Incorrect Device to IPA mapping, AFL

Anirban, ac4743@nau.edu

I decided to find and select a known critical vulnerability in the largest (several million lines of code) open-source project ever - the Linux operating system, or rather the Linux Kernel source code (available to freely browse on platforms like GitHub).

The vulnerability introduced incorrect mapping from a host's device memory to a protocol processor (used for hardware handling of network operations), and it was discovered late, much after the code already caused problems in that version of Linux and the enormous number of projects that used it. This critical bug was caused due to bit shifting of an unsigned 32-bit integer, followed by a subsequent expansion to a 64-bit equivalent which makes the shifted integer lose some significant bits and therefore have incorrect values.

This affected device memory translation for virtual machines running Linux with more than 4GB of address space (which is almost all VMs nowadays), and had a severe implication - incorrect memory values being mapped.

This vulnerability relates to the CIA classification triad in these ways:

1) Confidentiality is not breached, as this bug does not allow attackers unauthorized access to steal data. This tends to be more of an issue in terms of data corruption, with improper addressing of needs with more memory.

2) Integrity might have been impacted in terms of trustworthiness for use of Linux Kernel-based Virtual Machines (KVMs).

3) Availability is preserved in this case for stored content until 4GB of memory, as the information is in fact consistently and readily accessible for authorized parties. However, if one is placing things above 4GB of main memory, it leads to corruption and thus instability of the memory prior to it with the paging, which essentially has misinformation.

For reference, here is the code snippet from the Linux Kernel's open-sourced source code (in arch/arm/kvm/mmu.c) leading to the vulnerability: (i.e., the code prior to the fix, source: this GitHub commit)

```
if (vma->vm_flags & VM_PFNMAP) {

    gpa_t gpa = mem->guest_phys_addr + (vm_start -
mem->userspace_addr);

    phys_addr_t pa = (vma->vm_pgoff << PAGE_SHIFT) + vm_start
- vma->vm_start;

    ...

}
```

And this was the fix:

```
if (vma->vm_flags & VM_PFNMAP) {

    gpa_t gpa = mem->guest_phys_addr + (vm_start -
mem->userspace_addr);

    phys_addr_t pa = (phys_addr_t)vma->vm_pgoff << PAGE_SHIFT;

    pa += vm_start - vma->vm_start;

    ...

}
```

Basically, they needed to first cast the unsigned 32-bit integer to a 64-bit one early on and only after this cast should they shift the variable to prevent loss of higher-value bits. (the commit message is helpful and succinctly summarizes the fix!)

And thus, the fix was to change the order of operations - first cast the unsigned 32-bit integer to a 64-bit one (notice the explicit `(phys_addr_t)`) and then, only after this cast shift the variable (`... << PAGE_SHIFT`) to prevent loss of higher-value bits.

In this report, I'll be creating a small reproducible example of the vulnerability and elaborate on that (primarily since running afl-fuzz on the entire Linux source code for that version and even compiling the whole darn thing through afl-gcc would take me a lot of time!)

