NEU CY 5770 Software Vulnerabilities and Security

Instructor: Dr. Ziming Zhao

What we have learned so far?

- Vulnerabilities
  - Buffer overflow (out of bound write), format string vulnerability, heap vulnerabilities, race conditions, …
- Exploitation techniques
  - Return to Libc
  - Shellcode development
  - Return-oriented programming
  - …
- Mitigations at hardware, OS, compiler layers
  - Non-execution stack
  - Control-flow integrity
  - ASLR
  - …

The workflow of exploitation

1. Discover and understand the vulnerabilities or bugs
   a. code review, reverse engineering
   b. root cause, what memory is affected, memory address calculation
2. Assess exploitability
   a. what do we control? what mitigations are active?
3. Choose technique and develop exploit
   a. based on mitigations and available primitives
   b. craft payload, bypass protections, gain control

The workflow of exploitation

1. Discover and understand the vulnerabilities or bugs
    a. code review, reverse engineering
    b. root cause, what memory is affected, memory address calculation
2. Assess exploitability
    a. what do we control? what mitigations are active?
3. Choose technique and develop exploit
    a. based on mitigations and available primitives
    b. craft payload, bypass protections, gain control

overflowret4_32: short and simple

```
int vulfoo()
{
  char buf[40];

  gets(buf);
  return 0;
}

int main(int argc, char *argv[])
{
  vulfoo();
  printf("I pity the fool!\n");
}
```

Manually discover real-world bugs is not alway easy

```
static void
xmlSnprintfElementContent(char *buf, int size,
                    xmlElementContentPtr content, int englob)
{
    int len = strlen(buf);   // (1) how much is in the buffer

    if (len > size - 10) return;    // (2) safety check
    switch (content->type) {
        case XML_ELEMENT_CONTENT_PCDATA:
            strcat(buf, "#PCDATA");  // (3) append "#PCDATA"
            break;
        case XML_ELEMENT_CONTENT_ELEMENT:
            // ... append element name ...
            strcat(buf, content->name);
            break;
```

READ THE NEXT CODE BLOCK ⇒

```
        case XML_ELEMENT_CONTENT_SEQ:
        case XML_ELEMENT_CONTENT_OR:
            if (englob) strcat(buf, "(");          // (4) append "("
            xmlSnprintfElementContent(buf, size,
                             content->c1, 0);  // (5) recurse
            if (content->type == XML_ELEMENT_CONTENT_SEQ)
                strcat(buf, " , ");               // (6) append
            else
                strcat(buf, " | ");
            xmlSnprintfElementContent(buf, size,
                             content->c2, 0);  // (7) recurse again
            if (englob) strcat(buf, ")");          // (8) append ")"
            break;
    }
}
```

CVE-2017-9047 — libxml2, xmlSnprintfElementContent() in valid.c

A widely-used C library for parsing XML. Used by almost every Linux application, browser, and server that touches XML or HTML.

https://nvd.nist.gov/vuln/detail/cve-2017-9047

Manually discover real-world bugs is not alway easy

```
static void
xmlSnprintfElementContent(char *buf, int size,
                          xmlElementContentPtr content, int englob)
{
    int len = strlen(buf);   // (1) how much is in the buffer

    if (len > size - 10) return;    // (2) safety check
    switch (content->type) {
        case XML_ELEMENT_CONTENT_PCDATA:
            strcat(buf, "#PCDATA");  // (3) append "#PCDATA"
            break;
        case XML_ELEMENT_CONTENT_ELEMENT:
            // ... append element name ...
            strcat(buf, content->name);
            break;

READ THE NEXT CODE BLOCK ⇒
```

```
        case XML_ELEMENT_CONTENT_SEQ:
        case XML_ELEMENT_CONTENT_OR:
            if (englob) strcat(buf, "(");          // (4) append "("
            xmlSnprintfElementContent(buf, size,
                                      content->c1, 0);  // (5) recurse
            if (content->type == XML_ELEMENT_CONTENT_SEQ)
                strcat(buf, " , ");                // (6) append
            else
                strcat(buf, " | ");
            xmlSnprintfElementContent(buf, size,
                                      content->c2, 0);  // (7) recurse again
            if (englob) strcat(buf, ")");          // (8) append ")"
            break;
    }
}
```

Entry: len = size - 11 → check passes (size-11 < size-10) ✓
Step 5: recurse fills buffer to size - 4
Step 6: strcat(buf, " , ") → writes 3 bytes → buffer now at size - 1
Step 8: strcat(buf, ")") → writes 1 byte → OOB write ✗

Other reasons why manually discovering bugs is not feasible

Real-world codebases have millions of lines of code

- Linux kernel — ~40 million lines of code, growing by ~70,000 lines every release cycle
- Windows 10 — ~50 million lines of code
- macOS — over 80 million lines of code
- Google Chromium — ~20–30 million lines of code
- Android AOSP — ~7 million lines (core only, excluding external dependencies)

The workflow of exploitation

Automate bug discovery at scale

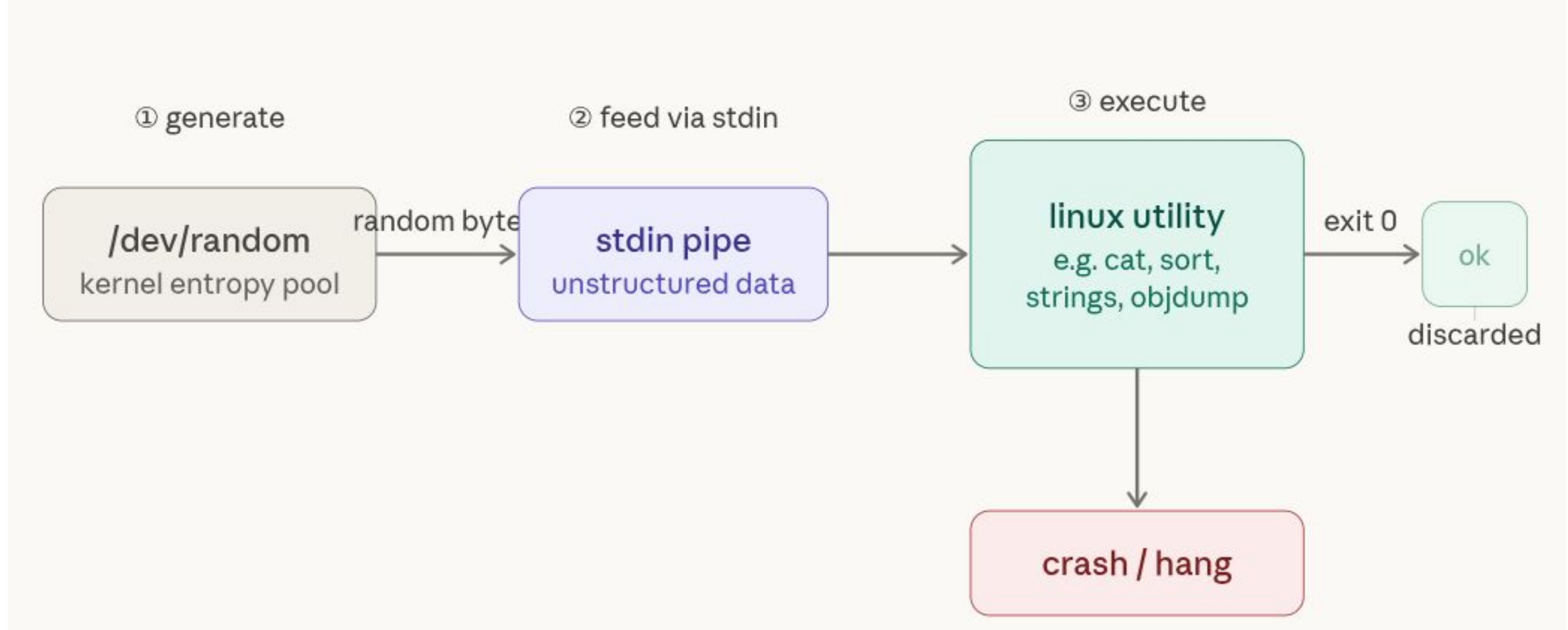
1. ugs
    b. root cause, wha y is affected, memory address calculation
2. Assess exploitability
    a. what do we control? what mitigations are active?
3. Choose technique and develop exploit
    a. based on mitigations and available primitives
    b. craft payload, bypass protections, gain control

