Identify Memory Corruption Bugs using Fuzzing

Asmita
Electrical and Computer Engineering
University of California, Davis
Davis, California
aasmita@ucdavis.edu

Sidharth Nagendran
Electrical and Computer Engineering
University of California, Davis
Davis, California
snagendran@ucdavis.edu

Antony Leo Joan Balthasar
Electrical and Computer Engineering
University of California, Davis
Davis, California
abalthasar@ucdavis.edu

***Abstract*—Software security validation is one of the most important aspects of any system. There are several existing solutions when it comes to generic software quality validation, but identifying issues with respect to security is still a challenge. The challenge grows by many folds when it comes to embedded firmware security validation. The existing tools developed for firmware analysis and verification has a very higher learning curve making it difficult for a feasible adoption. Though, apart from the higher learning curve , there are several other issues like scalability, efficiency that makes difficult for adoption of these tools in industry. As part of this project, the learning curve challenge for adoption of these tools have been targeted. The objective is the development of an automated framework for identification of memory bugs. The adopted method is hybrid combination of both fuzzing and symbolic execution using existing tools Symcc and AFL++. The associated challenges and improvements have also been discussed.**

***Index Terms*—Fuzzing, Software vulnerabilities, Symbolic execution**

I. Introduction

Softwares are prone to several memory related bugs that lead to exploitation like remote code execution, returned oriented programming (ROP), command injection, etc. Detection of these vulnerabilities are challenging in terms of resource, efforts, higher learning curve, scalability and efficiency of existing tools. Embedded firmwares that interact with low level hardware get more challenging in terms of security vulnerability detection. In this project, state of art existing tools that have potential to get adopted for embedded firmware have been targeted. The major objective is to have an automated tool leveraging the feature of existing tools such that it could be feasible to be used with lesser learning curve. The memory corruption bugs have been targeted.

The approach taken for this project is hybrid combination of symbolic execution [8] and fuzzing [9]. Symbolic Execution is becoming increasingly popular in program testing. Research over the past few decades has steadily improved the design and increased the performance of available implementations. Fuzzing is the art of automatic bug detection. The goal of fuzzing is to stress the application and cause unexpected behaviour, resource leaks or crashes. This process involves throwing invalid, unexpected or random data as inputs at a computer.

The reason to combine both symbolic execution and fuzzing is that the symbolic execution has a reputation of being a highly effective yet expensive technique to explore programs. It is often combined with fuzz testing such as hybrid fuzzing [9] where the fuzzer leverages heuristics to explore relatively easy-to-reach paths quickly, while symbolic execution [8] contributes test cases that reach the more difficult-to-explore parts of the target program.

The expected use case for this project is Memory corruption bugs, The graphs figure 1 and figure 2 is obtained from nvd.nist.gov. These graphs show the recent trend of memory corruption bugs as hardware vulnerabilities. The figure 1 symbolises the total matches by year and figure 2 symbolises the percent matches by year.

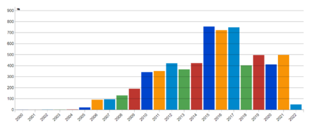


Fig. 1. Total Matches by Year

According to the data obtained there has been a significant increase in the number of memory corruption vulnerabilities in the recent years. So this project helps in identifying the memory corruption bugs.

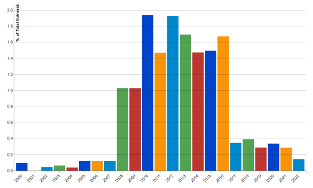


Fig. 2. Percent Matches by Year

II. Background and Existing Tools

A. Fuzzing

Fuzzing, also known as fuzz testing [9], is an automated software testing approach that includes feeding a computer program with erroneous, unexpected, or unpredictable data. The application is then checked for exceptions like as crashes, failed built-in code assertions, and suspected memory leaks. Fuzzers are typically used to evaluate programs that accept structured inputs. This structure has been specified.

An effective fuzzer provides semi-valid inputs that are "valid enough" in the sense that they are not explicitly rejected by the parser but do cause unexpected behaviors deeper in the program and are ”invalid enough” to disclose corner cases that have not been effectively addressed. Fuzzing is an intrinsically randomized procedure that adds a large number of variables that are beyond one’s control.

B. Symbolic execution

Symbolic execution [8] is a method of abstractly running a program, so that one abstract execution covers several alternative program inputs that share a common execution route through the code. The execution handles these inputs symbolically, ”returning” a result stated in terms of symbolic constants that reflect the input values.

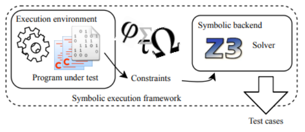


Fig. 3. Building Block of Symbolic Execution

The block diagram of Symbolic Execution is depicted in figure 3. Symbolic Execution does not investigate all pathways through the program, it is less general than abstract interpretation. Symbolic execution [8], on the other hand, may frequently avoid approximation in situations where AI must approximate in order to achieve analysis termination. This implies that symbolic execution can avoid issuing misleading warnings; each fault discovered by symbolic execution reflects a genuine, possible route through the program, and can be verified with a test case that demonstrates the error.

