedNodeId_parse /src/open62541/src/ua_types_lex.c:543:9
    #7 0x72eef9 in ExpandedNodeId_decodeJson /src/open62541/src/ua_types_encoding_json.c:1987:16
    #8 0x72946f in decodeFields /src/open62541/src/ua_types_encoding_json.c
    #9 0x732406 in Variant_decodeJson /src/open62541/src/ua_types_encoding_json.c:2260:12
    #10 0x7334c2 in UA_decodeJson /src/open62541/src/ua_types_encoding_json.c:2790:11
    #11 0x56c73b in LLVMFuzzerTestOneInput /src/open62541/tests/fuzz/fuzz_json_decode_encode.cc:22:28
    #12 0x43e003 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:611:15
    #13 0x43d7ea in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool, bool*) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:514:3
    #14 0x43eeb9 in fuzzer::Fuzzer::MutateAndTestOne() /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:757:19
    #15 0x43fb85 in fuzzer::Fuzzer::Loop(std::__Fuzzer::vector<fuzzer::SizedFile, std::__Fuzzer::allocator<fuzzer::SizedFile> >&) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:895:5
    #16 0x42eeef in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:912:6
    #17 0x458542 in main /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerMain.cpp:20:10
    #18 0x7f29d7ca4082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca)
    #19 0x41f92d in _start (/out/fuzz_json_decode_encode+0x41f92d)

DEDUP_TOKEN: __sanitizer_print_stack_trace--fuzzer::PrintStackTrace()--fuzzer::Fuzzer::AlarmCallback()
SUMMARY: libFuzzer: timeout
```

Figure 16: Error found while fuzzing open62541 with libFuzzer and fuzzing target, JSON decode encode



```
==16==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x626000000558 at pc 0x0000005e16f7 bp 0x7ffeb6cbc390 sp 0x7ffeb6cbc388
READ of size 4 at 0x626000000558 thread T0
SCARINESS: 27 (4-byte-read-heap-buffer-overflow-far-from-bounds)
    #0 0x5e16f6 in UA_Server_networkCallback /src/open62541/src/server/ua_server_binary.c:974:17
    #1 0x56c8bd in LLVMFuzzerTestOneInput /src/open62541/tests/fuzz/fuzz_binary_message.cc:64:5
    #2 0x43e003 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:611:15
    #3 0x43d7ea in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool, bool*) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:514:3
    #4 0x43f654 in fuzzer::Fuzzer::ReadAndExecuteSeedCorpora(std::__Fuzzer::vector<fuzzer::SizedFile, std::__Fuzzer::allocator<fuzzer::SizedFile> >&) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:826:7
    #5 0x43f889 in fuzzer::Fuzzer::Loop(std::__Fuzzer::vector<fuzzer::SizedFile, std::__Fuzzer::allocator<fuzzer::SizedFile> >&) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:857:3
    #6 0x42eeef in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:912:6
    #7 0x458542 in main /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerMain.cpp:20:10
    #8 0x7f1a205ef082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca)
    #9 0x41f92d in _start (/out/fuzz_binary_message+0x41f92d)

DEDUP_TOKEN: UA_Server_networkCallback--LLVMFuzzerTestOneInput--fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long)
Address 0x626000000558 is a wild pointer inside of access range of size 0x000000000004.
SUMMARY: AddressSanitizer: heap-buffer-overflow /src/open62541/src/server/ua_server_binary.c:974:17 in UA_Server_networkCallback
Shadow bytes around the buggy address:
  0x0c4c7fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4c7fff8060: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4c7fff8070: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4c7fff8080: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4c7fff8090: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c4c7fff80a0: fa fa fa fa fa fa fa fa fa fa fa[fa]fa fa fa fa
  0x0c4c7fff80b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4c7fff80c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4c7fff80d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4c7fff80e0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4c7fff80f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==16==ABORTING
MS: 0 ; base unit: 0000000000000000000000000000000000000000
0x48,0x45,0x4c,0x46,0x38,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0xff,0xff,0x0,0x0,0xff,0xff,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x18,0x0,0x0,0x0,0x6f,0x70,0x63,0x2e,0x74,0x63,0x70,0x3a,0x2f,0x2f,0x6c,0x6f,0x63,0x61,0x6c,0x68,0x6f,0x73,0x74,0x3a,0x34,0x38,0x34,0x30,0x0,0x0,0x80,0x0,
HELF8\000\000\000\000\000\000\000\377\377\000\000\377\377\000\000\000\000\000\000\000\000\000\000\030\000\000\000opc.tcp://localhost:4840\000\000\200\000
artifact_prefix='./'; Test unit written to ./crash-cc86090807f63c675be445037f5970a7a23a0c56
Base64: SEVMRjgAAAAAAAAA//8AAP//AAAAAAAAAAAABgAAABvcGMudGNwOi8vbG9jYWxob3N0OjQ4NDAAAIAA
```

Figure 17: Error found while fuzzing open62541 with libFuzzer and fuzzing target, binary message