Fuzzing or Fuzz Testing

What is fuzzing

- Automated testing
  - Feed unexpected, random, or mutated inputs to a program/function that reveal a bug
- Bug discovery
  - Detect crashes, hangs, memory errors, and undefined behavior
- Have been applied to all kinds of software, e.g., web, network protocol.
  - We focus on binary/native software in this class

In its simplest form: naive fuzzing



In its simplest form: naive fuzzing

```
TARGET="sort"   # change this to any util

while true; do
    cat /dev/urandom | head -c 500 | $TARGET > /dev/null 2>&1
    STATUS=$?
    if [ $STATUS -eq 139 ]; then
        echo "$TARGET crashed with SIGSEGV!"
    fi
done
```

exit code < 128: program exited normally (intentional exit) as in “return 0;” “return 1;”
exit code > 128: program was killed by a signal (crash/interrupt)

ARTICLE | FREE ACCESS

An empirical study of the reliability of UNIX utilities

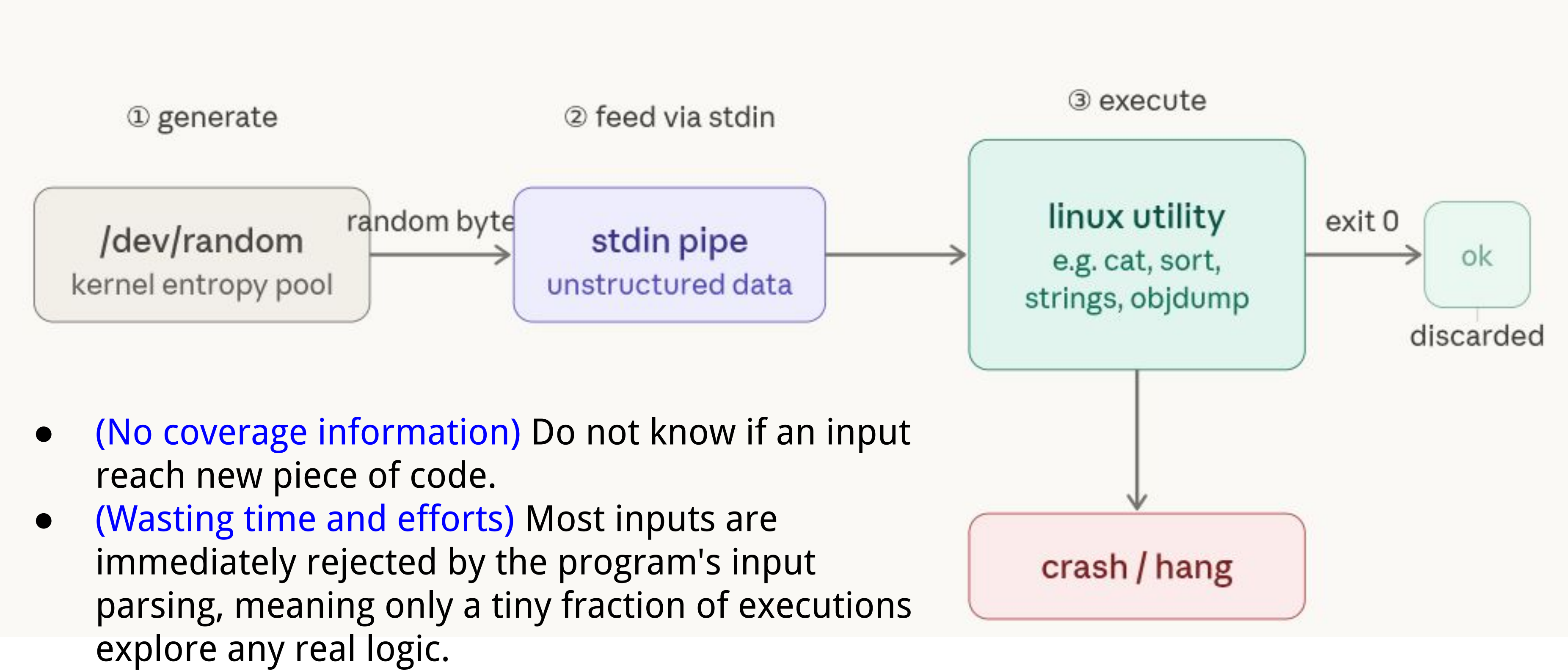
Authors: Barton P. Miller, Lars Fredriksen, Bryan So | Authors Info & Claims

Communications of the ACM, Volume 33, Issue 12 • Pages 32 - 44 • https://doi.org/10.1145/96267.96279

Published: 01 December 1990 Publication History

https://dl.acm.org/doi/10.1145/96267.96279

In its simplest form: naive fuzzing



- (No coverage information) Do not know if an input reach new piece of code.
- (Wasting time and efforts) Most inputs are immediately rejected by the program's input parsing, meaning only a tiny fraction of executions explore any real logic.
- (No mutation based on ‘good’ input) Every attempt is independent and blind.