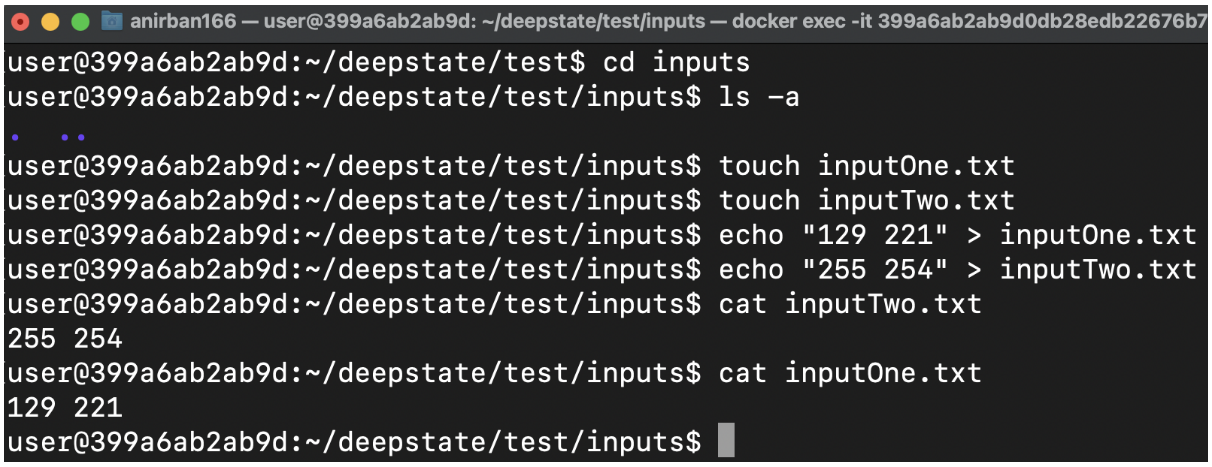
Here is the code that I used (since the Linux kernel is written in C, I did everything in that language itself), which includes a function named `func` that replicates or at least attempts to introduce similar behaviour that the vulnerability established:

```
#include <stdio.h>
#include <assert.h>
#include <inttypes.h>

uint16_t func(uint8_t x, uint8_t y)
{
  uint16_t data = (x << 3) + y;
  return data;
}

int main()
{
  uint8_t x, y;
  assert(scanf("%hhd %hhd", &x, &y) > 0);
  uint16_t z = func(x, y);
  printf("%hu", z);
}
```

The aforementioned function takes as input two 8-bit unsigned integers, performs a left shift on the first one by 3 bits, adds it with the second one, and then assigns the value to a 16-bit unsigned integer. A number in a certain range (0 to 255 here for `uint8_t`) shifted and added with another one of the same range has the potential to lead to the resultant value (that is stored and returned from the function) having loss of higher precision bits, when assigned to a data type with a higher range.

For the rest of the code, I just pass in two input 8-bit unsigned integers (using `assert` to properly deal with the input from `scanf`) from my main to that function, and print out the resultant integer (doesn't make any difference to print something for the purposes of fuzzing, but I still considered to keep it like a normal program which only relies on debugging through printing).

Now that my program is done, I proceed to create some inputs to feed to the fuzzer (AFL) as corpus. I first create a directory named ‘inputs’ and then inside it, I create two text files, both of which contain *valid* 8-bit unsigned integers in the format that my program takes as input (i.e., both separated by a space as I accept through my `scanf` function):



```
anirban166 — user@399a6ab2ab9d: ~/deepstate/test/inputs — docker exec -it 399a6ab2ab9d0db28edb22676b7
user@399a6ab2ab9d:~/deepstate/test$ cd inputs
user@399a6ab2ab9d:~/deepstate/test/inputs$ ls -a
.  ..
user@399a6ab2ab9d:~/deepstate/test/inputs$ touch inputOne.txt
user@399a6ab2ab9d:~/deepstate/test/inputs$ touch inputTwo.txt
user@399a6ab2ab9d:~/deepstate/test/inputs$ echo "129 221" > inputOne.txt
user@399a6ab2ab9d:~/deepstate/test/inputs$ echo "255 254" > inputTwo.txt
user@399a6ab2ab9d:~/deepstate/test/inputs$ cat inputTwo.txt
255 254
user@399a6ab2ab9d:~/deepstate/test/inputs$ cat inputOne.txt
129 221
user@399a6ab2ab9d:~/deepstate/test/inputs$
```

Note that as you can see above, I'm running everything in a Docker container (with an image running the Linux operating system), as I feel more open to experiment there and perform potentially risky actions like fuzzing (I say risky since running the fuzzing process for a very long amount of time could result anywhere from millions to trillions of writes, apart from costly forks calls, file I/O, and extraneous CPU usage), with no compromise when I mess up things.

Anyways, the next step is to compile my program. But it isn’t as simple as just using the default compiler for my OS/environment (which is gcc here), as AFL requires programs to be compiled through afl-specific versions that include their special instrumentation to help the fuzzer. One can either export these settings as environment variables or just use the appropriate version while running manually or including in build system files (like makefiles for make).