```
==11==ERROR: AddressSanitizer: ABRT on unknown address 0x00000000000b (pc 0x7fdea8f8000b bp 0x7fdea90f5588 sp 0x7ffe5b352e70 T0)
SCARINESS: 10 (signal)
    #0 0x7fdea8f8000b in raise (/lib/x86_64-linux-gnu/libc.so.6+0x4300b) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca)
    #1 0x7fdea8f5f858 in abort (/lib/x86_64-linux-gnu/libc.so.6+0x22858) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca)
    #2 0x7fdea8f5f728  (/lib/x86_64-linux-gnu/libc.so.6+0x22728) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca)
    #3 0x7fdea8f70fd5 in __assert_fail (/lib/x86_64-linux-gnu/libc.so.6+0x33fd5) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca)
    #4 0x56ca05 in LLVMFuzzerTestOneInput /src/open62541/tests/fuzz/fuzz_binary_decode.cc:79:5
    #5 0x43e003 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:611:15
    #6 0x43d7ea in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool, bool*) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:514:3
    #7 0x43eeb9 in fuzzer::Fuzzer::MutateAndTestOne() /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:757:19
    #8 0x43fb85 in fuzzer::Fuzzer::Loop(std::__Fuzzer::vector<fuzzer::SizedFile, std::__Fuzzer::allocator<fuzzer::SizedFile> >&) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerLoop.cpp:895:5
    #9 0x42eeef in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerDriver.cpp:912:6
    #10 0x458542 in main /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerMain.cpp:20:10
    #11 0x7fdea8f61082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca)
    #12 0x41f92d in _start (/out/fuzz_binary_decode+0x41f92d)

DEDUP_TOKEN: raise--abort--
AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: ABRT (/lib/x86_64-linux-gnu/libc.so.6+0x4300b) (BuildId: eebe5d5f4b608b8a53ec446b63981bba373ca0ca) in raise
==11==ABORTING
MS: 1 ChangeBit-; base unit: ace8ec11727e350d5cb169022cc4df2e7649fb67
0x3a,0x3,0x0,0x0,0x17,0x0,0xd4,0xa,0x17,0x0,0xd4,0xa,0x0,0x0,0x0,0x0,0x2,0x0,0x0,0x0,0x0,0xff,0x5b,0xba,0x5b,0x5b,0x27,0x27,0x37,0xbb,0x1b,0x0,0x17,0x0,0x17,0xff,0xc8,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xf
f,0xff,0x69,0x69,0x69,0x69,0x69,0x69,0xfd,0x69,0x69,0x3,
:\003\000\000\027\000\324\012\027\000\324\012\000\000\000\000\002\000\000\000\000\377[\272[[''7\273\033\000\027\000\027\377\310\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377iiiiii\375ii\003
artifact_prefix='./'; Test unit written to ./crash-f76bb7911a3182437919069e9c93d96baf83d234
Base64: OgMAABcA1AoXANQKAAAAAAIAAAAA/1u6W1snJze7GwAXABf/yP////////////////////////////aWlpaWlp/WlpAw==
```

Figure 18: Error found while fuzzing open62541 with libFuzzer and fuzzing target, binary decode

D Source code

In this chapter, we show different source codes of the SUTs and the fuzzing targets.

D.1 fuzz_mdns_message.cc

```
#include <cstdint>
#include <cstdio>
#include <libmdnsd/mdnsd.h>
#include <cstring>
#include <cstdlib>

/*
** Main entry point.  The fuzzer invokes this function with each
** fuzzed input.
*/
extern "C" int
LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {


    struct message m;
    memset(&m, 0, sizeof(struct message));

    unsigned char *dataCopy = (unsigned char *)malloc(size);
    if (!dataCopy) {
        return 0;
    }

    memcpy(dataCopy, data, size);

    bool success = message_parse(&m, dataCopy, size);

    free(dataCopy);

    if (!success)
        return 0;

    mdns_daemon_t *d = mdnsd_new(QCLASS_IN, 1000);

    struct sockaddr_storage addr;
    mdnsd_in(d, &m, (struct sockaddr *)&addr, 2000);

    mdnsd_free(d);

    return 0;
}
```

Figure 19: The source code of the MDNS message fuzzing target

D.2 fuzz_binary_decode.cc

