SPQR

SPQR

The Roman Xpl0it

A little green bug has come to get squashed!

Fuzzing like the Legendary Super Saiyan

by malweisse

About me

Andrea Fioraldi [@andreafioraldi, @malweisse on IRC]

22 years old, student of Engineering in Computer Science.

Binary analysis, exploitation, programming languages.

Capturing flags with TRX and mhackeroni.

Not so much skilled trumpeter, mountain bike and trekking lover, Dragon Ball fanboy, homebrewer.

Fuzzing Overview

American Fuzzy Lop

Useful strategies

Fuzzing WTF

From Wikipedia:

> *Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.*

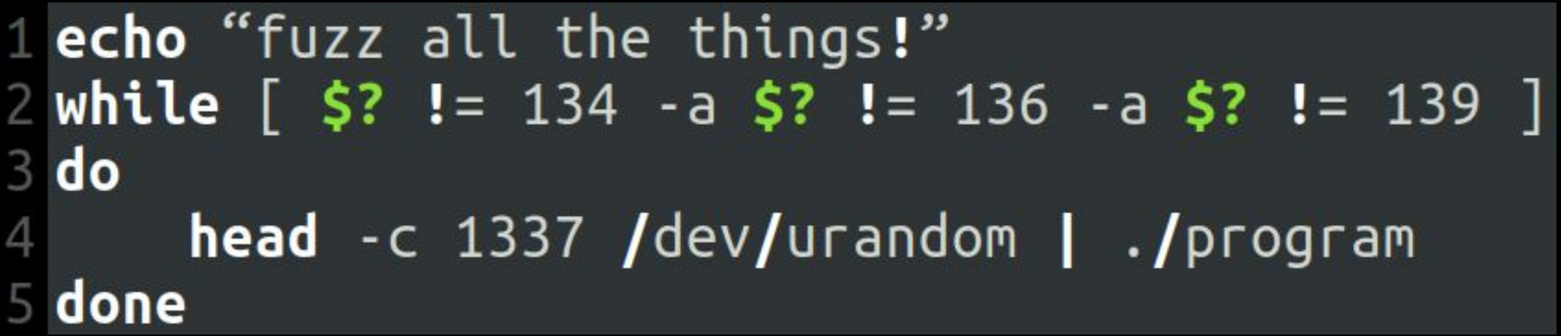
What you can do with a fuzzer:

> Find an input that triggers a bug.

What you can't do with a fuzzer:

> Prove that the fuzzed software has no bugs.

Fuzzing WTF



```
echo “fuzz all the things!”
while [ $? != 134 -a $? != 136 -a $? != 139 ]
do
    head -c 1337 /dev/urandom | ./program
done
```

That’s it.

See you the next month.

THIS IS FUZZING!

Fuzzing WTF

Fuzzing is useful to run A LOT of tests that you can never will generate by hand.

It is very effective with programs that have to process complex inputs.
A program with a parser is a good candidate to be fuzzed.

Some examples are images libraries, PDF readers, compilers, system utils (like readelf or objdump), HTTP servers, etc...

White-Box Fuzzing

A **white-box** fuzzer uses program analysis to maximize the number of explored paths. Tipically the employed analysis is symbolic execution (do you remember my past talk about angr?) like in SAGE [1].

We are interested in maximizing the number of the explored paths in order to increase the probability to reach a critical portion of the code.

This approach is common among all the type of fuzzers, generally we have that

MORE CODE COVERAGE = GOOD

Black-Box Fuzzing

A **black-box** fuzzer is not aware of the target program structure, like the dumb bash fuzzed showed before or the real fuzzer Radamsa [2].

There are smart black-box fuzzers which use heuristics based on the association of an input to the related output.
In this way the fuzzer can learn some information about the program structure.

Grey-Box Fuzzing

A **grey-box** fuzzer makes use of program instrumentation of the source code or the binary (remember the talk about Intel PIN?) to get information.

Typically this information are related to code coverage (**coverage-guided fuzzing**) but there are other possible analysis (e.g. taint tracking) that can be employed using instrumentation in order to get useful information that may help the fuzzer (e.g. CompCov [3], Vuzzer [4], Angora [5]).

In this talk we will focus on one of the most popular grey-box coverage-guided fuzzers, American Fuzzy Lop.

Input structure awareness

Beyond this distinctions, a fuzzer can be aware of the input structure or not. One of the most famous input-aware fuzzer is Peach [6]. With Peach you must describe the input structure using an XML model and then the fuzzer generates the testcases accordingly.

A recent twist of AFL, AFLSmart [7], combines American Fuzzy Lop (we will see how to use it and how it works) with Peach.

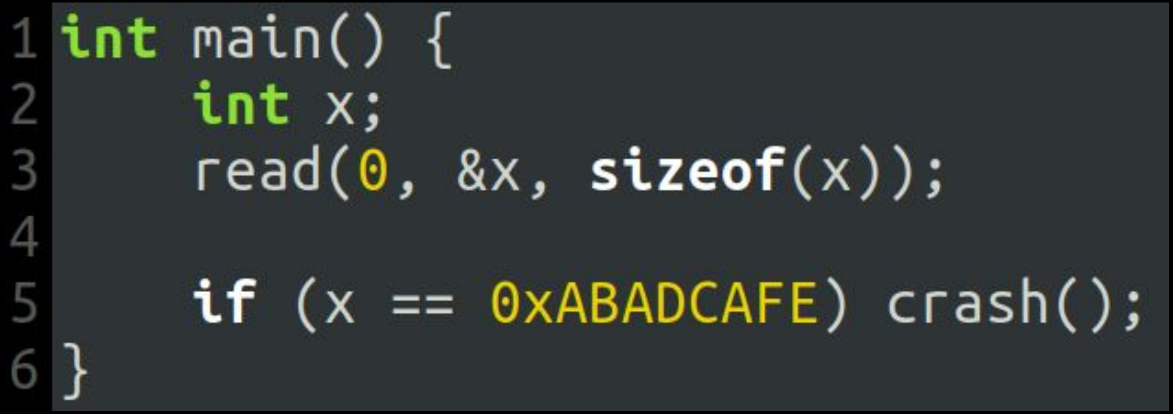
Other notable fuzzers

Other notable grey-box coverage-guided fuzzing are:

- LLVM libFuzzer [8]: the fuzzer provided by the LLVM toolchain.
- Google honggfuzz [9]: a powerful feedback-driven fuzzer (not only coverage but also instructions counting) with an hardware-based engine. It can make use of several hardware facilities exposed by Intel processors instead of instrumentation to get information (e.g. count the number of branches, remember ROPGun?)

Funky hybrid stuffs

The generation of specific inputs to bypass checks can be not trivial for a fuzzer. Consider the following program:



```c
int main() {
    int x;
    read(0, &x, sizeof(x));

    if (x == 0xABADCAFE) crash();
}
```

There is only a path, in this case grey-box == black-box.

Funky hybrid stuffs

The **hybrid fuzzing** approach aims to apply symbolic tracing when the fuzzer is stuck.
The general idea is to collect the constraints related to a path given a concrete input and then, when a branch which can generate a new state transition is encountered, negate those constraints in order to get an input that can bypass the branch.
Concolic execution is back guys.

For more info refers to the Driller [10] paper from UCSB.
Another recent promising tool is QSYM [11] from Georgia Tech.

Fuzzing Overview

➡ **American Fuzzy Lop**

Useful strategies

American Fuzzy Lop

AFL WTF

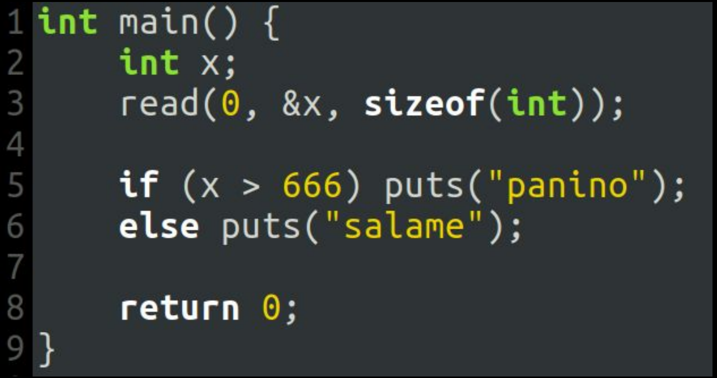
American Fuzzy Lop is one of the most popular fuzzers.
It was used to discover critical vulnerabilities on very widespread software like OpenSSH, PHP, MySQL, Firefox and many others.

It use feedbacks based on the edge coverage.

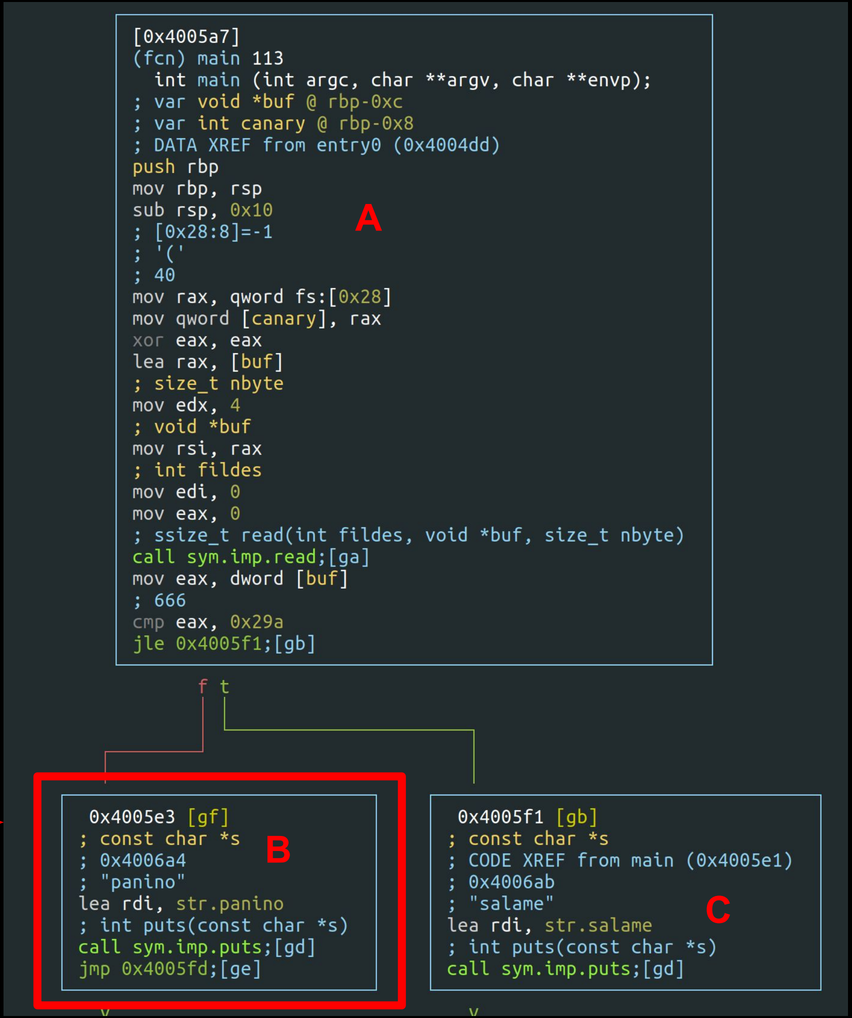
Standard code coverage logs which basic blocks are executed (e.g A B C), edge code coverage logs basic blocks transitions (e.g. A->C A->B).