C. Symcc

Symcc is the concept of compiling symbolic handling into target programs [11]. It can only be used when the source code is accessible, and hence does not support binary analysis.

There are four different parts to Symcc

1. Execution: The test program is run, and the system generates symbolic expressions that reflect the computations. These phrases are critical for reasoning about the program.

2. Symbolic Backend: The main aim of symbolically representing computations is to reason about them. The symbolic backend is made up of the components engaged in the reasoning process.

3. Forking and Scheduling: Some symbolic execution systems run the target program only once, maybe along the route defined by a particular program input, and then create additional program inputs based on that single execution.

D. AFL++

American fuzzy lop (AFL) is a free software fuzzer that uses evolutionary algorithms to effectively enhance test case code coverage [4]. The tool needs the user to supply a sample command that executes the tested application as well as at least one tiny example input file.

American fuzzy lop anticipates that the tested program will be constructed with the assistance of a utility program that equips the code with helper functions that follow control flow in order to enhance fuzzing speed [4]. This enables the fuzzer to identify changes in the target’s behavior in response to the input. In the event that this is not feasible, black-box testing is also supported.

E. Driller

Driller is a guided whitebox fuzzer that adds concolic execution to provide effective vulnerability excavation on top of cutting-edge fuzzing approaches [15]. It is an angr-based hybrid fuzzer, comparable to QSYM but slower due to its Python implementation and interpreter-based approach. It employs a more complex technique to coordinate the fuzzer and symbolic executor, checks the progress of the fuzzer, and switches to symbolic execution.

F. Symqemu

SymQEMU is a QEMU CPU emulation that may be paired with a relatively lightweight symbolic execution method, rather than a computationally expensive translation of the target program to an intermediate representation that is then symbolically interpreted [12]. SymQEMU explains how to get comparable performance increases in a binary-only situation while adhering to all of the extra requirements.

G. Memory corruption bugs

1) Stack based buffer overflow: Stack buffer overflow [13] problems occur when a software writes more data to a stack buffer than is actually allocated for that buffer [2]. This nearly invariably leads to the corruption of neighboring data on the stack, which can cause software crashes, erroneous behavior, or security concerns.

2) Heap based buffer overflow: A heap overflow condition is a buffer overflow [10] in which the buffer that can be overwritten is allocated in the heap section of memory, which normally means that the buffer was allocated using a malloc function. A heap overflow, also known as a heap overrun or heap smashing, is a form of buffer overflow that happens in the heap data space. Heap overflows are vulnerable in a different way than stack overflows [2]. At runtime, heap memory is dynamically allocated and often holds program data. The exploit is carried out by corrupting this data in precise ways, causing the program to rewrite internal structures such linked list pointers.

3) Integer overflow: An integer overflow happens when you attempt to store a value in an integer variable [16] that is greater than the variable’s maximum value. This is referred to as undefined behavior in the C standard. In reality, this generally translates to a value wrap if an unsigned integer was used and a sign and value change [2] if a signed integer was used. Integer overflows occur as a result of ”wild” increments/multiplications, which are typically caused by a lack of validation of the variables involved.

4) Improper data sanitization: In cases, where external input is accepted from external users, it gives the low hanging path for attackers to insert malicious payload. This exploitation is possible when the data being accepted is not sanitize before being processed. By data sanitizing, it means filtering the data/character at the input that could be sensitive to trigger privilege commands / paths in the system.

5) Out of bound reads: The software reads data past the end, or before the beginning, of the intended buffer. In most cases, this allows attackers to retrieve sensitive data from other memory regions or trigger a crash. A crash can occur when the code reads a variable quantity of data and thinks that a sentinel, such as a NULL in a string, exists to end the read process. If the expected sentinel is not found in out-of-bounds memory, extra data is read, resulting in a segmentation fault or a buffer overflow. The software may change an index or execute pointer arithmetic that refers to a memory address that is outside the buffer's bounds. Following that, a read operation yields undefined or unexpected results.

6) Null dereference: A null-pointer dereference [1] takes place when a pointer with a value of NULL is used as though it pointed to a valid memory area. While null-pointer dereferences are widespread, they can usually be discovered and fixed in a straightforward manner. They will always cause the process to crash until exception handling (on some platforms) is initiated, and even then, there is nothing that can be done to save the process.

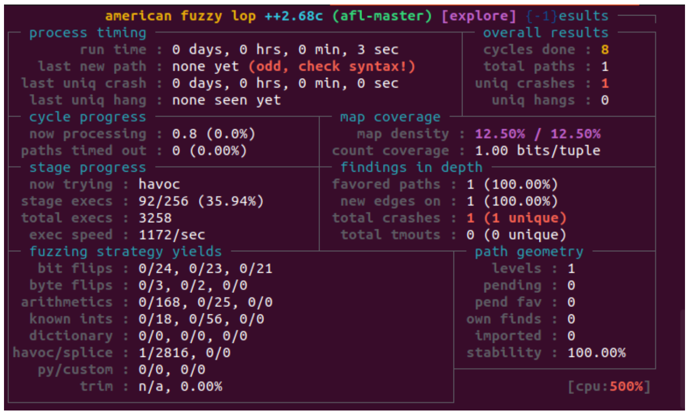
III. Implementation and Demo

The implementation is highly inspired by Symcc and AFL++. Initially, just the fuzzing approach was taken for bug identification that didn't result in better outcome. It has been covered in more detail in the challenges section. In order to overcome the problem, symbolic execution is combined using symcc tool. The implementation involved a lot of initial learning curve and setup efforts with respect to these tools. The automation part is done using Python.