The compilers vary depending on the platform being used (instrumentation will be different for good reasons). For instance, for C on Linux its afl-gcc, while on OS X its afl-clang. If the language I wrote the code in was C++, then I’d have to use afl-g++. (and if I was using AFL++ instead, then I’d need to use afl-gcc-fast for C, or afl-g++-fast for C++ respectively)

Since my container used Linux and since my code is written in C, I used afl-gcc to compile.

Here is a list of options I played around with and a bit of information on them based on my experience fuzzing with AFL:

1) Directories: Specifying input and output directories is a **must** for every AFL run. One can specify the path to the directory (relative to the directory one is currently in) using the flags `-i` and `-o` respectively.

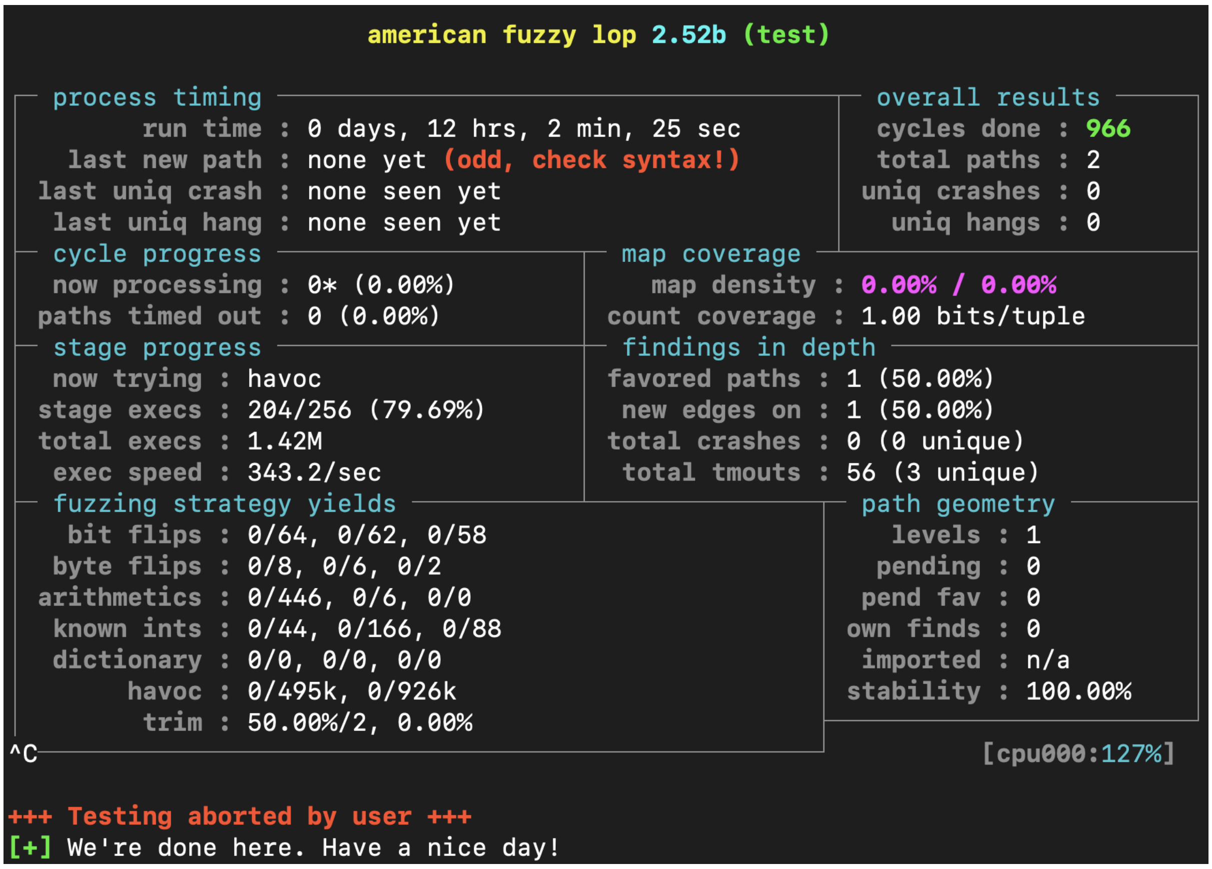
2) Time limit: By default, there is a timeout limit of 40ms that is set for each program execution, but one can specify it to be of a higher value using the `-t` flag. Since my program is very small and wouldn't translate to logic that consumes a lot of time, I did not need to extend this limit. Note that this must not be confused with the fuzzing time, as that is still indefinite or infinite by definition (since all afl-stages run in a loop), and the fuzzer would only stop when I or the user gives it a signal to stop from the terminal it got started.