These transitions are referred as tuples (e.g A->C is (A,C)).

Basic Blocks Coverage



```c
int main() {
    int x;
    read(0, &x, sizeof(int));

    if (x > 666) puts("panino");
    else puts("salame");

    return 0;
}
```



```
[0x4005a7]
(fcn) main 113
  int main (int argc, char **argv, char **envp);
; var void *buf @ rbp-0xc
; var int canary @ rbp-0x8
; DATA XREF from entry0 (0x4004dd)
push rbp
mov rbp, rsp
sub rsp, 0x10
; [0x28:8]=-1
; '('
; 40
mov rax, qword fs:[0x28]
mov qword [canary], rax
xor eax, eax
lea rax, [buf]
; size_t nbyte
mov edx, 4
; void *buf
mov rsi, rax
; int fildes
mov edi, 0
mov eax, 0
; ssize_t read(int fildes, void *buf, size_t nbyte)
call sym.imp.read;[ga]
mov eax, dword [buf]
; 666
cmp eax, 0x29a
jle 0x4005f1;[gb]
```

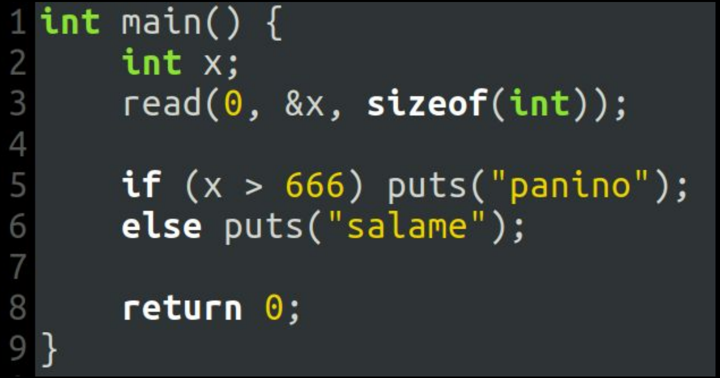
```
 0x4005e3 [gf]
; const char *s
; 0x4006a4
; "panino"
lea rdi, str.panino
; int puts(const char *s)
call sym.imp.puts;[gd]
jmp 0x4005fd;[ge]
```

```
 0x4005f1 [gb]
; const char *s
; CODE XREF from main (0x4005e1)
; 0x4006ab
; "salame"
lea rdi, str.salame
; int puts(const char *s)
call sym.imp.puts;[gd]
```

Basic Block B

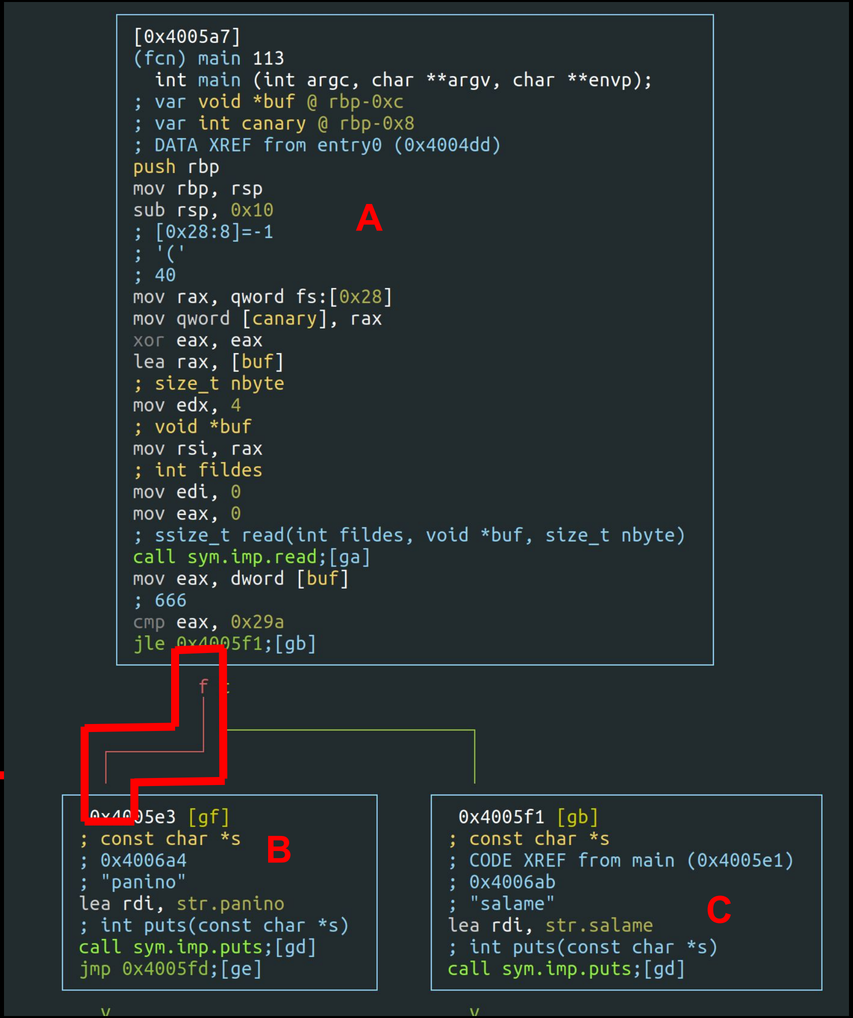
One entry, one exit

Edge Coverage



```c
int main() {
    int x;
    read(0, &x, sizeof(int));

    if (x > 666) puts("panino");
    else puts("salame");

    return 0;
}
```

Edge A -> B



```
[0x4005a7]
(fcn) main 113
  int main (int argc, char **argv, char **envp);
; var void *buf @ rbp-0xc
; var int canary @ rbp-0x8
; DATA XREF from entry0 (0x4004dd)
push rbp
mov rbp, rsp
sub rsp, 0x10
; [0x28:8]=-1
; '('
; 40
mov rax, qword fs:[0x28]
mov qword [canary], rax
xor eax, eax
lea rax, [buf]
; size_t nbyte
mov edx, 4
; void *buf
mov rsi, rax
; int fildes
mov edi, 0
mov eax, 0
; ssize_t read(int fildes, void *buf, size_t nbyte)
call sym.imp.read;[ga]
mov eax, dword [buf]
; 666
cmp eax, 0x29a
jle 0x4005f1;[gb]
```

```
0x4005e3 [gf]
; const char *s
; 0x4006a4
; "panino"
lea rdi, str.panino
; int puts(const char *s)
call sym.imp.puts;[gd]
jmp 0x4005fd;[ge]
```

```
0x4005f1 [gb]
; const char *s
; CODE XREF from main (0x4005e1)
; 0x4006ab
; "salame"
lea rdi, str.salame
; int puts(const char *s)
call sym.imp.puts;[gd]
```

AFL Instrumentation

from the AFL whitepaper [12]:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

shared_mem is a 64kb (fits in L2 chache) SHM shared between the fuzzer and the instrumented target program.
Collisions are possibles.

AFL Instrumentation

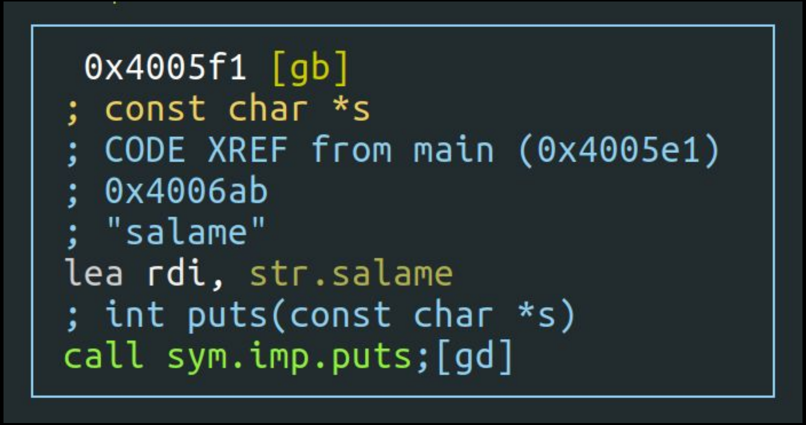
A program can be compiled using **afl-gcc** and **afl-g++** in order to insert instrumentation at compile-time.

There are also wrappers for clang when using LLVM mode, **afl-clang-fast** and **afl-clang-fast++** that build a faster instrumentation code using a custom LLVM pass.

With AFL in LLVM mode you can use also the other clang facilities for data tracking and validation like ASAN, we will see it in the best practices chapter.

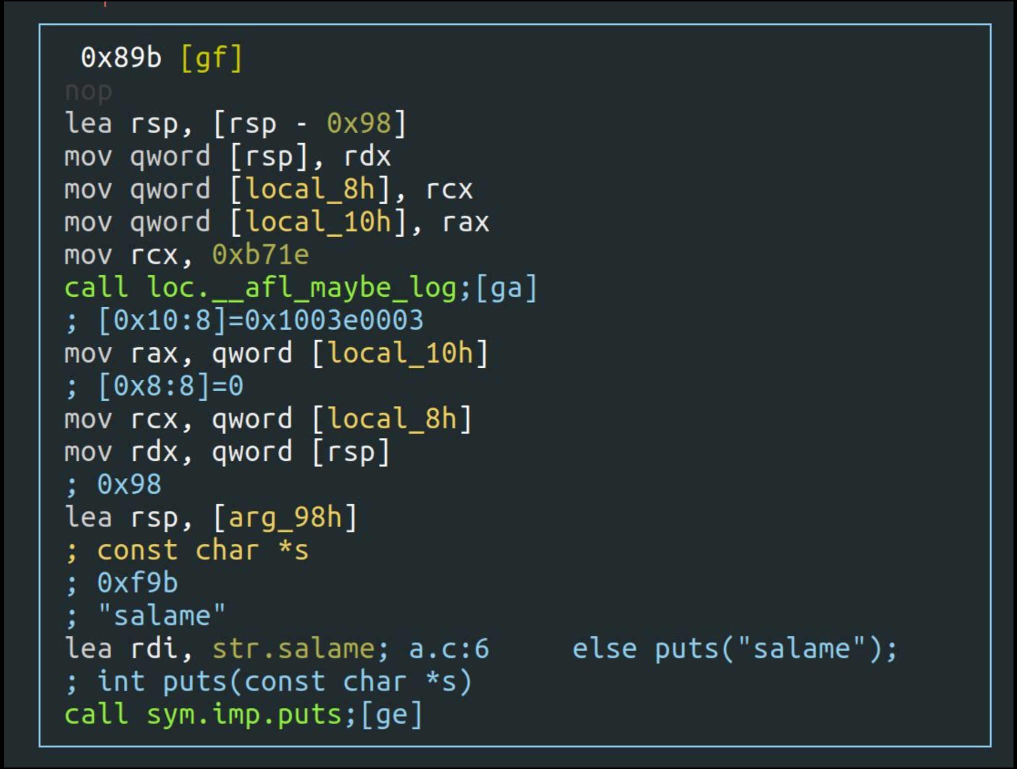
AFL Instrumentation

Standard GCC



```
 0x4005f1 [gb]
; const char *s
; CODE XREF from main (0x4005e1)
; 0x4006ab
; "salame"
lea rdi, str.salame
; int puts(const char *s)
call sym.imp.puts;[gd]
```