A. AFL++ and SymCC Implementation

The initial background setup involved installation of dependencies related of AFL++ and Symcc as mentioned on their github pages [14] [5]. The software under test is translated to an intermediate representation LLVM bitcode. Hence, the latest version of LLVM and clang was installed. During the compilation process, symbolic processing is embedded into the target program. As it work's on the intermediate representation of the compiler, it makes it adaptable to different programming languages. The LLVM bitcode of the targeted program is taken and the symbolic execution capabilities is compiled into the binary. During the program execution, the path to be taken by symbolic execution depends on the fuzzer that helps it to prioritize the test cases. During testing, the source code is compiled using symcc based compiler instead of normal compiler that helps in binary instrumentation. Symbolic execution generates programe input that triggers a certain security vulnerability, hence it involves reasoning process. It uses SMT solver Z3 [3] in the background.

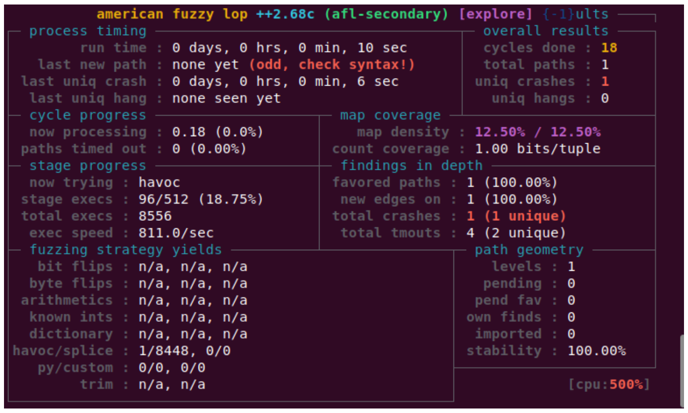
The feature of symbolic execution is combined with AFL++ that is the state-of-art fuzzing approach. It's a mutation based coverage guided fuzzer. The set of test cases are mutated to identify unexplored points in the program. When such points are identified, the test case that triggered it is saved as a test case queue. The two categories of mutation that are used are havoc and deterministic. Single deterministic mutation is involved on the test cases (for example bit flips) in case of deterministic approach. In case of havoc, mutations are randomly done. In the hybrid based approach that's implemented here, AFL++ fuzzer is assisted using Symcc to generate new test cases efficiently. AFL++ is used in parallel mode with master and secondary instances as shown in figure 4 and figure 5.



```
           american fuzzy lop ++2.68c (afl-master) [explore] {-1}esults ──┐
┌─ process timing ──────────────────────────────────┬─ overall results ──┐
│        run time : 0 days, 0 hrs, 0 min, 3 sec      │  cycles done : 8   │
│   last new path : none yet (odd, check syntax!)    │  total paths : 1   │
│ last uniq crash : 0 days, 0 hrs, 0 min, 0 sec      │ uniq crashes : 1   │
│  last uniq hang : none seen yet                    │   uniq hangs : 0   │
├─ cycle progress ────────────────┬─ map coverage ─┴────────────────────┤
│  now processing : 0.8 (0.0%)     │    map density : 12.50% / 12.50%     │
│ paths timed out : 0 (0.00%)      │ count coverage : 1.00 bits/tuple     │
├─ stage progress ────────────────┼─ findings in depth ─────────────────┤
│  now trying : havoc              │ favored paths : 1 (100.00%)          │
│ stage execs : 92/256 (35.94%)    │  new edges on : 1 (100.00%)          │
│ total execs : 3258               │ total crashes : 1 (1 unique)         │
│  exec speed : 1172/sec           │  total tmouts : 0 (0 unique)         │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ───────┤
│   bit flips : 0/24, 0/23, 0/21                   │    levels : 1         │
│  byte flips : 0/3, 0/2, 0/0                      │   pending : 0         │
│ arithmetics : 0/168, 0/25, 0/0                   │  pend fav : 0         │
│  known ints : 0/18, 0/56, 0/0                    │ own finds : 0         │
│  dictionary : 0/0, 0/0, 0/0                      │  imported : 0         │
│havoc/splice : 1/2816, 0/0                        │ stability : 100.00%   │
│  py/custom : 0/0, 0/0                            ├───────────────────────┘
│        trim : n/a, 0.00%                         │          [cpu:500%]
└──────────────────────────────────────────────────┘
```

Fig. 4. AFL Master instance fuzzing

The targeted programme is compiled using both symcc and afl-clang-fast compiler. The executable compiled using afl-clang is fed for the fuzzing using AFL++.