Note that AFL can be run in parallel fashion as well, i.e. in the master-slave model that it employs where one thread or process (if forks are enabled) diverges into more and every process either reports back to the master, or they get synced after performing the parallel fuzzing work till a stage is completed. (core difference here is that the master process makes deterministic choices, whereas the slave process often makes random choices (or more of the actual 'fuzzing'). The time limit then applies to all processes individually. (this model makes for efficient long runs, as in short runs the parallel process duplicate and increase the queue of inputs from one seed to another, making it large enough to lead to possible starvation in some processes)

3) Memory limit: This is very critical to specify when running the fuzzer in a constrained environment and with the default limit. In fact, the very first issue that I ran into while going with this setup was that I was running out of memory, since the default memory limit specification was too low (again, its specified by the environment and its constraints usually, apart from AFL's default of 50 megabytes) and wouldn't make for much of a progress in finding new paths (afl-showmap could be used to check this) through mutation.

For this, I used the `-m` option while compiling to set the memory limit to none, which literally allows no constraints on the resource usage (running `ulimit` would return unlimited), and I do this specifically to avoid an OOM fault in the dynamic linker that AFL at times could report due to the default cap being restrictive. Again, I'm running this on a container so the concerns are less.

4) Filename(s): Along with the executable name (after specification of the must-have input and output directories), one can specify the name of the file that fuzzers should target or read specifically (if there is one or a select few even) by setting the `-f` flag, which is usually combined with `@@` when not relying on stdin for input. I ran into the problem of not being able to detect new coverage paths with the 'odd, check syntax!' issue during my first run of over half a day, which indicated that something was wrong with my provided corpus or test cases:



```
                    american fuzzy lop 2.52b (test)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 12 hrs, 2 min, 25 sec      │  cycles done : 966    │