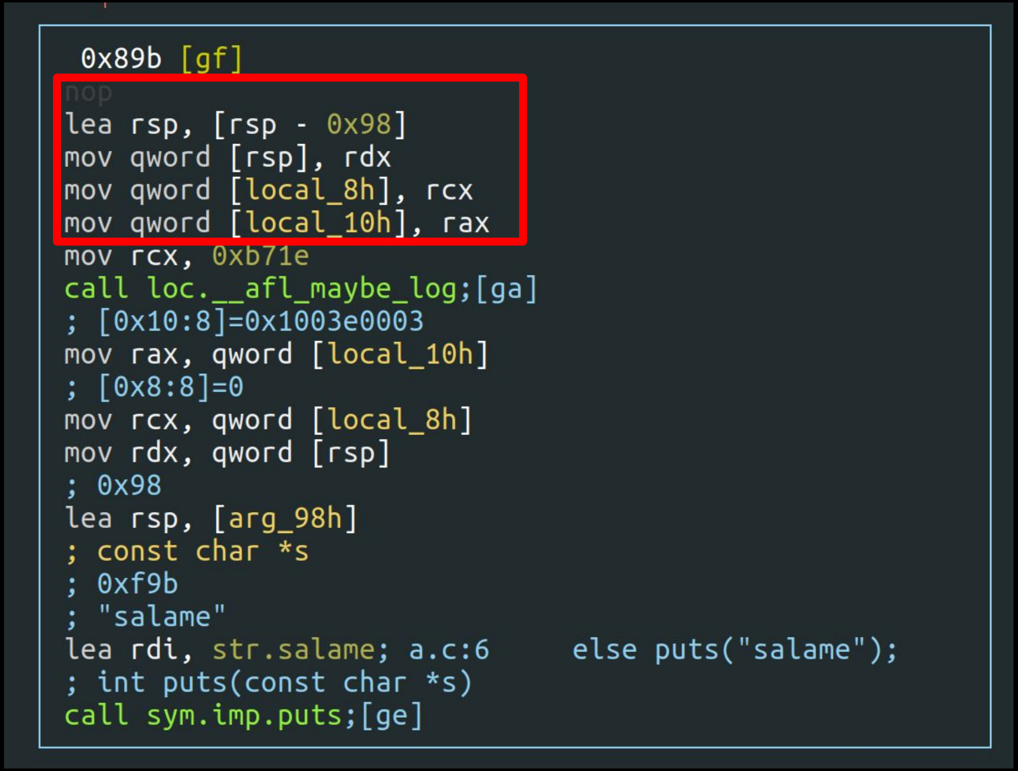
AFL GCC



```
 0x89b [gf]
nop
lea rsp, [rsp - 0x98]
mov qword [rsp], rdx
mov qword [local_8h], rcx
mov qword [local_10h], rax
mov rcx, 0xb71e
call loc.__afl_maybe_log;[ga]
; [0x10:8]=0x1003e0003
mov rax, qword [local_10h]
; [0x8:8]=0
mov rcx, qword [local_8h]
mov rdx, qword [rsp]
; 0x98
lea rsp, [arg_98h]
; const char *s
; 0xf9b
; "salame"
lea rdi, str.salame; a.c:6     else puts("salame");
; int puts(const char *s)
call sym.imp.puts;[ge]
```

AFL Instrumentation

Save registers that instrumentation will clobber

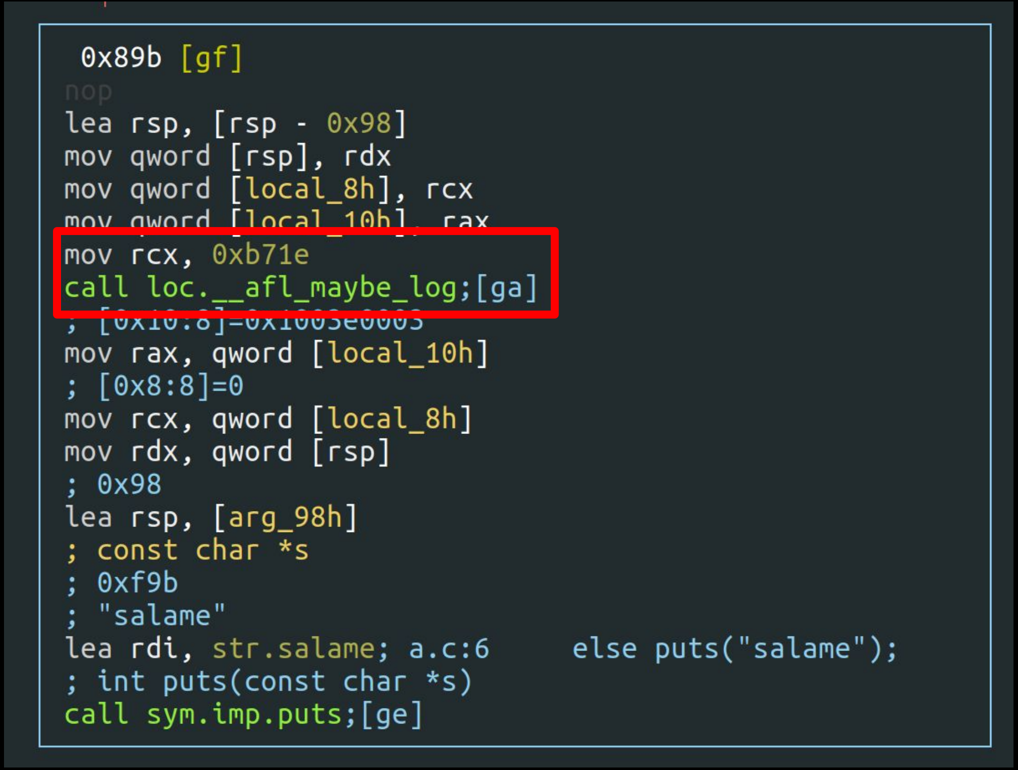


```
 0x89b [gf]
nop
lea rsp, [rsp - 0x98]
mov qword [rsp], rdx
mov qword [local_8h], rcx
mov qword [local_10h], rax
mov rcx, 0xb71e
call loc.__afl_maybe_log;[ga]
; [0x10:8]=0x1003e0003
mov rax, qword [local_10h]
; [0x8:8]=0
mov rcx, qword [local_8h]
mov rdx, qword [rsp]
; 0x98
lea rsp, [arg_98h]
; const char *s
; 0xf9b
; "salame"
lea rdi, str.salame; a.c:6      else puts("salame");
; int puts(const char *s)
call sym.imp.puts;[ge]
```

AFL Instrumentation

rcx = <compile time random>

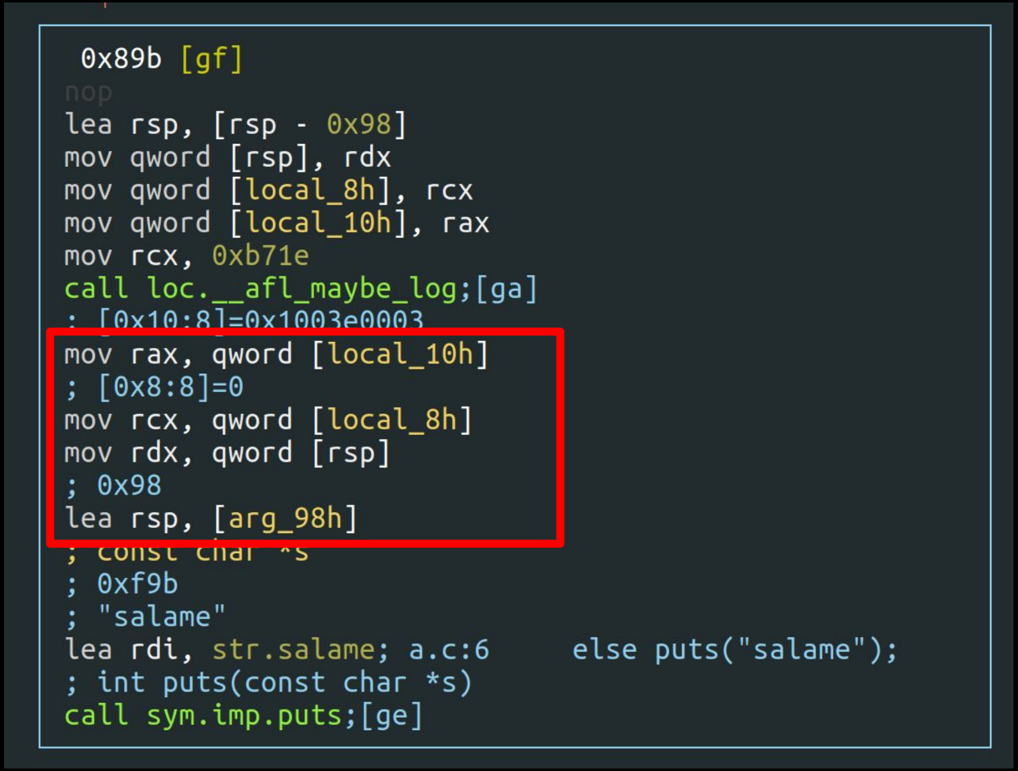
call __afl_maybe_log



```
 0x89b [gf]
nop
lea rsp, [rsp - 0x98]
mov qword [rsp], rdx
mov qword [local_8h], rcx
mov qword [local_10h], rax
mov rcx, 0xb71e
call loc.__afl_maybe_log;[ga]
; [0x10:8]=0x1003e0003
mov rax, qword [local_10h]
; [0x8:8]=0
mov rcx, qword [local_8h]
mov rdx, qword [rsp]
; 0x98
lea rsp, [arg_98h]
; const char *s
; 0xf9b
; "salame"
lea rdi, str.salame; a.c:6      else puts("salame");
; int puts(const char *s)
call sym.imp.puts;[ge]
```