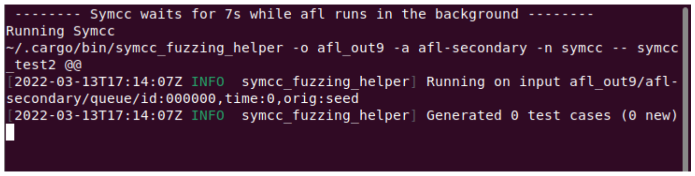
```
         american fuzzy lop ++2.68c (afl-secondary) [explore] {-1}ults ──┐
┌─ process timing ──────────────────────────────────┬─ overall results ──┐
│        run time : 0 days, 0 hrs, 0 min, 10 sec     │  cycles done : 18  │
│   last new path : none yet (odd, check syntax!)    │  total paths : 1   │
│ last uniq crash : 0 days, 0 hrs, 0 min, 6 sec      │ uniq crashes : 1   │
│  last uniq hang : none seen yet                    │   uniq hangs : 0   │
├─ cycle progress ────────────────┬─ map coverage ─┴────────────────────┤
│  now processing : 0.18 (0.0%)    │    map density : 12.50% / 12.50%     │
│ paths timed out : 0 (0.00%)      │ count coverage : 1.00 bits/tuple     │
├─ stage progress ────────────────┼─ findings in depth ─────────────────┤
│  now trying : havoc              │ favored paths : 1 (100.00%)          │
│ stage execs : 96/512 (18.75%)    │  new edges on : 1 (100.00%)          │
│ total execs : 8556               │ total crashes : 1 (1 unique)         │
│  exec speed : 811.0/sec          │  total tmouts : 4 (2 unique)         │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ───────┤
│   bit flips : n/a, n/a, n/a                      │    levels : 1         │
│  byte flips : n/a, n/a, n/a                      │   pending : 0         │
│ arithmetics : n/a, n/a, n/a                      │  pend fav : 0         │
│  known ints : n/a, n/a, n/a                      │ own finds : 0         │
│  dictionary : n/a, n/a, n/a                      │  imported : 0         │
│havoc/splice : 1/8448, 0/0                        │ stability : 100.00%   │
│  py/custom : 0/0, 0/0                            ├───────────────────────┘
│        trim : n/a, n/a                           │          [cpu:500%]
└──────────────────────────────────────────────────┘
```

Fig. 5. AFL Secondary instance fuzzing

After a few minutes of execution of master and secondary instances of fuzzer, symbolic execution is started using symcc_fuzzing_helper, as shown in figure 6

B. AFL++ and SymCC Execution Automation

The proposed implementation has 4 steps and they are,



```
-------- Symcc waits for 7s while afl runs in the background --------
Running Symcc
~/.cargo/bin/symcc_fuzzing_helper -o afl_out9 -a afl-secondary -n symcc -- symcc
_test2 @@
[2022-03-13T17:14:07Z INFO  symcc_fuzzing_helper] Running on input afl_out9/afl-
secondary/queue/id:000000,time:0,orig:seed
[2022-03-13T17:14:07Z INFO  symcc_fuzzing_helper] Generated 0 test cases (0 new)
```

Fig. 6. SymCC execution using symcc_fuzzing_helper

1. Initiate the AFL-Master to perform the deterministic checks
2. Initiate the AFL-Secondary to perform random tweaks parallel to AFL-Master
3. Wait for a certain time and initiate SymCC for symbolic execution of source code
4. Perform GDB back tracing to detect the part of code that leads to a crash.

Each step had to be manually initiated and the had to wait to a certain amount of time to trigger the next step. This process was found to be cumbersome and time consuming. Hence to mitigate this, a python script was developed to execute the shell commands to trigger each step and thereby automating the process. The algorithm used to develop these scripts is shown in figure 7.

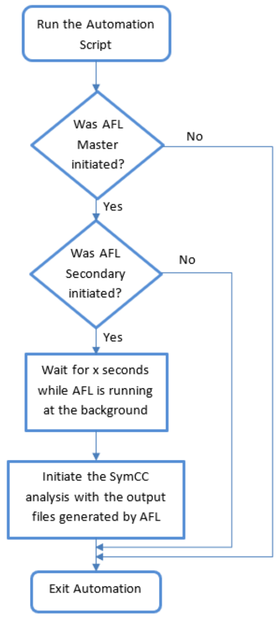

Run the Automation Script
Was AFL Master initiated?
No
Yes
Was AFL Secondary initiated?
No
Yes
Wait for x seconds while AFL is running at the background
Initiate the SymCC analysis with the output files generated by AFL
Exit Automation

Fig. 7. Algorithm of AFL++ and SymCC execution automation script

The script takes the path to the source file, path to afl-clang.c, path to symcc.c and wait time for SymCC execution as the input arguments. With the help of these it will trigger the AFL master fuzzing instance and AFL secondary fuzzing instance and allows them to run simultaneously. Then the algorithm will wait for a specific duration given by the user as one of the input arguments. SymCC will use the output files generated by AFL instances for its analysis, so it is advised to wait for a specific duration before triggering the SymCC execution. After the wait time, the SymCC execution will be triggered which will accelerate the crash detection by bring in symbolic execution algorithm along with fuzzing.