│   last new path : none yet (odd, check syntax!)      │  total paths : 2      │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0* (0.00%)        │    map density : 0.00% / 0.00%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 1 (50.00%)             │
│ stage execs : 204/256 (79.69%)      │  new edges on : 1 (50.00%)             │
│ total execs : 1.42M                 │ total crashes : 0 (0 unique)           │
│  exec speed : 343.2/sec             │  total tmouts : 56 (3 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/64, 0/62, 0/58                      │    levels : 1          │
│  byte flips : 0/8, 0/6, 0/2                         │   pending : 0          │
│ arithmetics : 0/446, 0/6, 0/0                       │  pend fav : 0          │
│  known ints : 0/44, 0/166, 0/88                     │ own finds : 0          │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 0/495k, 0/926k                        │ stability : 100.00%    │
│        trim : 50.00%/2, 0.00%                       ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000:127%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

It was at this point when I changed my program to read a file stream instead, and got to know of this explicit file specification option.

But soon after this, I found out that my test cases were not formatted properly, and that I also had a few warnings (width of bit shifts exceeding the range, for instance) and roadblocks in my initial program that led to preemptive runs where the execution stops at those points and does not get to the point of proper program completion. After fixing these, I was able to run my fuzzing process properly and find crashes.

5) Sanitizers: AFL being coverage-guided, has sanitizer-coverage instrumentation on the target binary and can run with sanitizers just by setting the usual `-fsanitizer=*` flags that most fuzzers use. For address sanitizer (or for help in detecting memory corruption bugs such as buffer overflows or accesses to a dangling pointer), `-fsanitize=address` should be used as a flag. For undefined behaviour (the result of any operation with undefined semantics; common examples being division by zero, dereferencing a null pointer, loading memory from a misaligned pointer, etc.), the corresponding flag is `-fsanitize=undefined`.