The input space

The input space - infinite

Input that can trigger a particular bug

Input that can reach specific logic code

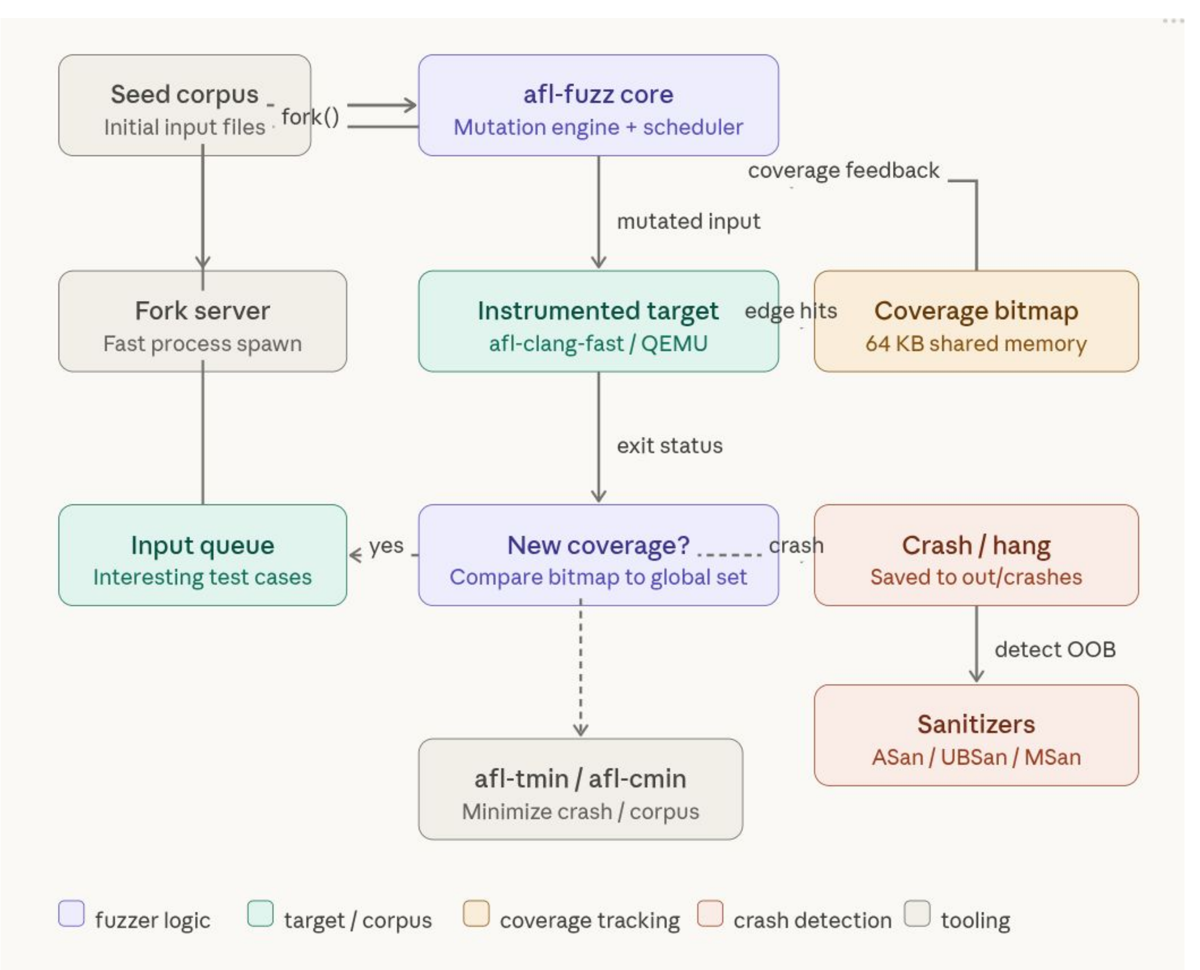
Modern Coverage-based Fuzzing

History of AFL/AFL++

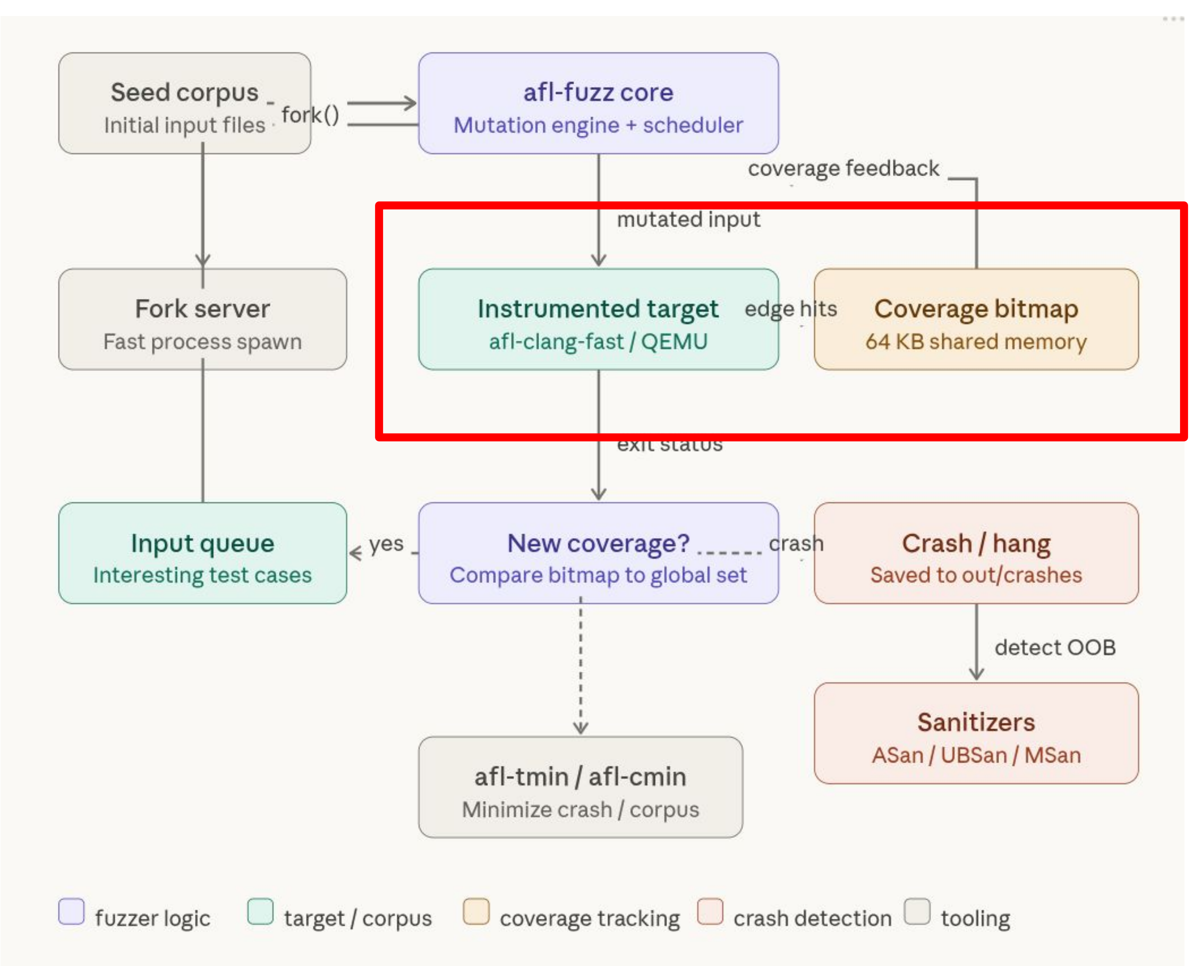
- AFL was created by Michał Zalewski at Google and first publicly released on November 12, 2013. It introduced the coverage-guided approach that became the foundation of modern fuzzing.
- AFL++ originated in 2019 as a community-driven fork after primary development of AFL ceased following Zalewski's departure from Google.
- AFL++ is now what everyone actually uses. It is faster, smarter, and actively maintained, while the original AFL is essentially frozen in time.

```
                american fuzzy lop 1.86b (test)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 2 sec        │  cycles done : 0      │
│   last new path : none seen yet                      │  total paths : 1      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 2 sec        │ uniq crashes : 1      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────┬─ map coverage ─────┴───────────────────────┤
│  now processing : 0 (0.00%)     │    map density : 2 (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 (100.00%)                │
│ stage execs : 1464/5000 (29.28%)│  new edges on : 1 (100.00%)                │
│ total execs : 1697              │ total crashes : 39 (1 unique)              │
│  exec speed : 626.5/sec         │   total hangs : 0 (0 unique)               │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ────────────┤
│   bit flips : 0/16, 1/15, 0/13                  │    levels : 1              │
│  byte flips : 0/2, 0/1, 0/0                     │   pending : 1              │
│ arithmetics : 0/112, 0/25, 0/0                  │  pend fav : 1              │
│  known ints : 0/10, 0/28, 0/0                   │ own finds : 0              │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a            │
│       havoc : 0/0, 0/0                          │  variable : 0              │
│        trim : n/a, 0.00%                        ├────────────────────────────┘
└─────────────────────────────────────────────────┘             [cpu: 92%]
```