C. GDB Automation

After the completion of SymCC and AFL++ analysis, it is imperative to use the crash data files to trace back the part of source code which lead to that crash. GDB is one such tool which helps to understand what happens inside the source code while a crash happens. In this implementation GDB tool has been integrated with a python script to automate the process of crash analysis. The algorithm of this script is shown in figure 8

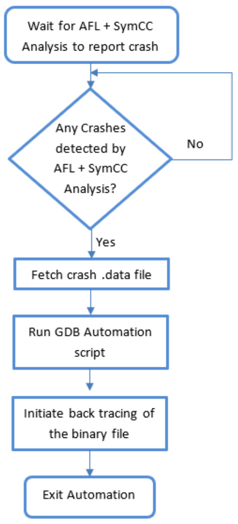
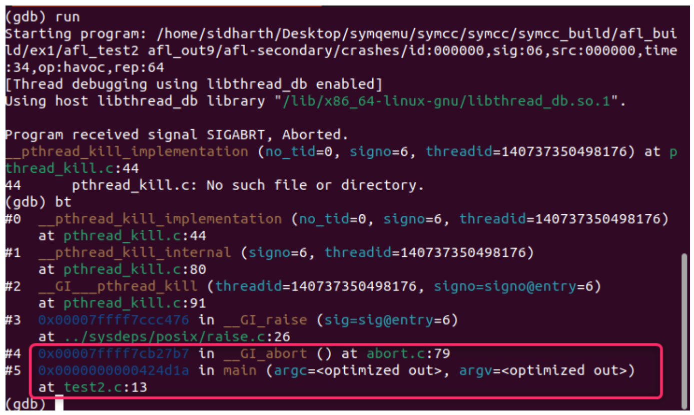

Wait for AFL + SymCC Analysis to report crash
Any Crashes detected by AFL + SymCC Analysis?
No
Yes
Fetch crash .data file
Run GDB Automation script
Initiate back tracing of the binary file
Exit Automation

Fig. 8. Algorithm of GDB automation script

The AFL++ tool will store the value of inputs for which the crash was detected. GDB automation script accepts the path of crash .data file as its input argument. With the help of this file, GDB will be able to back trace to the part of code which lead to the crash as shown in figure 9.



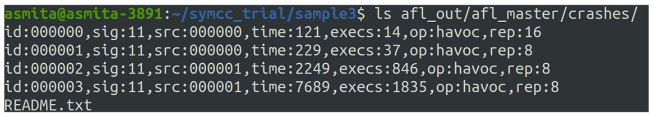
```
(gdb) run
Starting program: /home/sidharth/Desktop/symqemu/symcc/symcc/symcc_build/afl_bui
ld/ex1/afl_test2 afl_out9/afl-secondary/crashes/id:000000,sig:06,src:000000,time
:34,op:havoc,rep:64
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGABRT, Aborted.
__pthread_kill_implementation (no_tid=0, signo=6, threadid=140737350498176) at p
thread_kill.c:44
44      pthread_kill.c: No such file or directory.
(gdb) bt
#0  __pthread_kill_implementation (no_tid=0, signo=6, threadid=140737350498176)
    at pthread_kill.c:44
#1  __pthread_kill_internal (signo=6, threadid=140737350498176)
    at pthread_kill.c:80
#2  __GI___pthread_kill (threadid=140737350498176, signo=signo@entry=6)
    at pthread_kill.c:91
#3  0x00007ffff7ccc476 in __GI_raise (sig=sig@entry=6)
    at ../sysdeps/posix/raise.c:26
#4  0x00007ffff7cb27b7 in __GI_abort () at abort.c:79
#5  0x0000000000424d1a in main (argc=<optimized out>, argv=<optimized out>)
    at test2.c:13
(gdb) 
```

Fig. 9. Results of GDB Back Tracing Analysis

D. Triaging Automation

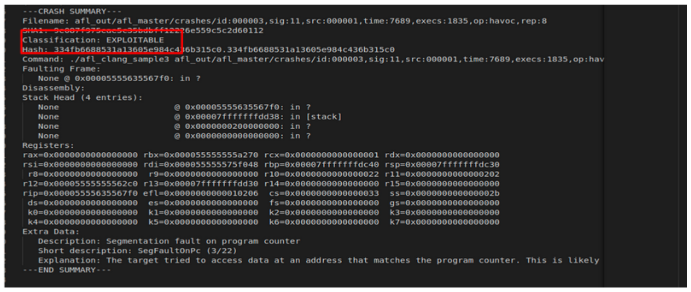
The outcome after the hybrid application of symbolic execution and fuzzing provides with all the possible crashes as shown in figure 10.



```
asmita@asmita-3891:~/symcc_trial/sample3$ ls afl_out/afl_master/crashes/
id:000000,sig:11,src:000000,time:121,execs:14,op:havoc,rep:16
id:000001,sig:11,src:000000,time:229,execs:37,op:havoc,rep:8
id:000002,sig:11,src:000001,time:2249,execs:846,op:havoc,rep:8
id:000003,sig:11,src:000001,time:7689,execs:1835,op:havoc,rep:8
README.txt
```