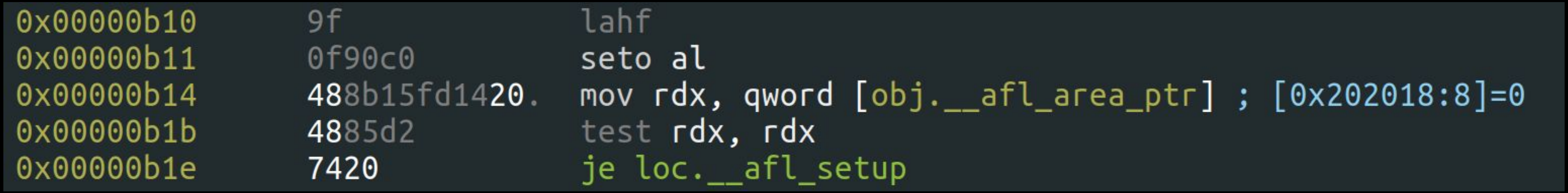
AFL Instrumentation

Restore registers

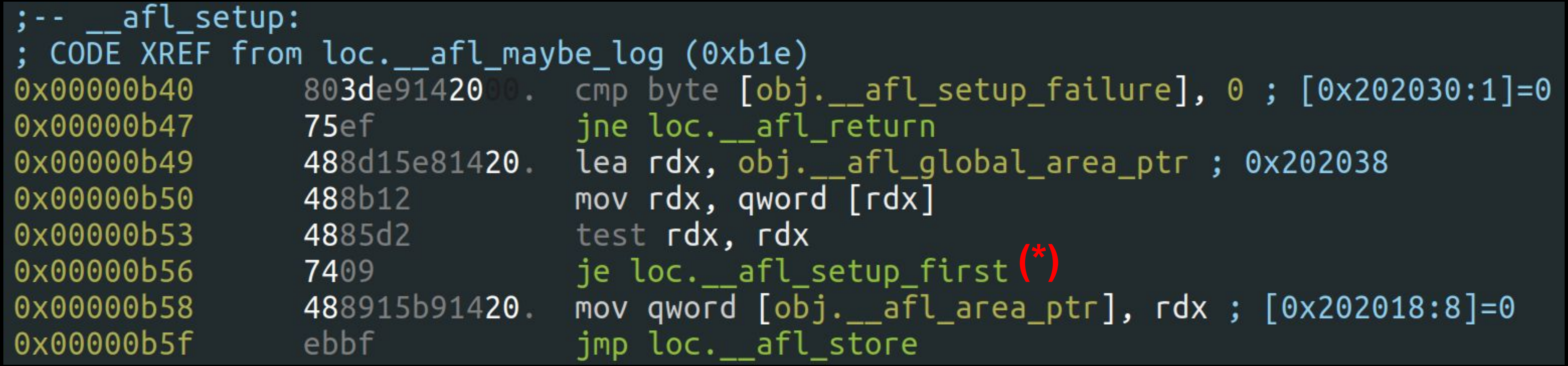


```
 0x89b [gf]
nop
lea rsp, [rsp - 0x98]
mov qword [rsp], rdx
mov qword [local_8h], rcx
mov qword [local_10h], rax
mov rcx, 0xb71e
call loc.__afl_maybe_log;[ga]
; [0x10:8]=0x1003e0003
mov rax, qword [local_10h]
; [0x8:8]=0
mov rcx, qword [local_8h]
mov rdx, qword [rsp]
; 0x98
lea rsp, [arg_98h]
; const char *s
; 0xf9b
; "salame"
lea rdi, str.salame; a.c:6     else puts("salame");
; int puts(const char *s)
call sym.imp.puts;[ge]
```

AFL Instrumentation



```
0x00000b10    9f              lahf
0x00000b11    0f90c0          seto al
0x00000b14    488b15fd1420.   mov rdx, qword [obj.__afl_area_ptr] ; [0x202018:8]=0
0x00000b1b    4885d2          test rdx, rdx
0x00000b1e    7420            je loc.__afl_setup
```



```
;-- __afl_setup:
; CODE XREF from loc.__afl_maybe_log (0xb1e)
0x00000b40    803de9142000.   cmp byte [obj.__afl_setup_failure], 0 ; [0x202030:1]=0
0x00000b47    75ef            jne loc.__afl_return
0x00000b49    488d15e81420.   lea rdx, obj.__afl_global_area_ptr ; 0x202038
0x00000b50    488b12          mov rdx, qword [rdx]
0x00000b53    4885d2          test rdx, rdx
0x00000b56    7409            je loc.__afl_setup_first (*)
0x00000b58    488915b91420.   mov qword [obj.__afl_area_ptr], rdx ; [0x202018:8]=0
0x00000b5f    ebbf            jmp loc.__afl_store
```

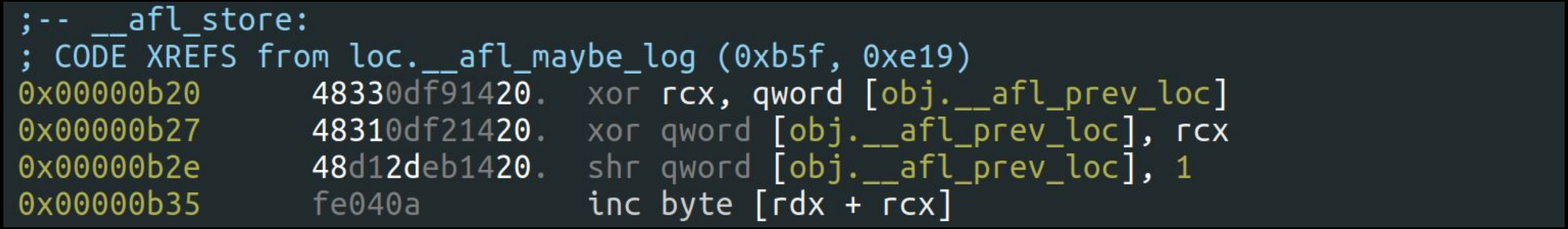
(*) AFL initialization code, we will discuss it later

AFL Instrumentation

Recap:

rcx = <compile time random> (0xb71e in our example)

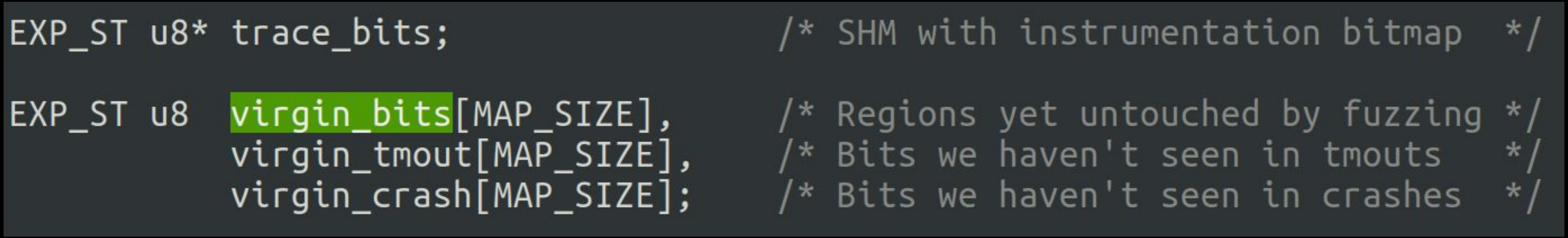
rdx = __afl_area_ptr



```
;-- __afl_store:
; CODE XREFS from loc.__afl_maybe_log (0xb5f, 0xe19)
0x00000b20      48330df91420.  xor rcx, qword [obj.__afl_prev_loc]
0x00000b27      48310df21420.  xor qword [obj.__afl_prev_loc], rcx
0x00000b2e      48d12deb1420.  shr qword [obj.__afl_prev_loc], 1
0x00000b35      fe040a         inc byte [rdx + rcx]
```

AFL Heuristics

The fuzzer maintains a collections of tuples seen in all executions of the target programs.



```
EXP_ST u8* trace_bits;                      /* SHM with instrumentation bitmap  */

EXP_ST u8  virgin_bits[MAP_SIZE],           /* Regions yet untouched by fuzzing */
           virgin_tmout[MAP_SIZE],          /* Bits we haven't seen in tmouts   */
           virgin_crash[MAP_SIZE];          /* Bits we haven't seen in crashes  */
```

Comparing with virgin_bits the fuzzer can know if an input trigger a new local-state (*) and if so it is inserted in a queue for additional processing later and virgin_bits updated.

AFL Heuristics

(*) A local-state is defined as (OR):

- The input register a new tuple in trace_bits
- The input increase the hit_count of a previously seen tuple

hit_count is increases each time a tuple is encountered (shared_mem[...]++).

To avoid path explosion hit_count is divided into buckets:
1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+

So of the previous hit_count is 3 and the new is 4 the input is considered, if the previous was 4 and the new 5 it is not considered.

AFL Queue

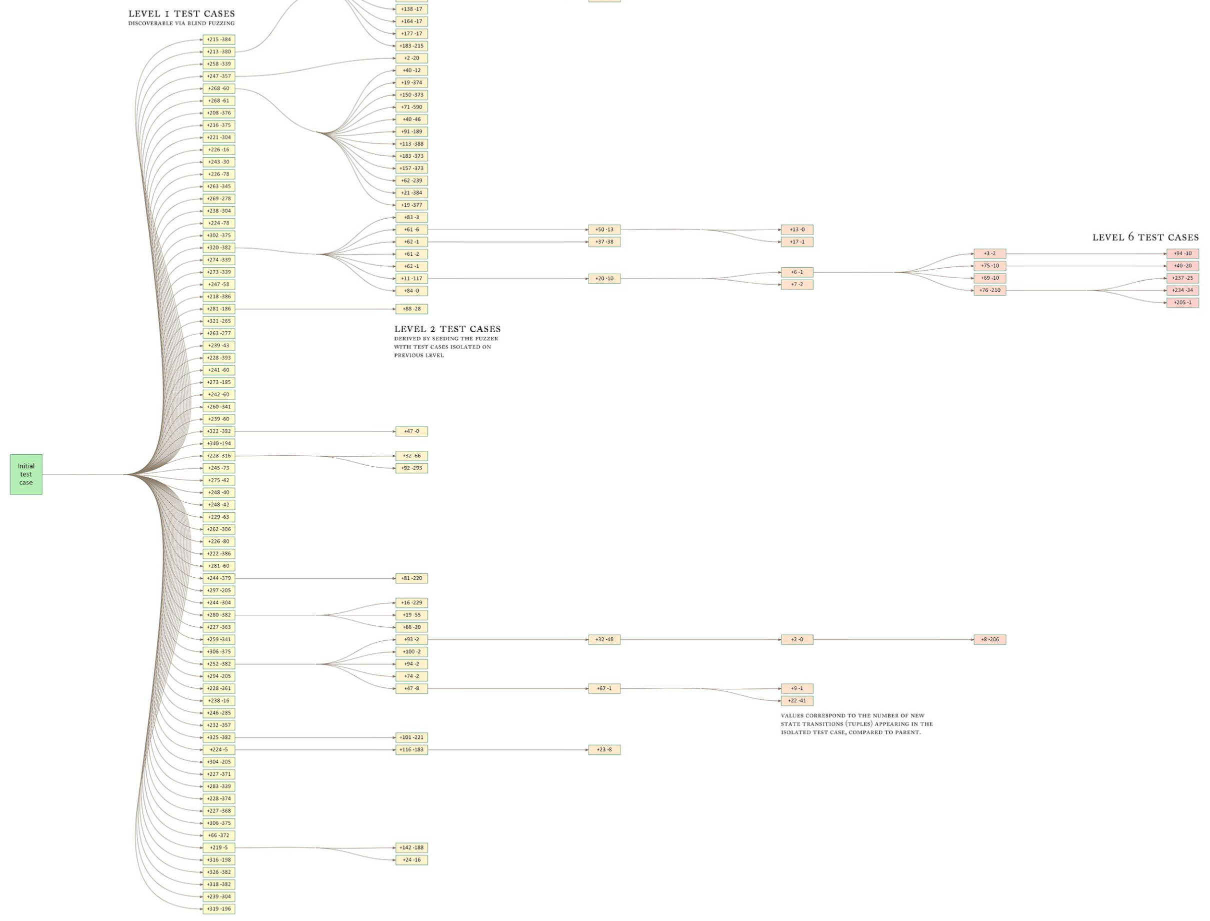
The queue of inputs always grows and new inputs are added as supplement, not replacement.