AFL++ Architecture



AFL++ Architecture



overflowlocal7_64

```
int vulfoo()
{
	int is_admin = 0;
	char buf[16];

	fread(buf, 1, 32, stdin);

	if (strncmp(buf, "admin", 5) == 0)
		is_admin = 1;

	if (is_admin)
		print_flag();

	return 0;
}

int main(int argc, char *argv[])
{
	vulfoo();
}
```

```
int print_flag()
{
	FILE *fp = NULL;
	char buff[MAX_FLAG_SIZE] = {0};

	fp = fopen("/flag", "r");

	if (fp == NULL)
	{
		printf("Error: Cannot open the flag 
file!!!\n");
		return 1;
	}

	fread(buff, MAX_FLAG_SIZE - 2, 1, fp);
	printf("The flag is: %s\n", buff);
	fclose(fp);
	return 0;
}
```

Makefile

```
CFLAGS=-Wall -O0 -fno-stack-protector
EXPODIR=.
_dummy:=$(shell mkdir -p ../../../software-security-course-binaries/bufferoverflow)
INCLUDEDIR=../../commons

$(EXPODIR)/overflowlocal7_gcc_64: *.c
	gcc $(CFLAGS) -I$(INCLUDEDIR) main.c -o $@

$(EXPODIR)/overflowlocal7_llvm_64: *.c
	clang $(CFLAGS) -I$(INCLUDEDIR) main.c -o $@

$(EXPODIR)/overflowlocal7_afl_64: *.c
	afl-clang-fast -I$(INCLUDEDIR) -O0 -fno-stack-protector main.c -o $@

$(EXPODIR)/overflowlocal7_afl_asan_64: *.c
	afl-clang-fast -I$(INCLUDEDIR) -O0 -fsanitize=address -fno-stack-protector main.c -o $@
```

AFL++ Instrumentation

afl-clang-fast

- LLVM pass-based AFL inserts instrumentation at compile time
- Goal: track which branch edges are covered by each input
  - *A branch is any point where control flow can go to more than one place depending on a condition*
- Uses a shared memory bitmap to record edge transitions
- Each edge = transition from one basic block to another
- New edge covered → input is interesting → saved for mutation

```
afl-clang-fast -O0 main.c -o ./binary
```

```
afl-clang-fast -I../../commons -O0 -fno-stack-protector main.c -o overflowlocal7_afl_64
afl-clang-fast++2.59d by <lszekeres@google.com>
afl-llvm-pass++2.59d by <lszekeres@google.com>
[+] Instrumented 6 locations (non-hardened mode, ratio 100%).
```

Why 6 instrumentation locations?

```c
int vulfoo()
{
	int is_admin = 0;
	char buf[16];

	fread(buf, 1, 32, stdin); // L1

	if (strncmp(buf, "admin", 5) == 0) // L2
		is_admin = 1;

	if (is_admin) // L3
		print_flag(); // L4

	return 0;
}

int main(int argc, char *argv[])
{
	vulfoo();
}
```

```c
int print_flag()
{
	FILE *fp = NULL;
	char buff[MAX_FLAG_SIZE] = {0};

	fp = fopen("/flag", "r"); // L5

	if (fp == NULL) // L6
	{
		printf("Error: Cannot open the flag 
file!!!\n"); // L7
		return 1; // L8
	}

	fread(buff, MAX_FLAG_SIZE - 2, 1, fp);
	printf("The flag is: %s\n", buff);
	fclose(fp);
	return 0;
}
```

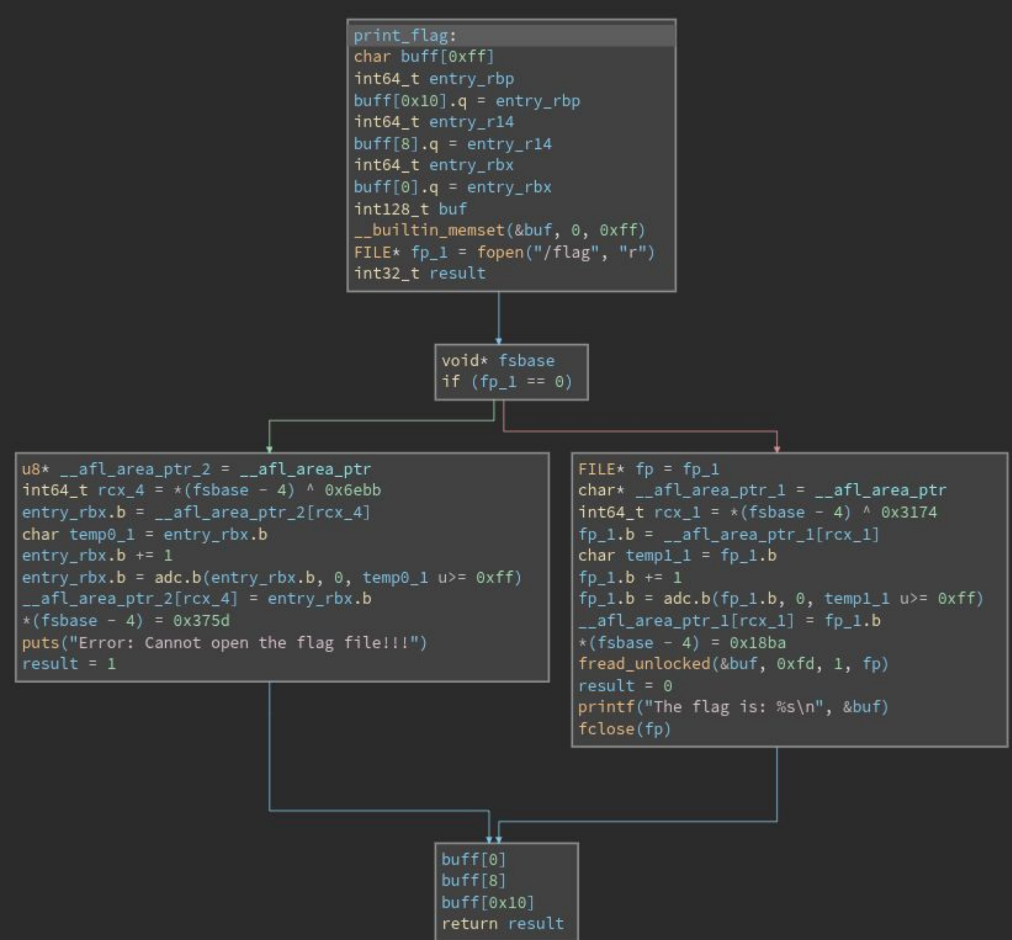
vulfoo() - 4 instrumentation locations

```
                 fread()
                    |
        ____________________________
        |                          |
 strncmp == "admin"        strncmp != "admin"
  is_admin = 1               is_admin = 0
        |                          |
   if(is_admin)               if(is_admin)
      TRUE                       FALSE
        |                          |
   print_flag()               return 0
        |                      [PATH 3]
  _______________
  |             |
fopen OK      fopen FAIL
print flag    print error
[PATH 1]      [PATH 2]
```