Fig. 10. Crashes generated after fuzzing

In case of larger programs with more number of crashes it involves a lot of manual effort to triage through each of the crashes and analyse to identify if they are exploitable or not and figure out what part of the code caused the suspected issues. The outcome of fuzzing is not very self-explanatory, there is a lot of manual analysis required to identify the actual cause of vulnerability and to test if that is exploitable. As part of this project, a small step towards automation of triage has been attempted using a tool called crashwalk [6]. It traverses through the the crashes generated by the fuzzer, and debugs the program using each input that caused the crash. After getting the crash, crashwalk analyses the program status using debugger. Each test case is then bucketed based on the hash of the program back trace. This helps to analyse that the test cases inside same bucket followed the same execution path. Apart from this, it performs automated analysis using exploitable GDB plugin that analyzes crashes to determine the type of bug that caused the crash and how likely is it to be exploitable as shown in figure 11. Thus, it helps to prioritize the bugs that should be verified in depth for further verification.



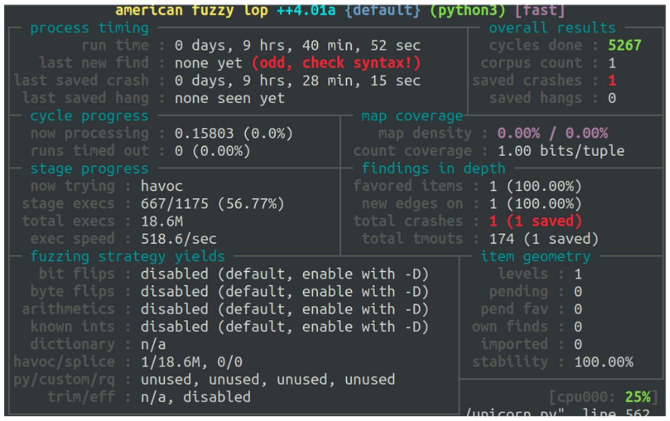
```
---CRASH SUMMARY---
Filename: afl_out/afl_master/crashes/id:000003,sig:11,src:000001,time:7689,execs:1835,op:havoc,rep:8
SHA1: 9c807f975cac5e35bdbff12226e559c5c2d60112
Classification: EXPLOITABLE
Hash: 334fb6688531a13605e984c436b315c0.334fb6688531a13605e984c436b315c0
Command: ./afl_clang_sample3 afl_out/afl_master/crashes/id:000003,sig:11,src:000001,time:7689,execs:1835,op:hav
Faulting Frame:
   None @ 0x00005555635567f0: in ?
Disassembly:
Stack Head (4 entries):
   None                        @ 0x00005555635567f0: in ?
   None                        @ 0x00007fffffffdd38: in [stack]
   None                        @ 0x0000000200000000: in ?
   None                        @ 0x0000000000000000: in ?
Registers:
rax=0x0000000000000000 rbx=0x000055555555a270 rcx=0x0000000000000001 rdx=0x0000000000000000
rsi=0x0000000000000000 rdi=0x000055555575f048 rbp=0x00007fffffffdc40 rsp=0x00007fffffffdc30
 r8=0x0000000000000000  r9=0x0000000000000000 r10=0x0000000000000022 r11=0x0000000000000202
r12=0x00005555555562c0 r13=0x00007fffffffdd30 r14=0x0000000000000000 r15=0x0000000000000000
rip=0x00005555635567f0 efl=0x0000000000010206  cs=0x0000000000000033  ss=0x000000000000002b
 ds=0x0000000000000000  es=0x0000000000000000  fs=0x0000000000000000  gs=0x0000000000000000
 k0=0x0000000000000000  k1=0x0000000000000000  k2=0x0000000000000000  k3=0x0000000000000000
 k4=0x0000000000000000  k5=0x0000000000000000  k6=0x0000000000000000  k7=0x0000000000000000
Extra Data:
   Description: Segmentation fault on program counter
   Short description: SegFaultOnPc (3/22)
   Explanation: The target tried to access data at an address that matches the program counter. This is likely
---END SUMMARY---
```

Fig. 11. Automated triage to show the likelihood of bug exploitability

There are other approaches that can be incorporated to make the automation better. This includes record and replay feature using PANDA [7], exploration mode supported by AFL via afl-min that is used to reduce the fuzzing corpus size.