Sanitizers can also be set using predefined binary environment variables. For instance, instead of the above flags, one can use `AFL_USE_ASAN=1` and `AFL_USE_UBSAN=1`. (Note that there are also other types of sanitizers such as control flow sanitizer, leak sanitizer, thread sanitizer, etc. that AFL supports, and all have such flags with the syntax `AFL_USE_*`)

For my program, I did not use any sanitizers since I know I do not have any memory leaks or undefined behaviour for the simple code that I wrote, and it would just take more time for the fuzzer to run each execution with these flags set.

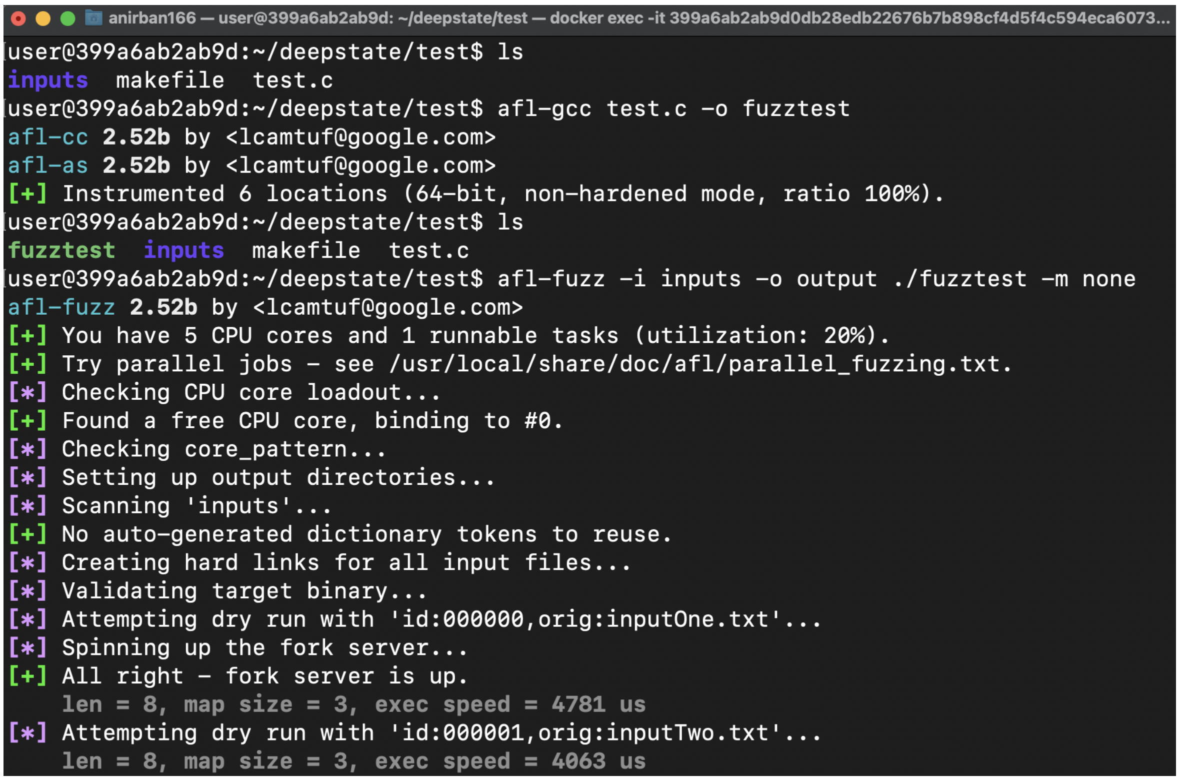
That's pretty much all I have experimented with and come to know about for the core flags that are used. (more can be found in the manual)

Finally, here's the command I used for compiling, linking the target binary, and then executing the fuzzer on it:

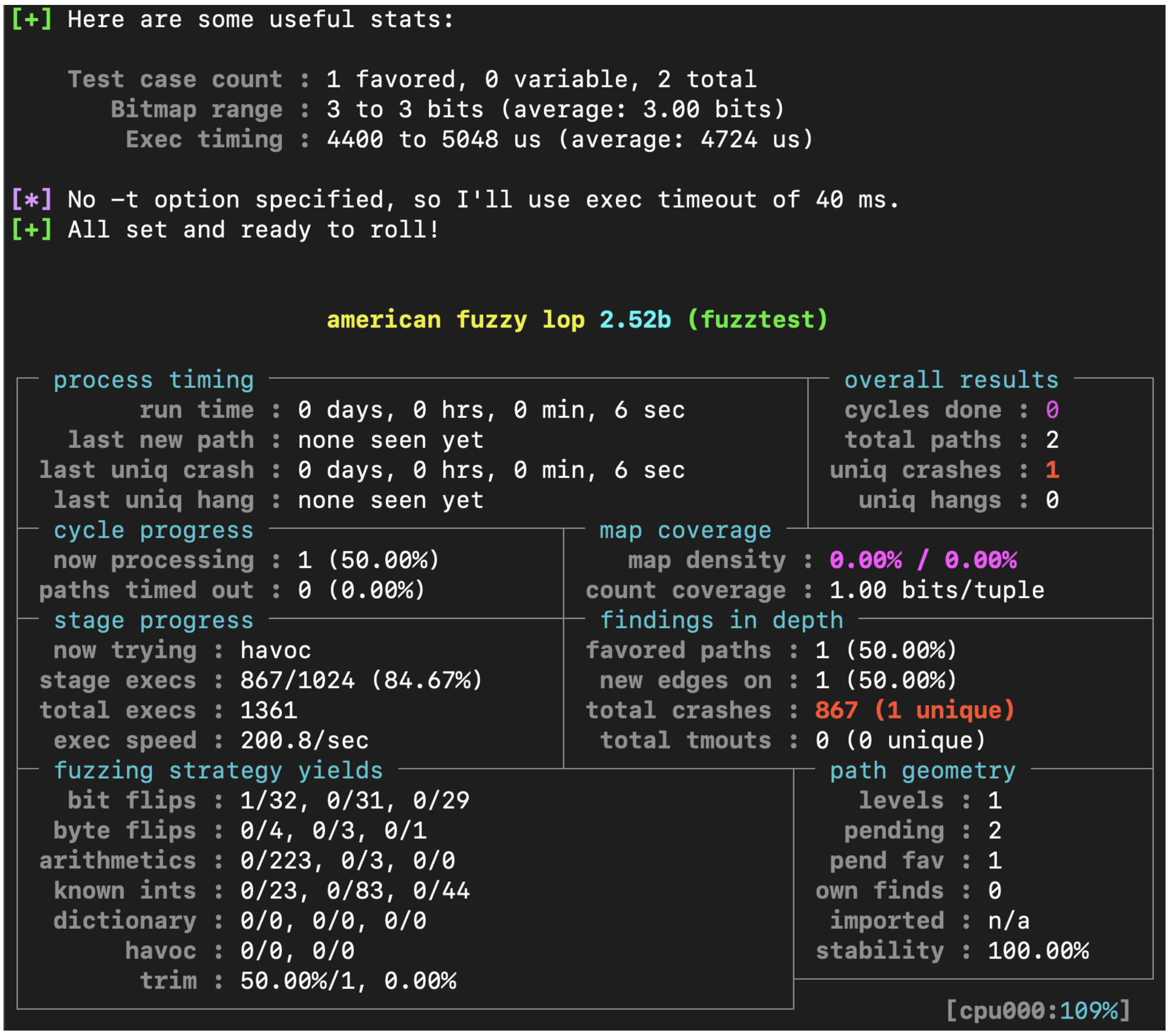
```
afl-gcc test.c -o fuzztest && afl-fuzz -i inputs -o out
./fuzztest -m none
```

For reference, I've included screenshots below of my directory setup, and of running the aforementioned command after I finished writing and modifying my program (`test.c`) and inputs. (I also used make before so there's a makefile, but since I'm dealing with only one file and the compilation command is small, I just decided not to use it)

One can also see the fuzzer execute and run properly/successfully:



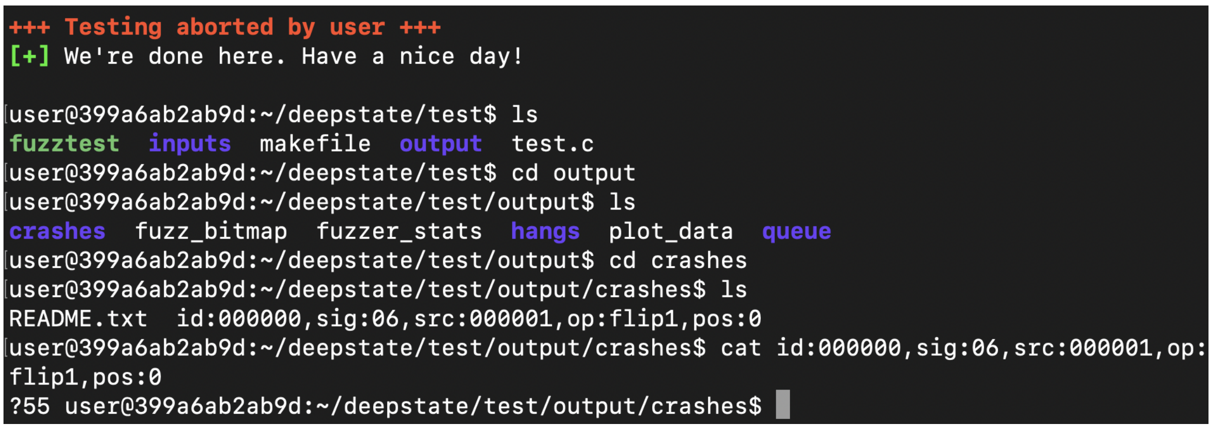
```
anirban166 — user@399a6ab2ab9d: ~/deepstate/test — docker exec -it 399a6ab2ab9d0db28edb22676b7b898cf4d5f4c594eca6073...
user@399a6ab2ab9d:~/deepstate/test$ ls
inputs  makefile  test.c
user@399a6ab2ab9d:~/deepstate/test$ afl-gcc test.c -o fuzztest
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 6 locations (64-bit, non-hardened mode, ratio 100%).
user@399a6ab2ab9d:~/deepstate/test$ ls
fuzztest  inputs  makefile  test.c
user@399a6ab2ab9d:~/deepstate/test$ afl-fuzz -i inputs -o output ./fuzztest -m none
afl-fuzz 2.52b by <lcamtuf@google.com>
[+] You have 5 CPU cores and 1 runnable tasks (utilization: 20%).
[+] Try parallel jobs - see /usr/local/share/doc/afl/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Setting up output directories...
[*] Scanning 'inputs'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:inputOne.txt'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 8, map size = 3, exec speed = 4781 us
[*] Attempting dry run with 'id:000001,orig:inputTwo.txt'...
    len = 8, map size = 3, exec speed = 4063 us
```