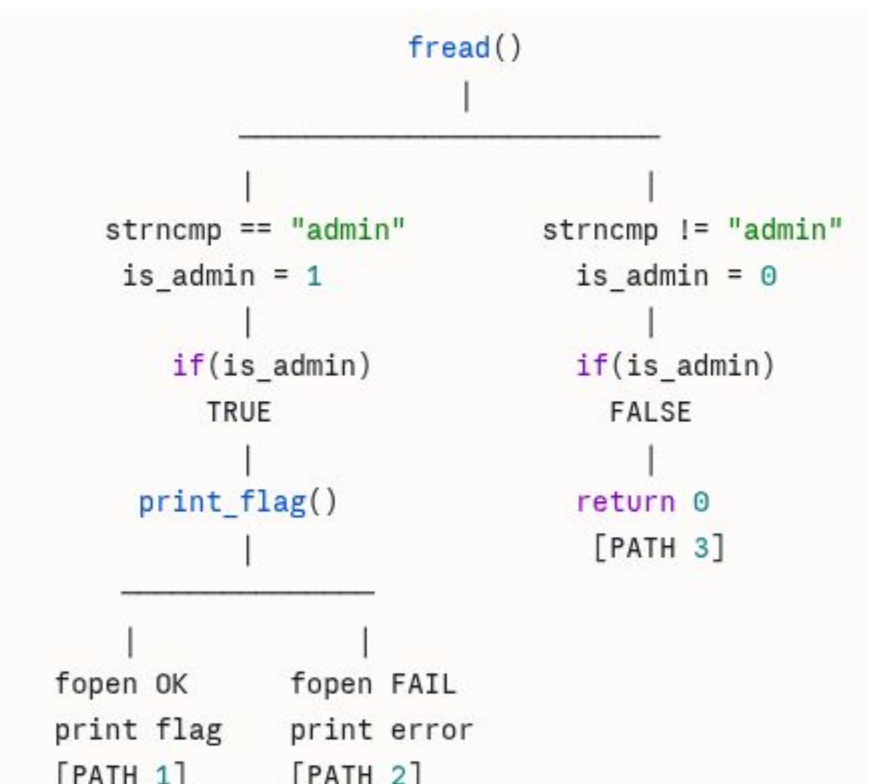
```
vulfoo:
char buf[0x10]
int64_t entry_r14
buf[8].q = entry_r14
FILE* entry_rbx
buf[0].q = entry_rbx
int32_t buf_1
fread(&buf_1, 1, 0x14, __TMC_END__)
char var_124
void* fsbase
if (((var_124 ^ 0x6e) | (0x696d6461 ^ buf_1)) == 0)

u8* __afl_area_ptr_1 = __afl_area_ptr
int64_t rax_2 = *(fsbase - 4) ^ 0xafc
char rdx_1 = __afl_area_ptr_1[rax_2]
__afl_area_ptr_1[rax_2] = adc.b(rdx_1 + 1, 0, rdx_1 u>= 0xff)
*(fsbase - 4) = 0x57e
int128_t buf_2
__builtin_memset(&buf_2, 0, 0xff)
FILE* fp = fopen("/flag", "r")

char rdx_2
if (fp == 0)

u8* __afl_area_ptr_3 = __afl_area_ptr
int64_t rax_7 = *(fsbase - 4) ^ 0x3767
rdx_2 = __afl_area_ptr_3[rax_7]
__afl_area_ptr_3[rax_7] = adc.b(rdx_2 + 1, 0, rdx_2 u>= 0xff)
*(fsbase - 4) = 0x1bb3
puts("Error: Cannot open the flag file!!!")

u8* __afl_area_ptr_2 = __afl_area_ptr
int64_t rax_4 = *(fsbase - 4) ^ 0xb496
rdx_2 = __afl_area_ptr_2[rax_4]
__afl_area_ptr_2[rax_4] = adc.b(rdx_2 + 1, 0, rdx_2 u>= 0xff)
*(fsbase - 4) = 0x5a4b
fread_unlocked(&buf_2, 0xfd, 1, fp)
printf("The flag is: %s\n", &buf_2)
fclose(fp)

char* __afl_area_ptr_4 = __afl_area_ptr
int64_t rcx_6 = *(fsbase - 4) ^ 0x65a7
entry_rbx.b = __afl_area_ptr_4[rcx_6]
char temp0 = entry_rbx.b
entry_rbx.b += 1
entry_rbx.b = adc.b(entry_rbx.b, 0, temp0 u>= 0xff)
__afl_area_ptr_4[rcx_6] = entry_rbx.b
```

print_flag() - 2 instrumentation locations

```
  print_flag()
       |
 ______________
  |          |
fopen OK   fopen FAIL
print flag print error
[PATH 1]   [PATH 2]
```



Three paths of this program



What is instrumented? The Coverage Bitmap Formula

```
// Each instrumented basic block gets a random ID (cur_location)
// AFL updates the bitmap on every edge transition:

bitmap[prev_loc ^ cur_location]++
prev_loc = cur_location >> 1
```

What is the AFL Bitmap?

The bitmap is a **64KB** shared memory array of 65536 unsigned bytes, where **each byte represents the hit count of one edge (branch transition)** in each execution of the program. Bitmap is reset after each run. Virgin_map does not reset.

```
// from afl source — config.h
#define MAP_SIZE_POW2 16
#define MAP_SIZE (1 << MAP_SIZE_POW2) // 65536 bytes

uint8_t bitmap[MAP_SIZE]; // one byte per edge slot
```

bitmap[i] == 0 → edge never taken
bitmap[i] == 1 → edge taken once
bitmap[i] == 255 → edge taken 255+ times (saturated)

Why prev_loc = cur_location >> 1?

// The >> 1 trick makes A→B and B→A distinguishable edges

The XOR of two block IDs uniquely identifies the edge between them

// Without it: A^B == B^A (same bucket, collision!)

edge A→B: bitmap[A ^ B]++
edge B→A: bitmap[B ^ A]++

A ^ B == B ^ A   ← XOR is commutative!

Dissect an instrumentation point - *vulfoo() if (is_admin) taken*

```
mov  r14, qword ptr [rip + __afl_prev_loc@GOTTPOFF]
movsxd rax, dword ptr fs:[r14]
mov  rcx, qword ptr [rip + __afl_area_ptr]
xor  rax, 29006           ; cur_location = 29006
mov  dl, byte ptr [rcx + rax]
add  dl, 1
adc  dl, 0
mov  byte ptr [rcx + rax], dl
mov  dword ptr fs:[r14], 14503   ; prev_loc = 29006>>1
```

What do rax and rcx hold?

Dissect an instrumentation point - *vulfoo() if (is_admin) taken*

```
mov  r14, qword ptr [rip + __afl_prev_loc@GOTTPOFF]
movsxd rax, dword ptr fs:[r14]
mov  rcx, qword ptr [rip + __afl_area_ptr]
xor  rax, 29006          ; cur_location
mov  dl, byte ptr [rcx + rax]
add  dl, 1
adc  dl, 0
mov  byte ptr [rcx + rax], dl
mov  dword ptr fs:[r14], 14503   ; prev_loc
```

What is 29006?