IV. Challenges and Failures

The project phase dealt with several challenges including the initial time consumption and efforts required for the dependencies setup related to existing tools. Apart from that, in cases of real world firmwares, fuzzing took a very long time and considerable resource. Despite of running our fuzzer for more that 48 hours, just one crash was detected. As shown in figure 12, this test was performed against Tenda router firmware, but despite of running 48 hours, the output showed same 1 crash that was shown after 9 hours of execution.



```
            american fuzzy lop ++4.01a {default} (python3) [fast]
┌─ process timing ────────────────────────────────────┬─ overall results ──┐
│        run time : 0 days, 9 hrs, 40 min, 52 sec      │  cycles done : 5267 │
│   last new find : none yet (odd, check syntax!)      │ corpus count : 1    │
│last saved crash : 0 days, 9 hrs, 28 min, 15 sec      │saved crashes : 1    │
│ last saved hang : none seen yet                      │  saved hangs : 0    │
├─ cycle progress ─────────────────────┬─ map coverage┴────────────────────┤
│  now processing : 0.15803 (0.0%)     │    map density : 0.00% / 0.00%     │
│  runs timed out : 0 (0.00%)          │ count coverage : 1.00 bits/tuple   │
├─ stage progress ─────────────────────┼─ findings in depth ───────────────┤
│  now trying : havoc                  │ favored items : 1 (100.00%)       │
│ stage execs : 667/1175 (56.77%)      │  new edges on : 1 (100.00%)       │
│ total execs : 18.6M                  │ total crashes : 1 (1 saved)       │
│  exec speed : 518.6/sec              │  total tmouts : 174 (1 saved)     │
├─ fuzzing strategy yields ────────────┴───────────┬─ item geometry ───────┤
│   bit flips : disabled (default, enable with -D) │    levels : 1         │
│  byte flips : disabled (default, enable with -D) │   pending : 0         │
│ arithmetics : disabled (default, enable with -D) │  pend fav : 0         │
│  known ints : disabled (default, enable with -D) │ own finds : 0         │
│  dictionary : n/a                                │  imported : 0         │
│havoc/splice : 1/18.6M, 0/0                       │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused     ├───────────────────────┘
│    trim/eff : n/a, disabled                      │          [cpu000: 25%]
└──────────────────────────────────────────────────┘/unicorn.py", line 562
```

Fig. 12. Fuzzing challenge, just 1 crash

Apart from the time and resource involved from fuzzing, one of the major hurdles was setting up the real world embedded firmware to make it work for the fuzzing task. In such cases, the access to the actual hardware might not be possible, neither the source code. There is just the access to the firmware binary that needs to be successfully emulated before starting the fuzzing part. During the implementation phase, this was the most struggling point that consumed the majority of time but still failed. The reason is that every embedded devices have their respective customized implementation that makes it very difficult for automation. As in this case, though the emulation for a particular version series of Tenda router was successful, but when switched to different version or different router, the implemented framework failed, things had to be done from scratch that was customized to that particular device and firmware version. Hence, it's challenging when applied to real world firmwares.

V. Future Works

There are several incremental stages that are needed to be worked upon to make this tool aligned towards embedded firmwares. The current state of art tool is generic towards different kinds of softwares, the further work is required to make it more efficient by customizing it with respect to embedded firmwares. In order to achieve this, milestones include getting in-depth source code understanding of exiting state of art tools like AFL++, Qemu, Symcc; applying the implemented framework towards CTFs (capture the flags) for example Darpa Grand Challenge, analyze the drawback and then improve the design accordingly with the required modification in the existing tools. Further, extending it towards real world embedded firmwares such that we have a reliable and scalable tool for embedded firmware vulnerability detection.

References

[1] Nathaniel Ayewah and William Pugh. Null dereference analysis in practice. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop*

on Program analysis for software tools and engineering, pages 65–72, 2010.

[2] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE, 2000.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] 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.

[5] https://github.com/AFLplusplus/AFLplusplus. Aflplusplus.

[6] https://github.com/bnagy/crashwalk. Crashwalk.

[7] https://github.com/panda re/panda. Panda.

[8] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[9] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[10] Maryam Mouzarani, Babak Sadeghiyan, and Mohammad Zolfaghari. A smart fuzzing method for detecting heap-based buffer overflow in executable codes. In *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 42–49. IEEE, 2015.

[11] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198. USENIX Association, August 2020.

[12] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *Proceedings of the 2021 Network and Distributed System Security Symposium*, 2021.

[13] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[14] https://github.com/eurecom-s3/symcc S3 group at Eurecom. Symcc.

[15] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[16] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Citeseer, 2009.

APPENDIX

In this appendices we give the code created for the automation of the AFL++ SymCC execution, crash analysis using GDB and Triage automation.