```
[+] Here are some useful stats:

    Test case count : 1 favored, 0 variable, 2 total
       Bitmap range : 3 to 3 bits (average: 3.00 bits)
        Exec timing : 4400 to 5048 us (average: 4724 us)

[*] No -t option specified, so I'll use exec timeout of 40 ms.
[+] All set and ready to roll!


                       american fuzzy lop 2.52b (fuzztest)

┌─ process timing ─────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 0 hrs, 0 min, 6 sec        │  cycles done : 0  │
│   last new path : none seen yet                      │  total paths : 2  │
│ 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 : 1 (50.00%)        │    map density : 0.00% / 0.00%     │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple   │
├─ stage progress ────────────────────┼─ findings in depth ────────────────┤
│  now trying : havoc                 │ favored paths : 1 (50.00%)         │
│ stage execs : 867/1024 (84.67%)     │  new edges on : 1 (50.00%)         │
│ total execs : 1361                  │ total crashes : 867 (1 unique)     │
│  exec speed : 200.8/sec             │  total tmouts : 0 (0 unique)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────┤
│   bit flips : 1/32, 0/31, 0/29                      │    levels : 1      │
│  byte flips : 0/4, 0/3, 0/1                         │   pending : 2      │
│ arithmetics : 0/223, 0/3, 0/0                       │  pend fav : 1      │
│  known ints : 0/23, 0/83, 0/44                      │ own finds : 0      │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a    │
│       havoc : 0/0, 0/0                              │ stability : 100.00%│
│        trim : 50.00%/1, 0.00%                       ├────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000:109%]
```

For the run above, (a screenshot of which I took early on) I already found a unique crash, with 867 crashes of the same nature in just 6 seconds. That's how blazingly fast AFL is!

I stop the fuzzer soon since I already found one unique crash which should be sufficient to demonstrate its working. Now, if we were to triage or identify the crashing element and get insightful results from AFL, we are almost always concerned about the 'crashes' directory that gets created in the output directory that we specify (along with other files that might be of interest to the user). Within it are files that each contain inputs that led to or signify a unique crash that has been detected by the fuzzer, given with as many details as can be provided in the name of the file itself:



```
+++ Testing aborted by user +++
[+] We're done here. Have a nice day!

user@399a6ab2ab9d:~/deepstate/test$ ls
fuzztest  inputs  makefile  output  test.c
user@399a6ab2ab9d:~/deepstate/test$ cd output
user@399a6ab2ab9d:~/deepstate/test/output$ ls
crashes  fuzz_bitmap  fuzzer_stats  hangs  plot_data  queue
user@399a6ab2ab9d:~/deepstate/test/output$ cd crashes
user@399a6ab2ab9d:~/deepstate/test/output/crashes$ ls
README.txt  id:000000,sig:06,src:000001,op:flip1,pos:0
user@399a6ab2ab9d:~/deepstate/test/output/crashes$ cat id:000000,sig:06,src:000001,op:
flip1,pos:0
?55 user@399a6ab2ab9d:~/deepstate/test/output/crashes$ 
```

As you can see, the first crash in this run that was found to be unique was given by the input ?55 with a following white space, as detected by the bit flip stage (as indicated by op:flip1 which means the operation bitflip, by one bit) in AFL (involves flipping of n-consecutive bits by XOR'ing them with others).

Now this is a bug I genuinely didn't expect from my program, as I was only expecting the misinformation from the bit shift on valid inputs, while I was unaware invalid inputs like the one above existed which could make my program crash. In fact, now that I think of it, the vulnerability I was looking for wouldn't lead to a crash too unless I deal with `z` in my program in a mischievous manner or have a strict failing assert to compare the value with and without the bit shift, so I wouldn't have been able to find that particular issue in terms of a 'crash' with AFL easily.

Notes from my libFuzzer usage: (extra!)

I was tempted to use libFuzzer through DeepState since that would make things easier, but I tried to do it standalone just like I did with AFL, without its integration in a python environment that can be called easily via a command line executor that DeepState has. (it's just one fuzzer as part of the backend that DeepState sets up)

Unlike AFL, libFuzzer doesn't need to have a compiler with its instrumentation, and can just use gcc for Linux or clang for OS X (for C i.e., and g++/clang++ for C++).

Only requirement is to link the fuzz target with the flag `-fsanitize=fuzzer`, and to preferably also use a sanitizer such as AddressSanitizer with the flag `-fsanitize=address`.

This is the compiling and linking statement I used to build the binary:
```
gcc -g -fsanitize=address,fuzzer test.c -o fuzztest
```
And this is the command I used to execute the fuzzer on it: (I specify the number of individual runs, not letting the fuzzer run infinitely - the value for that is -1 or the default indicating an infinite loop)
```
./fuzztest -max_len=512 -runs=4269
```

`-g` enables debugging information. `-max_len` lets me set up the maximum size (in bytes) for my inputs. I don't specify a seed so it’s randomly generated, but one can specify using `-seed=N`.

To filter the error files, I used `grep ERROR *.log` (for crashes, it’s `*crash`) and then piped in commands to filter further, as per need.

To have parallel runs with libFuzzer or to increase its fuzzing efficiency by using more CPU processes say, one can run the fuzzer with `-jobs=N` (`N=32 && ./executable ... -jobs=$N` for instance) which will spawn `N` independent jobs but no more than half of the number of cores one has, and then one can use `-workers=N` to set the number of allowed parallel jobs. Setting a high job count also allows one to find and then filter shallow bugs in the first few runs so that new interesting paths can be discovered soon after (since information is passed in between parallel runs), just so that the fuzzer isn't stuck on the same logic.