Dissect an instrumentation point - *vulfoo() if (is_admin) taken*

```
mov  r14, qword ptr [rip + __afl_prev_loc@GOTTPOFF]
movsxd rax, dword ptr fs:[r14]
mov  rcx, qword ptr [rip + __afl_area_ptr]
xor  rax, 29006          ; cur_location = 29006
mov  dl, byte ptr [rcx + rax]
add  dl, 1
adc  dl, 0
mov  byte ptr [rcx + rax], dl
mov  dword ptr fs:[r14], 14503   ; prev_loc = 29006>>1
```

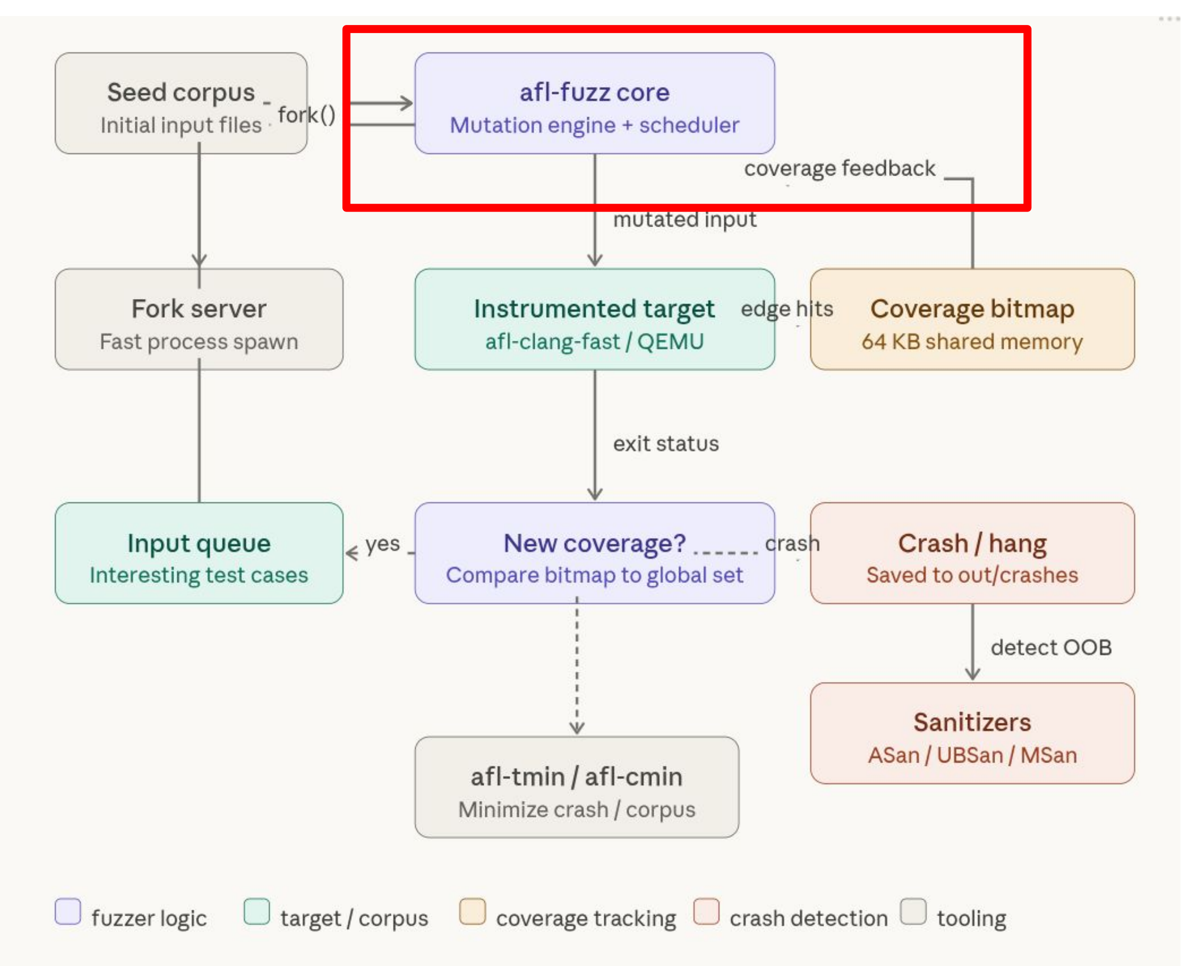
ADC?

Dissect an instrumentation point - *vulfoo() if (is_admin) taken*

```
mov  r14, qword ptr [rip + __afl_prev_loc@GOTTPOFF]
movsxd rax, dword ptr fs:[r14]
mov  rcx, qword ptr [rip + __afl_area_ptr]
xor  rax, 29006          ; cur_location = 29006
mov  dl, byte ptr [rcx + rax]
add  dl, 1
adc  dl, 0
mov  byte ptr [rcx + rax], dl
mov  dword ptr fs:[r14], 14503   ; prev_loc = 29006>>1
```

Update prev_loc

AFL++ Architecture



AFL mutation strategies

How AFL transforms seed inputs into bug-finding inputs

For each queue input:

- stage 1: bit flips ← deterministic, systematic
- stage 2: byte flips ← deterministic, systematic
- stage 3: arithmetics ← deterministic, systematic
- stage 4: known ints ← deterministic, systematic
- stage 5: havoc ← RANDOM, chaotic
- stage 6: splice ← combine two inputs

Fuzz command

```
afl-fuzz -i inputs -o outputs -- ./target_binary
```

```
                 american fuzzy lop 2.57b (overflowlocal7_afl_64)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 19 sec       │  cycles done : 53     │
│   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 : 1 (50.00%)        │    map density : 0.00% / 0.01%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : splice 9              │ favored paths : 2 (100.00%)            │
│ stage execs : 47/48 (97.92%)        │  new edges on : 2 (100.00%)            │
│ total execs : 90.9k                 │ total crashes : 0 (0 unique)           │
│  exec speed : 4762/sec              │  total tmouts : 2 (2 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/72, 0/70, 0/66                      │    levels : 1          │
│  byte flips : 0/9, 0/7, 0/3                         │   pending : 0          │
│ arithmetics : 0/503, 0/0, 0/0                       │  pend fav : 0          │
│  known ints : 0/51, 0/194, 0/132                    │ own finds : 0          │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 0/32.8k, 0/56.9k                      │ stability : 100.00%    │
│        trim : 10.00%/2, 0.00%                       ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 63%]
```

Why not finding bugs/crashes?

```
int vulfoo()
{
	int is_admin = 0;
	char buf[16];

	fread(buf, 1, 32, stdin);

	if (strncmp(buf, "admin", 5) == 0)
		is_admin = 1;

	if (is_admin)
		print_flag();

	return 0;
}

int main(int argc, char *argv[])
{
	vulfoo();
}
```

```
int print_flag()
{
	FILE *fp = NULL;
	char buff[MAX_FLAG_SIZE] = {0};

	fp = fopen("/flag", "r");

	if (fp == NULL)
	{
		printf("Error: Cannot open the flag
file!!!\n");
		return 1;
	}

	fread(buff, MAX_FLAG_SIZE - 2, 1, fp);
	printf("The flag is: %s\n", buff);
	fclose(fp);
	return 0;
}
```

Runtime Sanitizers

Turn silent bugs into loud crashes

What is a Sanitizer?