This allows to explore progressively possible mutually exclusive features of the program.

This means that, on average, the queue reach a size between 1k and 10k elements for most programs.

```
struct queue_entry {

  u8* fname;                          /* File name for the test case      */
  u32 len;                            /* Input length                     */

  u8  cal_failed,                     /* Calibration failed?              */
      trim_done,                      /* Trimmed?                         */
      was_fuzzed,                     /* Had any fuzzing done yet?        */
      passed_det,                     /* Deterministic stages passed?     */
      has_new_cov,                    /* Triggers new coverage?           */
      var_behavior,                   /* Variable behavior?               */
      favored,                        /* Currently favored?               */
      fs_redundant;                   /* Marked as redundant in the fs?   */

  u32 bitmap_size,                    /* Number of bits set in bitmap     */
      exec_cksum;                     /* Checksum of the execution trace  */

  u64 exec_us,                        /* Execution time (us)              */
      handicap,                       /* Number of queue cycles behind    */
      depth;                          /* Path depth                       */

  u8* trace_mini;                     /* Trace bytes, if kept             */
  u32 tc_ref;                         /* Trace bytes ref count            */

  struct queue_entry *next,           /* Next element, if any             */
                     *next_100;       /* 100 elements ahead               */

};
```

AFL-FUZZ, GZIP BINARY
2,000 EXECS/SEC, 1 CORE, 5 HOURS



AFL Queue Evaluation

We said that inputs are only added to queue and not removed, so can happen that the edge coverage of one of the lastest inputs is a superset of the coverage of an old input.

AFL is not dumb. It, periodically, evaluates the testcases in queue.
It assigns a score proportional on execution latency and file size.

Then, for each tuple, it selects the testcase with lowest score.

AFL Queue Evaluation

From the whitepaper:

1. Find next tuple not yet in the temporary working set,
2. Locate the winning queue entry for this tuple,
3. Register *all* tuples present in that entry's trace in the working set,
4. Go to #1 if there are any missing tuples in the set.

All the located winning queue entries are marked as **favored**.

AFL Mutator

The main challenges of a fuzzer on input mutation are the following:

- If the mutation is too poor then the fuzzer can’t reach a good coverage
- If the mutation is too aggressive then probably the fuzzer will generate many testcases that will fail on the early parsing stage

AFL employs a set of deterministic stages on the target input and then, on the later stages, it moves on not-deterministic fuzzing and recombination of different inputs.

AFL Mutator - Deterministic

The deterministic stages are the following [13]:

- Walking bit flips (with auto extras detection)
- Simple arithmetics
- Interesting integers (0, 1, INT_MAX, …)

AFL Mutator - Havoc

The not deterministic stage (havoc) perform this operations at random locations:

- Bit flip (spooky!)
- Insert a interesting integer
- Random endian additions/subtractions
- Single byte set to random value
- Block deletion / duplication / memset

AFL Mutator - Splice

This is the last resort stage, invoked when all the stages before failed to find a new interesting derived input.

It recombines two different inputs from the queue (one is the current that we are mutating) that differs, at least, two locations.

After recombination this fused input is passed in the havoc stage.

Usually this discovers around 20% of new tuples. Think about a double free vulnerability in a HTTP parser, this is the only effective stage to find it.

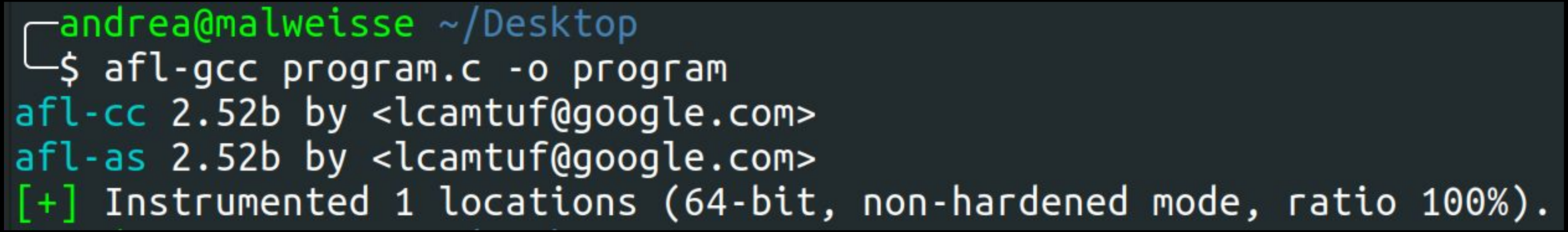
AFL Forkserver

The loader invocation of execve is an useless overhead for a fuzzer.
AFL is a good kid and it’s aware about this [14].

A small piece of code is injected in the ELF so that the child process stops at main. Each requested execution (this is done with a pipe) is a fork of such child process.

With this hack AFL avoids the overhead of loading and dynamic linking (LD_BIND_NOW is employed).

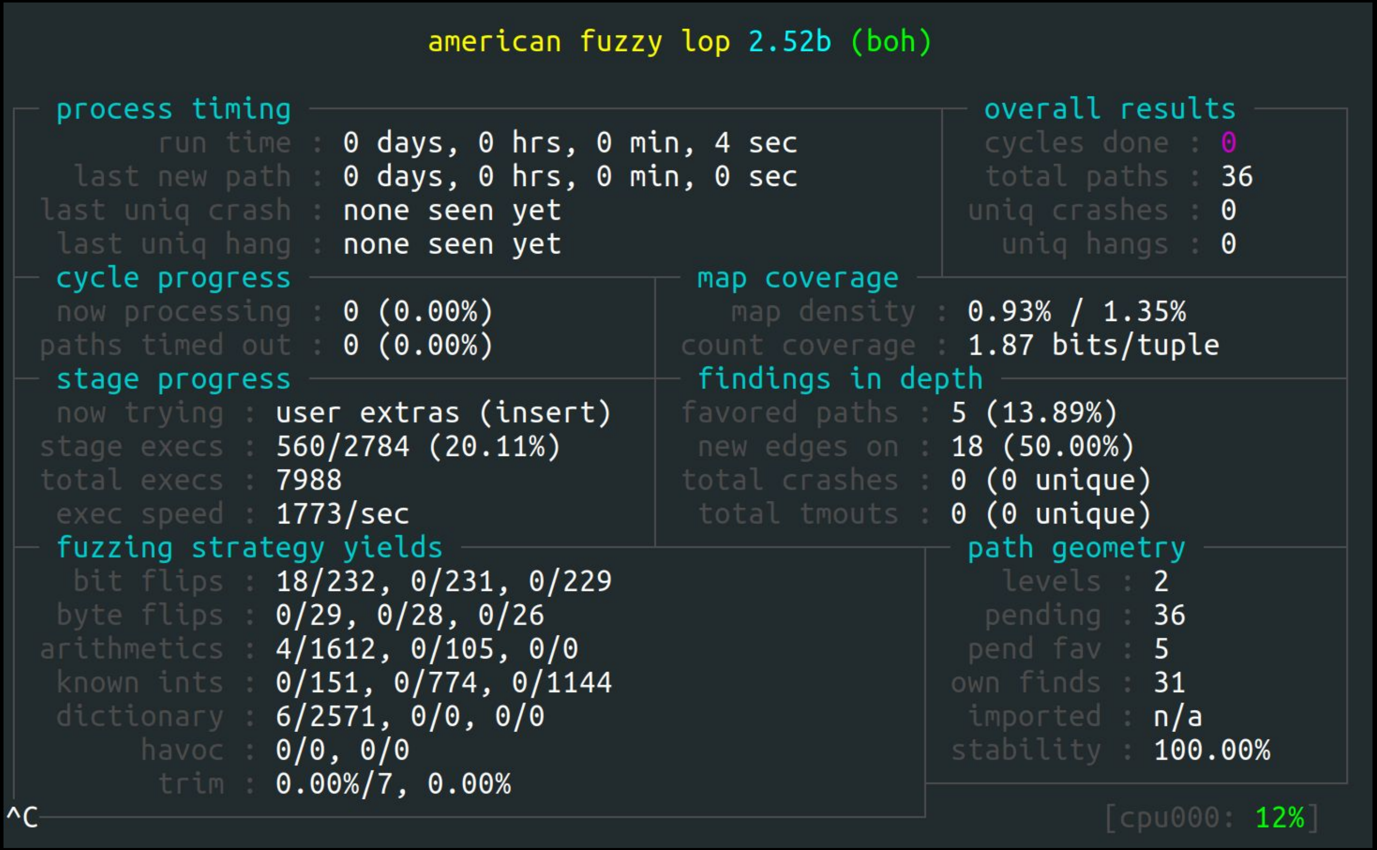
AFL Command line



```
┌─andrea@malweisse ~/Desktop
└─$ afl-gcc program.c -o program
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1 locations (64-bit, non-hardened mode, ratio 100%).
```

```
$ afl-fuzz -i input_directory -o afl_out -- ./program
```