```
#include "custom_memory_manager.h"
#include <open62541/plugin/log_stdout.h>
#include <open62541/server_config_default.h>
#include <open62541/types.h>

/*
** Main entry point.  The fuzzer invokes this function with each
** fuzzed input.
*/
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t
    size) {
    if(size <= 6)
        return 0;

    // set the available memory
    if(!UA_memoryManager_setLimitFromLast4Bytes(data, size))
        return 0;

    data += 4;
    size -= 4;

    // get some random type
    uint16_t typeIndex = (uint16_t)(data[0] | data[1] << 8);
    data += 2;
    size -= 2;

    if(typeIndex >= UA_TYPES_COUNT)
        return UA_FALSE;

    void *dst = UA_new(&UA_TYPES[typeIndex]);
    if(!dst)
        return 0;

    const UA_ByteString binary = {
            size, //length
            (UA_Byte *) (void *) data
    };

    UA_StatusCode ret = UA_decodeBinary(&binary, dst, &UA_TYPES[
        typeIndex], NULL);
    if(ret != UA_STATUSCODE_GOOD) {
        UA_delete(dst, &UA_TYPES[typeIndex]);
        return 0;
    }
```

Figure 20: The first part of the source code of the binary decode fuzzing target

```c
    // copy the datatype to test
    void *dstCopy = UA_new(&UA_TYPES[typeIndex]);
    if(!dstCopy) {
        UA_delete(dst, &UA_TYPES[typeIndex]);
        return 0;
    }
    ret = UA_copy(dst, dstCopy, &UA_TYPES[typeIndex]);
    if(ret != UA_STATUSCODE_GOOD) {
        UA_delete(dst, &UA_TYPES[typeIndex]);
        UA_delete(dstCopy, &UA_TYPES[typeIndex]);
        return 0;
    }

    // compare with copy
    UA_assert(UA_order(dst, dstCopy, &UA_TYPES[typeIndex]) ==
        UA_ORDER_EQ);
    UA_delete(dstCopy, &UA_TYPES[typeIndex]);

    // now also test encoding
    size_t encSize = UA_calcSizeBinary(dst, &UA_TYPES[typeIndex]);
    UA_ByteString encoded;
    ret = UA_ByteString_allocBuffer(&encoded, encSize);
    if(ret != UA_STATUSCODE_GOOD) {
        UA_delete(dst, &UA_TYPES[typeIndex]);
        return 0;
    }

    ret = UA_encodeBinary(dst, &UA_TYPES[typeIndex], &encoded);
    UA_assert(ret == UA_STATUSCODE_GOOD);

    UA_ByteString_clear(&encoded);
    UA_delete(dst, &UA_TYPES[typeIndex]);
    return 0;
}
```

Figure 21: The second part of the source code of the binary decode fuzzing target

```c
    // check if the message has the correct size, i.e. the count matches the number of bytes

        /* Process questions */
        my(m->qd, (sizeof(struct question) * m->qdcount), struct question *);
        for (i = 0; i < m->qdcount; i++) {
                if (!_label(m, &buf, m->_bufEnd, &(m->qd[i].name))) {
            return false;
        }
                if (buf + 4 > m->_bufEnd) {
            return false;
        }
                m->qd[i].type  = net2short(&buf);
                m->qd[i].clazz = net2short(&buf);
        }
    if (buf > m->_bufEnd) {
        return false;
    }

        /* Process rrs */
        my(m->an, (sizeof(struct resource) * m->ancount), struct resource *);
        my(m->ns, (sizeof(struct resource) * m->nscount), struct resource *);
        my(m->ar, (sizeof(struct resource) * m->arcount), struct resource *);
        if (!_rrparse(m, m->an, m->ancount, &buf, m->_bufEnd))
                return false;
        if (!_rrparse(m, m->ns, m->nscount, &buf, m->_bufEnd))
                return false;
        if (!_rrparse(m, m->ar, m->arcount, &buf, m->_bufEnd))
                return false;
        return true;
}
```

Figure 22: The part of the source code where the rrparse function is called with the wrong size