- A compiler feature that instruments your binary to detect bugs at runtime. Part of LLVM/Clang and GCC.
- Turns silent memory corruption into immediate loud crashes. Acts as an oracle — tells the fuzzer 'this input triggered bad behavior'

- Enabled at compile time via -fsanitize=... flags

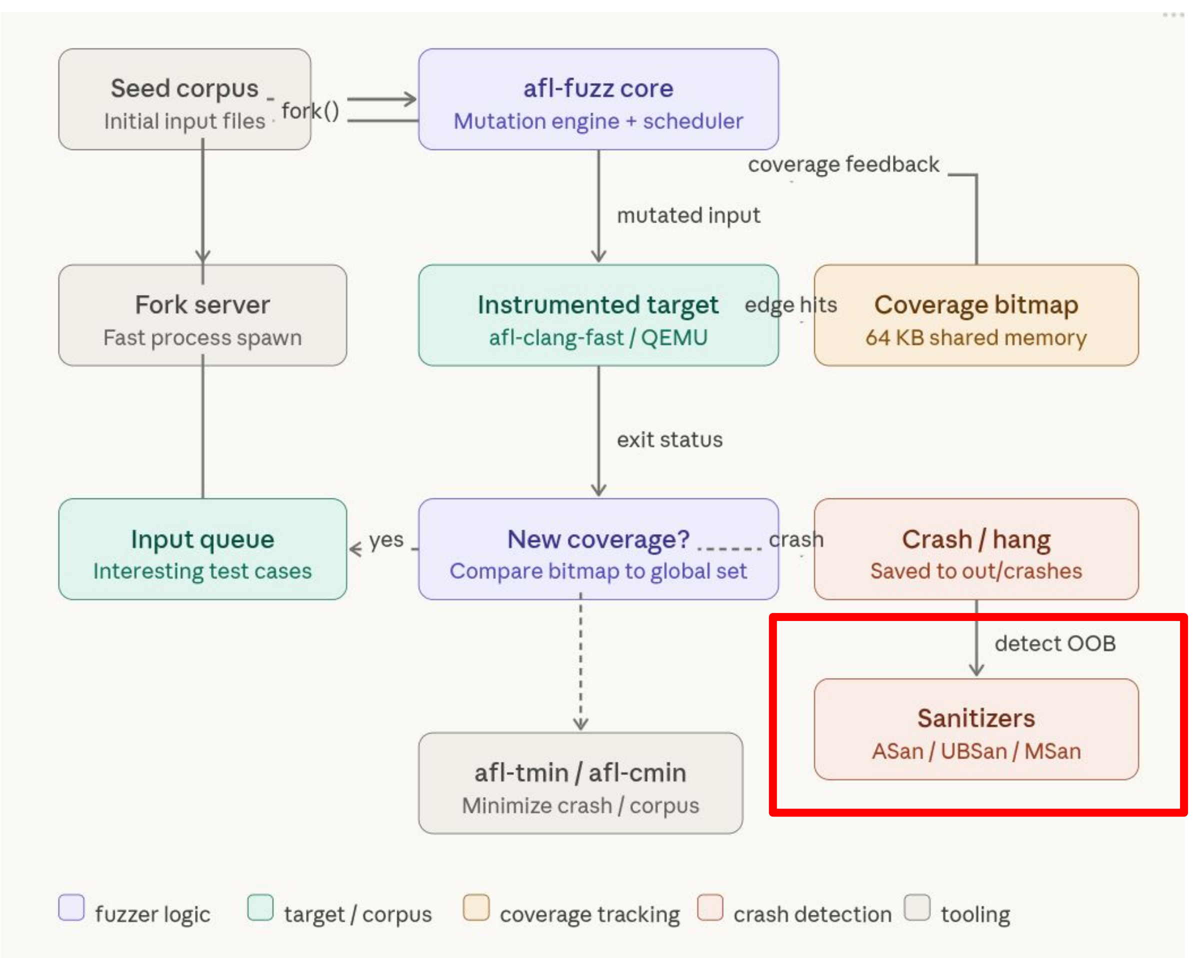
Makefile

```
CFLAGS=-Wall -O0 -fno-stack-protector
EXPODIR=.
_dummy:=$(shell mkdir -p ../../../software-security-course-binaries/bufferoverflow)
INCLUDEDIR=../../commons

$(EXPODIR)/overflowlocal7_gcc_64: *.c
	gcc $(CFLAGS) -I$(INCLUDEDIR) main.c -o $@

$(EXPODIR)/overflowlocal7_llvm_64: *.c
	clang $(CFLAGS) -I$(INCLUDEDIR) main.c -o $@

$(EXPODIR)/overflowlocal7_afl_64: *.c
	afl-clang-fast -I$(INCLUDEDIR) -O0 -fno-stack-protector main.c -o $@

$(EXPODIR)/overflowlocal7_afl_asan_64: *.c
	afl-clang-fast -I$(INCLUDEDIR) -O0 -fsanitize=address -fno-stack-protector main.c -o $@
```

LLVM/GCC built-in sanitizers

Sanitizer	Flag	Detects
ASan	-fsanitize=address	Heap/stack overflow, use-after-free, double-free
MSan	-fsanitize=memory	Uninitialized memory reads
UBSan	-fsanitize=undefined	Undefined behavior (int overflow, null deref)
TSan	-fsanitize=thread	Data races, deadlocks
LSan	-fsanitize=leak	Memory leaks

AFL++ Architecture



AddressSanitizer (ASan) USENIX ATC’12

AddressSanitizer: A Fast Address Sanity Checker

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov
Google
{kcc,bruening,glider,dvyukov}@google.com

Abstract

Memory access bugs, including buffer overflows and uses of freed heap memory, remain a serious problem for programming languages like C and C++. Many memory error detectors exist, but most of them are either slow or detect a limited set of bugs, or both.

This paper presents AddressSanitizer, a new memory

zones around stack and global objects to detect overflows and underflows. The current implementation is based on the LLVM [4] compiler infrastructure. The run-time library replaces `malloc`, `free` and related functions, creates poisoned redzones around allocated heap regions, delays the reuse of freed heap regions, and does error reporting.