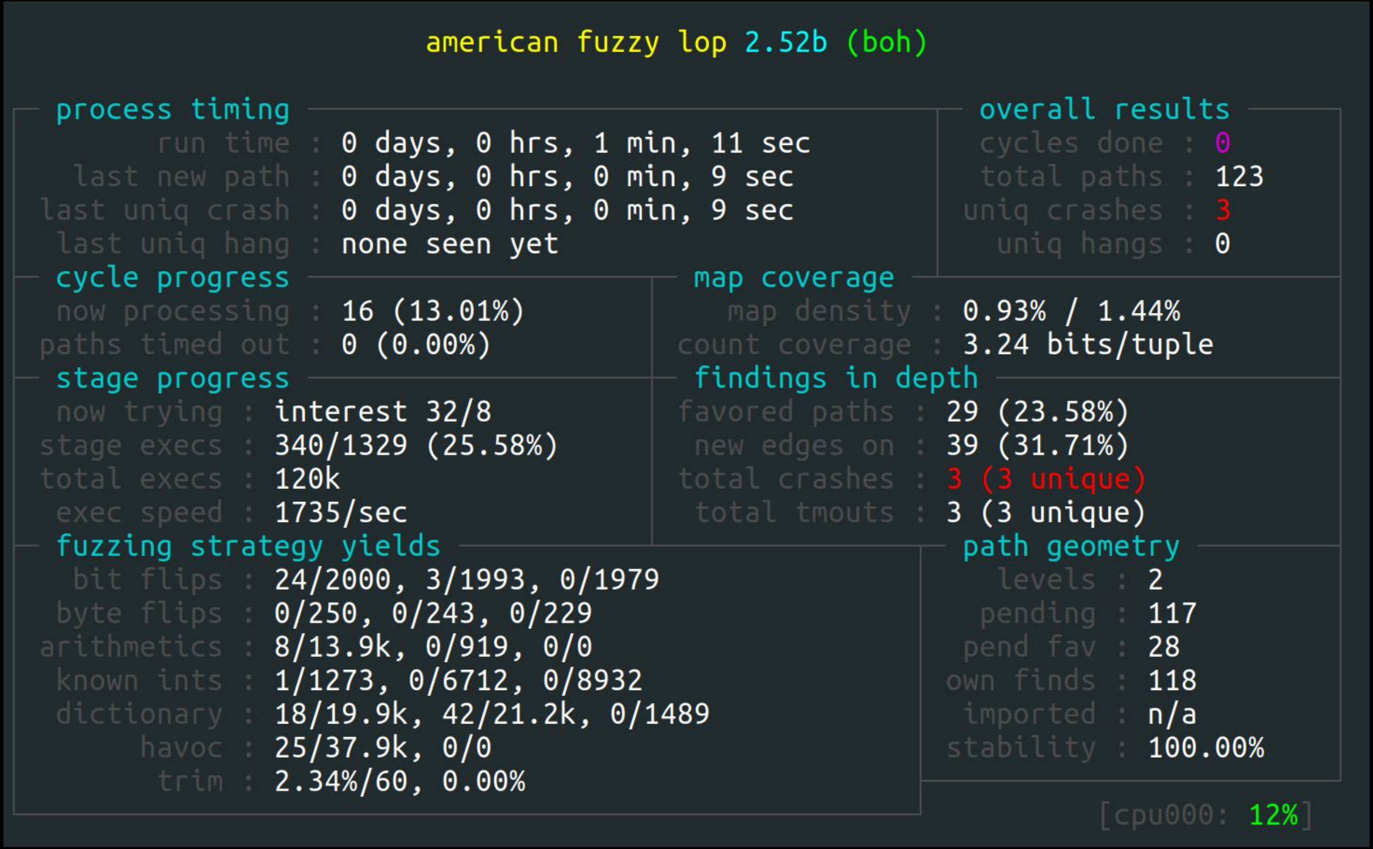
AFL Status Screen



```
                    american fuzzy lop 2.52b (boh)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 4 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec        │  total paths : 36     │
│ 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 : 0.93% / 1.35%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.87 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : user extras (insert)  │ favored paths : 5 (13.89%)             │
│ stage execs : 560/2784 (20.11%)     │  new edges on : 18 (50.00%)            │
│ total execs : 7988                  │ total crashes : 0 (0 unique)           │
│  exec speed : 1773/sec              │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 18/232, 0/231, 0/229                  │    levels : 2          │
│  byte flips : 0/29, 0/28, 0/26                      │   pending : 36         │
│ arithmetics : 4/1612, 0/105, 0/0                    │  pend fav : 5          │
│  known ints : 0/151, 0/774, 0/1144                  │ own finds : 31         │
│  dictionary : 6/2571, 0/0, 0/0                      │  imported : n/a        │
│       havoc : 0/0, 0/0                              │ stability : 100.00%    │
│        trim : 0.00%/7, 0.00%                        ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000: 12%]
```

BUUUGGGGSSSSS!

AFL Status Screen



```
                    american fuzzy lop 2.52b (boh)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 1 min, 11 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 9 sec        │  total paths : 123    │
│ last uniq crash : 0 days, 0 hrs, 0 min, 9 sec        │ uniq crashes : 3      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 16 (13.01%)       │    map density : 0.93% / 1.44%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 3.24 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 32/8         │ favored paths : 29 (23.58%)            │
│ stage execs : 340/1329 (25.58%)     │  new edges on : 39 (31.71%)            │
│ total execs : 120k                  │ total crashes : 3 (3 unique)           │
│  exec speed : 1735/sec              │  total tmouts : 3 (3 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 24/2000, 3/1993, 0/1979                │    levels : 2          │
│  byte flips : 0/250, 0/243, 0/229                    │   pending : 117        │
│ arithmetics : 8/13.9k, 0/919, 0/0                    │  pend fav : 28         │
│  known ints : 1/1273, 0/6712, 0/8932                 │ own finds : 118        │
│  dictionary : 18/19.9k, 42/21.2k, 0/1489             │  imported : n/a        │
│       havoc : 25/37.9k, 0/0                          │ stability : 100.00%    │
│        trim : 2.34%/60, 0.00%                        ├────────────────────────┘
└──────────────────────────────────────────────────────┘          [cpu000: 12%]
```

AFL Qemu mode

And when source code is not avaiable?

The target program can be runned inside QEMU that instruments the generated intermediate representation (TCG).

At https://github.com/andreafioraldi/afl
you can find a version with TCG block chaining cache in the parent [15] implemented in a thread safe way as a patch to QEMU 3.1.0

To start AFL in QEMu mode add the -Q parameter to the command line.

AFL LLVM Mode

With afl-gcc the instrumentation code is inserted in the produced asm file.
Compiler optimizations are hard to apply in this case so this cause a slowdown.
afl-gcc instrumentation is also x86 dependant.

With LLVM mode instrumentation is applied to the LLVM IR and to overcome this problem.

See /llvm_mode/README.llvm.

The related compiler is afl-clang-fast.

AFL Deferred instrumentation

In many application the initialization work can be time consuming.

The stop-at-main approach of the forkserver may not be enough.

With LLVM mode we can define a **deferred entry point** after main just putting this snipped in the code at the target location:

```
#ifdef __AFL_HAVE_MANUAL_CONTROL
  __AFL_INIT();
#endif
```

AFL Persistent mode

Fork() is too heavy? If the application that we are fuzzing is stateless we can use the **persistent mode** feature of afl-clang-fast.

A single long-lived process is reused to test multiple inputs.

```
while (__AFL_LOOP(1000)) {

    /* Read input data. */
    /* Call library code to be fuzzed. */
    /* Reset state. */

}
```

Trimming

File size has a dramatic impact on fuzzing. Starting with a good initial set is not enough sometimes, the fuzzer can increase the size of files in the queue anyways.

To revert this trend AFL, by itself, when starts to fuzz an input taken from the queue it tries to delete part of the input and see if this affect the instrumentation output. If a deletion does not affect the trace_bits checksum it is committed to disk.

Parallel fuzzing

AFL works on a single core.

To exploit all the power of your CPU you must run several AFL instances.

But how we can avoid to fuzz multiple times the same inputs? AFL does the job for us, it sync interesting testcases in the queue between different instances.

To do this simply specify the same output directory (-o afl_out) and give a role and a name for each fuzzer.

Parallel fuzzing

There are two roles: **master** (-M name) and **slave** (-S name).

A master instance perform all the steps discussed before. On the contrary, a slave instance perform only havoc and splice (and splice is executed always) without touching deterministic stages.

```
afl-fuzz -i initial_dir -o afl_out -M afl_master -- ./program
```

By experience you need only a master. In some cases you need only slaves cause deterministic stages are not interesting.

AFL Output directory

```
afl_out /
        [FUZZER_1] /
                crashes /
                hangs /
                queue /
                fuzz_bitmap (copy of virgin_bits)
                fuzzer_stats
                plot_data
        [FUZZER_2] /
                ...
```

Fuzzing Overview

American Fuzzy Lop

➡ **Useful strategies**

Testcase minimization

The **afl-tmin** tool can be employed, outside the fuzzer, to minimize a testcase in a better way. The steps are the following:

1. Fill with ‘0’ large blocks of data
2. Perform block deletion with decreasing size in a binary seach spirit
3. Alphabet normalization, count unique character and replace with ‘0’
4. Byte-by-byte normalization of not-’0’ bytes

Only the tweaks that produce the exactly bitmap of the starting input are keeped.

Corpus minimization

The **afl-cmin** is used for different type of minimization of all testcases in the queue.

It removes all the testcases that are obsolete in the queue (remind that we said that AFL keeps in queue also when find a new testcase with a superset coverage?)

How we must run afl-cmin? Tipically, by experience, wait until the master instance does the first cycle (a cycle is when all inputs in queue are yet fuzzed e AFL reiterate the queue). Usually the slaves instances performs many cycles during the time of one master cycle.

Corpus minimization

Just stop the fuzzers, run afl-cmin on the queue directory (save it before) and, optionally, run afl-tmin on all files (very time consuming, consider parallelization).

When a fuzzer is stopped, you can restart it specifying to use the queue as initial inputs directory.

```
afl-fuzz -i - -o afl_out -- ./program
```

Dictionary

The user can specify some dictionary keywords to be used during fuzzing.

The -x parameter allows you to specify a file that contains a list of interesting keywords.

For instance, if you are fuzzing an HTTP parser, you may include HTTP, GET, Host:, etc… as interesting keyword.

AFL insert this keywords both in the deterministic stages and the havoc stage.

LibTokenCap

But when we are not aware of this interesting keywords?

LibTokenCap can be used to instrument strcmp(), memcmp(), and related functions to automatically extract syntax tokens.

if (strcmp(input, “pippo”) == 0) goto new_path;

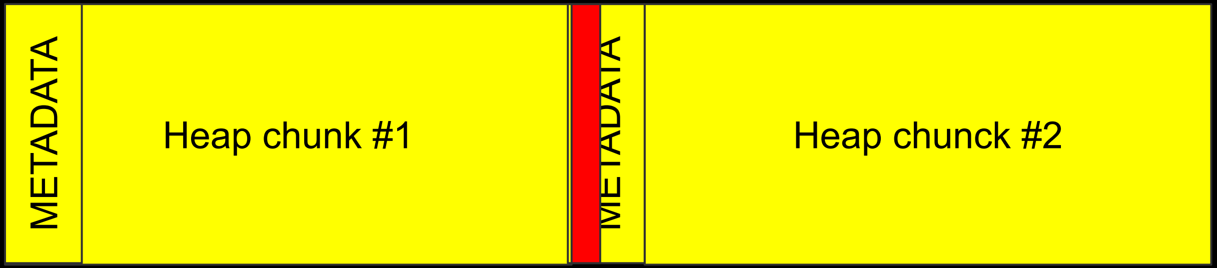
“pippo” is logged cause it is in a read-only section.