A. Code for AFL Master instance

```
import os
import sys
import subprocess
#----getting the paths to required files
f_source = sys.argv[1]
f_afl = sys.argv[2]
f_symcc = sys.argv[3]
#----stripping the source code filename
f_source_basename =
        os.path.basename(f_source)
f_afl_basename = os.path.basename(f_afl)
#----creating path names
f_bin = f_source_basename.split(".")
f_afl_bin = 'afl_'+f_bin[0]
f_symcc_bin = 'symcc_'+f_bin[0]
f_afl_fuzz =
    f_afl.replace('/'+f_afl_basename,'')
print(f_afl)
print(f_afl_basename)
print(f_afl_fuzz)
#----checking the paths
if not(os.path.exists(f_source)):
    print("Source path does not exist")
if not(os.path.exists(f_afl)):
    print("afl_clang path does not exist")
if not(os.path.exists(f_symcc)):
    print("symcc path does not exist")
cmd1 = "sudo rm -r afl_out1/"
os.system(cmd1)
#exporting path
cmd6 = "export PATH=$PATH:" + "f_afl_fuzz"
os.system(cmd6)
#----creating afl_clang binary file
cmd2 = f_afl + " -O0 " +
        f_source_basename + " -o " +
        f_afl_bin
os.system(cmd2)
#----creating symcc binary file
cmd3 = f_symcc + " -O0 " +
        f_source_basename + " -o " +
        f_symcc_bin
os.system(cmd3)

#----calling afl_secondary in new shell
cmd5 = "gnome-terminal -x sh -c " +
        "\"python automation_secondary.py
        {source, afl-clang, symcc}; bash\""
os.system(cmd5)
#----runnign afl master instance
cmd4 = f_afl_fuzz + "/afl-fuzz " +
        "-M afl-master -i corpus/
        -o afl_out10 -m none-- " +
        " ./" + f_afl_bin +" @@"
os.system(cmd4)
```

B. Code for AFL Secondary Instance

```
import os
import sys
import subprocess
#----getting the paths to required files
f_source = sys.argv[1]
f_afl = sys.argv[2]
f_symcc = sys.argv[3]
#----stripping the source code filename
f_source_basename =
        os.path.basename(f_source)
f_afl_basename = os.path.basename(f_afl)
#----creating path names
f_bin = f_source_basename.split(".")
f_afl_bin = 'afl_'+f_bin[0]
```

```
f_symcc_bin = 'symcc_'+f_bin[0]
f_afl_fuzz =
        f_afl.replace('/'+f_afl_basename,'')
#----checking the paths
if not(os.path.exists(f_source)):
    print("Source path does not exist")
if not(os.path.exists(f_afl)):
    print("afl_clang path does not exist")
if not(os.path.exists(f_symcc)):
    print("symcc path does not exist")
#----exporting path
cmd1 = "export PATH=$PATH:" + "f_afl_fuzz"
os.system(cmd1)
#----calling symcc in a new terminal
cmd5 = "gnome-terminal -x sh -c " +
        "\"python automation_symcc.py 10
        {source, afl-clang, symcc}; bash\""
print(cmd5)
os.system(cmd5)
#----runnign afl secondary instance
cmd2 = f_afl_fuzz + "/afl-fuzz " + "
    -S afl-secondary -i corpus/
    -o afl_out10 -m none -- " +
    " ./" + f_afl_bin + " @@"
print(cmd2)
os.system(cmd2)
```

C. Code for Symcc execution

```
import os
import sys
import subprocess
#----getting the paths to required files
sleep_time = sys.argv[1]
f_source = sys.argv[2]
f_afl = sys.argv[3]
f_symcc = sys.argv[4]
#----stripping the source code filename
f_source_basename =
    os.path.basename(f_source)
f_afl_basename = os.path.basename(f_afl)
#----creating path names
f_bin = f_source_basename.split(".")
f_afl_bin = 'afl_'+f_bin[0]
f_symcc_bin = 'symcc_'+f_bin[0]
f_afl_fuzz =
    f_afl.replace('/'+f_afl_basename,'')
#----checking the paths
if not(os.path.exists(f_source)):
    print("Source path does not exist")
if not(os.path.exists(f_afl)):
    print("afl_clang path does not exist")
if not(os.path.exists(f_symcc)):
    print("symcc path does not exist")
#----exporting path
cmd1 = "export PATH=$PATH:" + "f_afl_fuzz"
os.system(cmd1)
print("Symcc waits for " + sleep_time +
    "s while afl runs in the background")
cmd2 = "sleep " + str(sleep_time)
os.system(cmd2)
print("Running Symcc")
#----runnign symcc instance
cmd3 = "~/.cargo/bin/symcc_fuzzing_helper
    -o afl_out10 -a afl-secondary -n
    symcc -- " + f_symcc_bin + " @@"
print(cmd3)
os.system(cmd3)
```

D. Code for GDB crash Analysis

```
#!/usr/bin/python3.10

import os
import sys
import subprocess
import sys

source_path = sys.argv[1]
data_path = sys.argv[2]

cmd1 = "gdb --args "+ "./" +
    source_path + " " + data_path
print(cmd1)
os.system(cmd1)
```

E. Code for Triage automation

```
#!/usr/bin/python3.10
import os
import sys
import subprocess
import sys
source_path = sys.argv[1]
# print(cmd1)
os.system("rm crashwalk.db")
os.system("rm *.txt")
cmd1 = "cwtriage -afl -seen -root" + source_path
os.system(cmd1)
print('Executed cwtriage')
os.system("cwdump crashwalk.db > triage.txt")
print("Dumped the analysis in triage.txt")
```