https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf

ASan

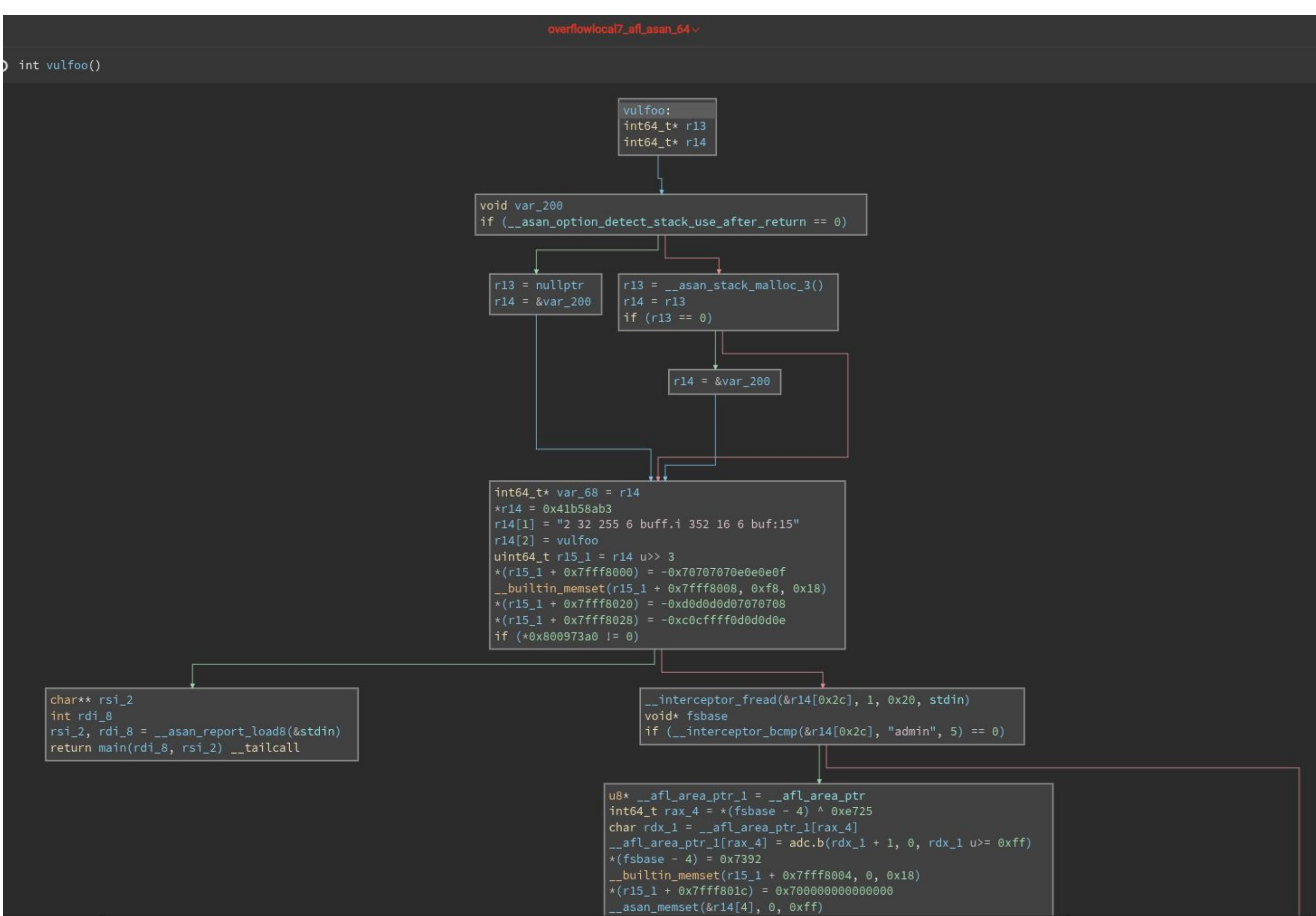
- Maps a shadow memory region tracking state of every byte in your program
- Every memory access is instrumented: is this address valid to read/write?
- Inserts red zones. Poisoned buffer zones around every allocation
- On bad access: prints full stack trace and calls abort() → SIGABRT
- ~2x slowdown — fast enough to use with AFL

ASan

Stack layout for: char buf[16]

```
RED ZONE (32 bytes)
return address
saved rbp
is_admin (4 bytes)
RED ZONE (32 bytes) ← write here → SIGABRT
buf[15]
buf[14]
...
buf[0] ← fread writes here first
RED ZONE (32 bytes)
```

vulfoo() with ASan



Fuzz command

```
afl-fuzz -i inputs -o outputs -m none -- ./target_binary
```

```
              american fuzzy lop 2.57b (overflowlocal7_afl_asan_64)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 19 sec        │  cycles done : 7       │
│   last new path : none yet (odd, check syntax!)       │  total paths : 2       │
│ last uniq crash : 0 days, 0 hrs, 0 min, 18 sec        │ uniq crashes : 1       │
│  last uniq hang : none seen yet                       │   uniq hangs : 0       │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 0.00% / 0.01%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 2 (100.00%)            │
│ stage execs : 91/192 (47.40%)       │  new edges on : 2 (100.00%)            │
│ total execs : 13.5k                 │ total crashes : 2118 (1 unique)        │
│  exec speed : 697.7/sec             │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/72, 0/70, 0/66                      │    levels : 1          │
│  byte flips : 0/9, 0/7, 0/3                         │   pending : 0          │
│ arithmetics : 0/503, 0/0, 0/0                       │  pend fav : 0          │
│  known ints : 0/51, 0/194, 0/132                    │ own finds : 0          │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 1/5760, 0/6480                        │ stability : 100.00%    │
│        trim : 10.00%/2, 0.00%                       ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 63%]
```

Crash found!

```
➜  outputs git:(master) ✗ ls ./crashes
id:000000,sig:06,src:000000,op:havoc,rep:128  README.txt
➜  outputs git:(master) ✗ cat ./crashes/id:000000,sig:06,src:000000,op:havoc,rep:128| xxd
00000000: 2c2c 2c2c 2c2c 2c2c 2c2c 2c2c 2c2c 2c2c  ,,,,,,,,,,,,,,,,
00000010: 2c2c 2c2c 2c2c 2c2c cd00 6e2c 2c2c 2c2c  ,,,,,,,,..n,,,,,
00000020: 2c2c 2c2c 2c2c 2c2c 2c2c 2c2c 2c2c 2c2c  ,,,,,,,,,,,,,,,,
00000030: 2c2c 2ccd 006e 0a00 cc13 f5              ,,,..n.....
```

id:000000,sig:06,src:000000,op:havoc,rep:128

A more complicated example

parser

```
typedef struct {
    char *name;          /* heap allocated */
    int   age;
    int   score;
    char  cmd[16];       /* fixed size */
} Config;

void set_name(Config *cfg, const char *value) {
    char buf[64];
    strcpy(buf, value);          /* no bounds check — stack overflow */
    cfg->name = strdup(buf);}

void print_config(Config *cfg) {
    printf("Name  : %s\n", cfg->name);
    printf("Age   : %d\n", cfg->age);
    printf("Score : %d\n", cfg->score);}

void set_score(Config *cfg, int score) {
    cfg->score = score;
    int normalised = 100 / score;        /* crash when score == 0 */
    printf("Normalised score: %d\n", normalised);}

void set_cmd(Config *cfg, const char *value) {
    strcpy(cfg->cmd, value);
    printf("Command set: %s\n", cfg->cmd);}
```

```
void parse_line(Config *cfg, char *line) {
    int len = strlen(line);
    if (len > 0 && line[len-1] == '\n')
        line[--len] = '\0';

    if (strncmp(line, "NAME=", 5) == 0) {
        set_name(cfg, line + 5);
    } else if (strncmp(line, "AGE=", 4) == 0) {
        cfg->age = atoi(line + 4);
    } else if (strncmp(line, "SCORE=", 6) == 0) {
        set_score(cfg, atoi(line + 6));
    } else if (strncmp(line, "CMD=", 4) == 0) {
        set_cmd(cfg, line + 4);
    } else if (strcmp(line, "PRINT") == 0) {
        print_config(cfg);
    } else {
        printf("Unknown key: %s\n", line);}}

int main(void) {
    Config cfg;
    memset(&cfg, 0, sizeof(cfg));   /* cfg.name starts as NULL */

    char line[256];
    while (fgets(line, sizeof(line), stdin)) {
        parse_line(&cfg, line);}

    free(cfg.name);
    return 0;}
```