Load it with LD_PRELOAD=./libctokencap.so and set also AFL_TOKEN_FILE as output file.

ASAN / MSAN

Crash does not mean security issue and viceversa a security issue may not cause a crash.

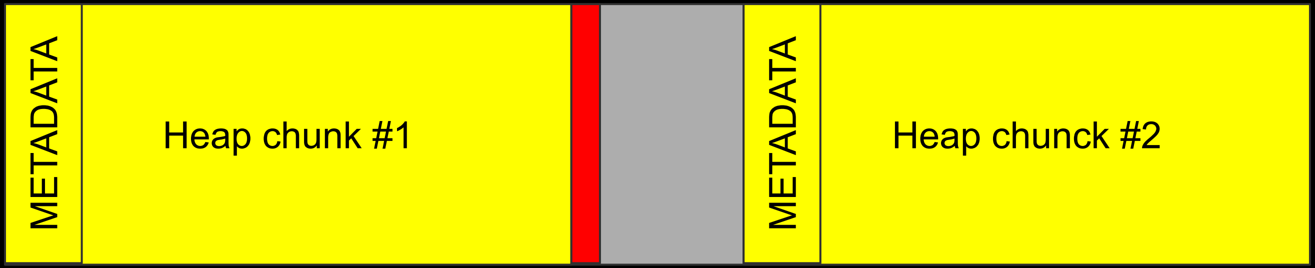
Consider this bug that causes a crash:



One byte overflow on chunk metadata

ASAN / MSAN

The following situation is a security issue that does not cause a crash

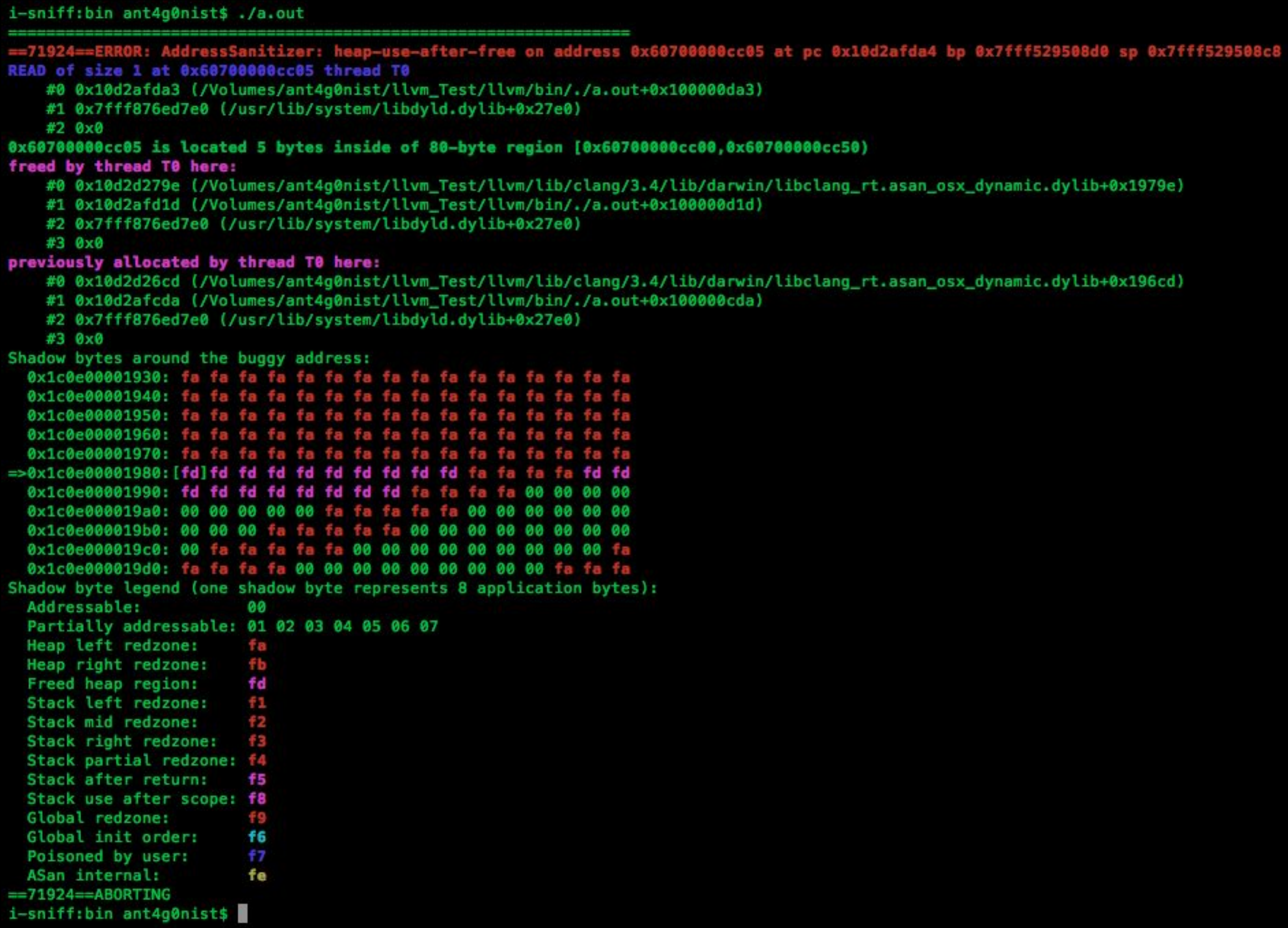


One byte overflow on usable memory

ASAN / MSAN

A very useful tool that developers uses to detect this type of bugs is the memcheck plugin of Valgrind [16].
However, this tool is expensive.
The LLVM toolchain offers similar facilities that, instead of dynamic instrumentation, are based on compile-time instrumentation.

They are AddressSanitizer [17] (ASAN), a fast memory errors detector, and MemorySanitizer [18] (MSAN) a detector of uninitialized reads.



```
i-sniff:bin ant4g0nist$ ./a.out
=================================================================
==71924==ERROR: AddressSanitizer: heap-use-after-free on address 0x60700000cc05 at pc 0x10d2afda4 bp 0x7fff529508d0 sp 0x7fff529508c8
READ of size 1 at 0x60700000cc05 thread T0
    #0 0x10d2afda3 (/Volumes/ant4g0nist/llvm_Test/llvm/bin/./a.out+0x100000da3)
    #1 0x7fff876ed7e0 (/usr/lib/system/libdyld.dylib+0x27e0)
    #2 0x0
0x60700000cc05 is located 5 bytes inside of 80-byte region [0x60700000cc00,0x60700000cc50)
freed by thread T0 here:
    #0 0x10d2d279e (/Volumes/ant4g0nist/llvm_Test/llvm/lib/clang/3.4/lib/darwin/libclang_rt.asan_osx_dynamic.dylib+0x1979e)
    #1 0x10d2afd1d (/Volumes/ant4g0nist/llvm_Test/llvm/bin/./a.out+0x100000d1d)
    #2 0x7fff876ed7e0 (/usr/lib/system/libdyld.dylib+0x27e0)
    #3 0x0
previously allocated by thread T0 here:
    #0 0x10d2d26cd (/Volumes/ant4g0nist/llvm_Test/llvm/lib/clang/3.4/lib/darwin/libclang_rt.asan_osx_dynamic.dylib+0x196cd)
    #1 0x10d2afcda (/Volumes/ant4g0nist/llvm_Test/llvm/bin/./a.out+0x100000cda)
    #2 0x7fff876ed7e0 (/usr/lib/system/libdyld.dylib+0x27e0)
    #3 0x0
Shadow bytes around the buggy address:
  0x1c0e00001930: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0e00001940: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0e00001950: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0e00001960: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0e00001970: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x1c0e00001980:[fd]fd fd fd fd fd fd fd fd fd fa fa fa fa fd fd
  0x1c0e00001990: fd fd fd fd fd fd fd fd fa fa fa fa 00 00 00 00
  0x1c0e000019a0: 00 00 00 00 00 fa fa fa fa fa 00 00 00 00 00 00
  0x1c0e000019b0: 00 00 00 fa fa fa fa fa 00 00 00 00 00 00 00 00
  0x1c0e000019c0: 00 fa fa fa fa fa 00 00 00 00 00 00 00 00 00 fa
  0x1c0e000019d0: fa fa fa fa 00 00 00 00 00 00 00 00 00 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
  Heap right redzone:    fb
  Freed heap region:     fd
  Stack left redzone:    f1
  Stack mid redzone:     f2
  Stack right redzone:   f3
  Stack partial redzone: f4
  Stack after return:    f5
  Stack use after scope: f8
  Global redzone:        f9
  Global init order:     f6
  Poisoned by user:      f7
  ASan internal:         fe
==71924==ABORTING
i-sniff:bin ant4g0nist$
```

ASAN/MSAN

This facilities can be enabled during the compilation setting the following env variables:

AFL_USE_ASAN=1   AFL_USE_MSAN=1

Their overhead are high for fuzzing! To avoid explosion of your RAM when using ASAN compile the target program as 32 bit.

LibDislocator

ASAN too expensive? AFL ships the poor man library for heap-based memory errors detection!

It replace libc functions like malloc(), calloc(), free() via LD_PRELOAD (like tokencap).

It is slow and memory consuming (all buffer allocated are placed before a mmapped region with PROT_NONE to detect r/w overflow).

Also heap canaries are employed to detect negative writes (but not reads).

Useful additional tools

- **afl-cov** to visualize coverage explored by AFL

  https://github.com/mrash/afl-cov

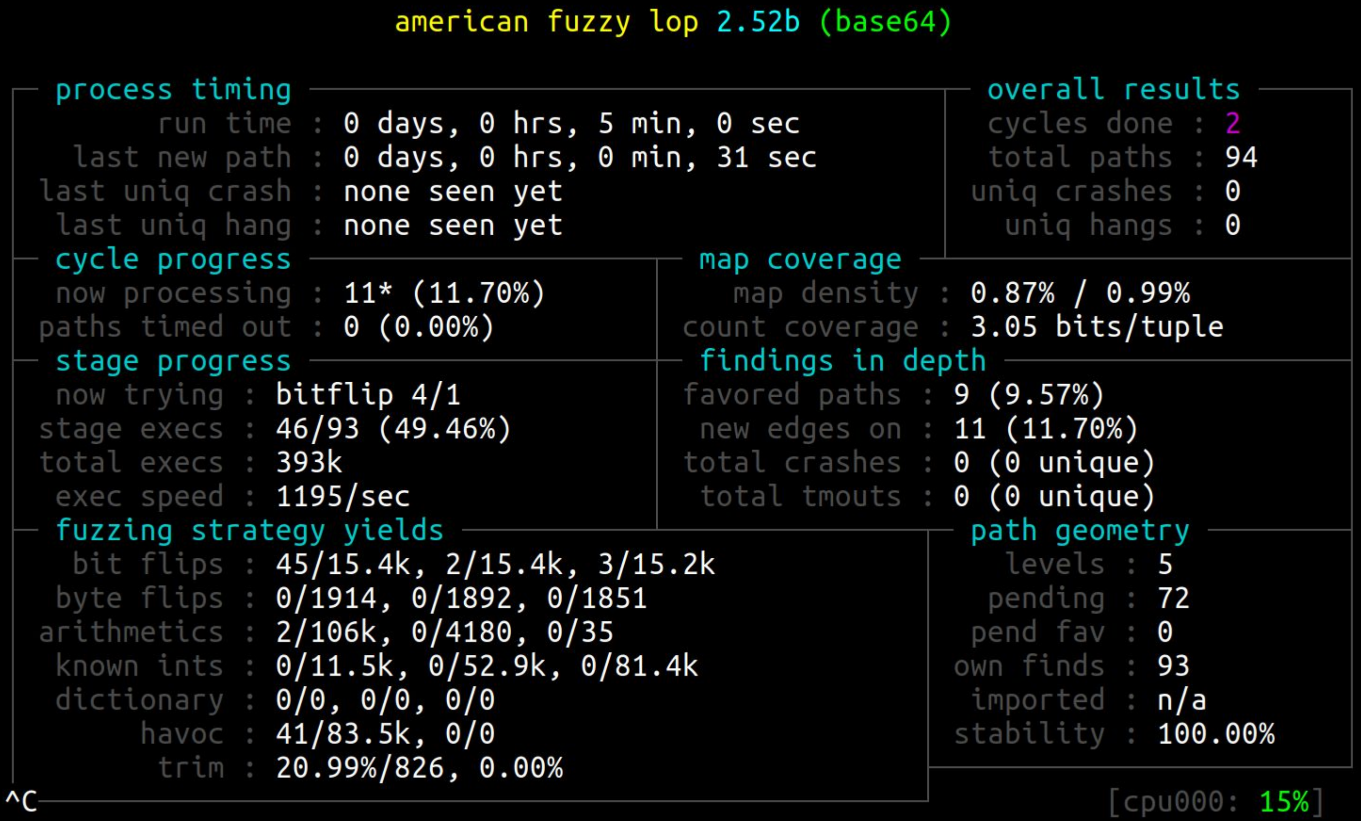
- **crashwalk** for crashes deduplication

  https://github.com/bnagy/crashwalk

- **afl-ddmin-mod** for a sophisticated minimization tool (slower than afl-tmin)

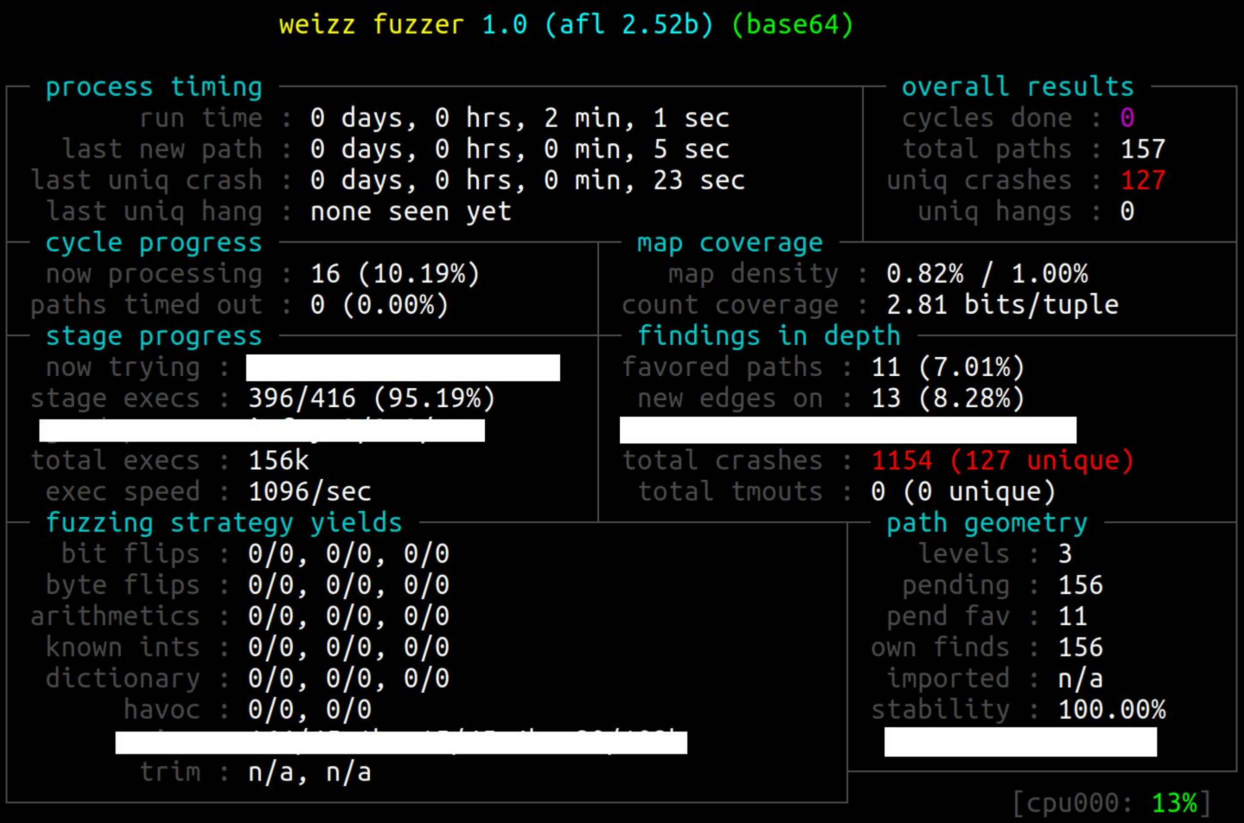
  https://github.com/MarkusTeufelberger/afl-ddmin-mod

The shit that I’m writing



```
                american fuzzy lop 2.52b (base64)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 5 min, 0 sec        │  cycles done : 2     │
│   last new path : 0 days, 0 hrs, 0 min, 31 sec       │  total paths : 94    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────┬─ map coverage ─────┴──────────────────────┤
│  now processing : 11* (11.70%)  │    map density : 0.87% / 0.99%           │
│ paths timed out : 0 (0.00%)     │ count coverage : 3.05 bits/tuple         │
├─ stage progress ────────────────┼─ findings in depth ───────────────────────┤
│  now trying : bitflip 4/1       │ favored paths : 9 (9.57%)                │
│ stage execs : 46/93 (49.46%)    │  new edges on : 11 (11.70%)              │
│ total execs : 393k              │ total crashes : 0 (0 unique)             │
│  exec speed : 1195/sec          │  total tmouts : 0 (0 unique)             │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ──────────┤
│   bit flips : 45/15.4k, 2/15.4k, 3/15.2k        │    levels : 5            │
│  byte flips : 0/1914, 0/1892, 0/1851            │   pending : 72           │
│ arithmetics : 2/106k, 0/4180, 0/35              │  pend fav : 0            │
│  known ints : 0/11.5k, 0/52.9k, 0/81.4k         │ own finds : 93           │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a          │
│       havoc : 41/83.5k, 0/0                     │ stability : 100.00%      │
│        trim : 20.99%/826, 0.00%                 ├──────────────────────────┘
^C──────────────────────────────────────────────────┘          [cpu000: 15%]
```

The shit that I'm writing



```
                    weizz fuzzer 1.0 (afl 2.52b) (base64)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 2 min, 1 sec         │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 5 sec         │  total paths : 157    │
│ last uniq crash : 0 days, 0 hrs, 0 min, 23 sec        │ uniq crashes : 127    │
│  last uniq hang : none seen yet                       │   uniq hangs : 0      │
├─ cycle progress ─────────────────┬─ map coverage ─────┴───────────────────────┤
│  now processing : 16 (10.19%)     │    map density : 0.82% / 1.00%             │
│ paths timed out : 0 (0.00%)       │ count coverage : 2.81 bits/tuple           │
├─ stage progress ─────────────────┼─ findings in depth ────────────────────────┤
│  now trying : [redacted]          │ favored paths : 11 (7.01%)                 │
│ stage execs : 396/416 (95.19%)    │  new edges on : 13 (8.28%)                 │
│ [redacted]                        │ [redacted]                                 │
│ total execs : 156k                │ total crashes : 1154 (127 unique)          │
│  exec speed : 1096/sec            │  total tmouts : 0 (0 unique)               │
├─ fuzzing strategy yields ────────┴───────────────┬─ path geometry ────────────┤
│   bit flips : 0/0, 0/0, 0/0                       │    levels : 3              │
│  byte flips : 0/0, 0/0, 0/0                       │   pending : 156            │
│ arithmetics : 0/0, 0/0, 0/0                       │  pend fav : 11             │
│  known ints : 0/0, 0/0, 0/0                       │ own finds : 156            │
│  dictionary : 0/0, 0/0, 0/0                       │  imported : n/a            │
│       havoc : 0/0, 0/0                            │ stability : 100.00%        │
│ [redacted]                                        │ [redacted]                 │
│        trim : n/a, n/a                            ├────────────────────────────┘
└──────────────────────────────────────────────────┘          [cpu000: 13%]
```

Thank you!

QUESTIONS?

References

[1] SAGE - https://dl.acm.org/citation.cfm?id=2094081

[2] Radamsa - https://gitlab.com/akihe/radamsa

[3] Compare Coverage - https://github.com/googleprojectzero/CompareCoverage

[4] Vuzzer -

http://sharcs-project.eu/m/filer_public/48/8c/488c5fb7-9aad-4c87-ab9c-5ff251ebc73d/vuzzer_ndss17.pdf

[5] Angora - https://arxiv.org/abs/1803.01307

[6] Peach - http://www.peach.tech/resources/peachcommunity/

[7] AFLSmart - https://github.com/aflsmart/aflsmart

[8] libFuzzer - https://llvm.org/docs/LibFuzzer.html

[9] honggfuzz - https://github.com/google/honggfuzz

[10] Driller - https://www.cs.ucsb.edu/~vigna/publications/2016_NDSS_Driller.pdf

References

[11] QSYM - https://github.com/sslab-gatech/qsym

[12] AFL whitepaper - http://lcamtuf.coredump.cx/afl/technical_details.txt

[13] Binary fuzzing strategies - https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html

[14] Fuzzing random programs without execve -

https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html

[15] Improving AFL's QEMU mode performance - https://abiondo.me/2018/09/21/improving-afl-qemu-mode/

[16] Memcheck: a memory error detector - http://valgrind.org/docs/manual/mc-manual.html

[17] AddressSanitizer - https://clang.llvm.org/docs/AddressSanitizer.html

[18] MemorySanitizer - https://clang.llvm.org/docs/MemorySanitizer.html