AFL 소스코드 분석

스마트보안전공 202334843 이동하

AFL++ 소스코드 분석

제 과실로 AFL이 아닌 AFL++ 을 분석하였습니다.
모든 코드를 분석하는데에 실패하였습니다.



American Fuzzy Lop plus plus (AFL++)

Release version: 4.30c

GitHub version: 4.31a (review target)

AFL++ is a superior fork to Google's AFL

Ref : https://aflplus.plus/

$git clone https://github.com/AFLplusplus/AFLplusplus

$git checkout stable

$code .



Ref : https://github.com/AFLplusplus/AFLplusplus

$git clone https://github.com/AFLplusplus/AFLplusplus

$git checkout stable

$code .



Target :

./AFLplusplus/instrumentation

./AFLplusplus/src

./AFLplusplus/include

Ref : https://github.com/AFLplusplus/AFLplusplus





- qemu mode
- frida mode
- nyx_mode
- coresight_mode

- custom mutator
- Mopt mutator
- And more

- GCC plugin
- CLANG
- afl-cc





platform

- qemu mode
- frida mode
- nyx_mode
- coresight_mode

- custom mutator
- Mopt mutator
- And more

- GCC plugin
- CLANG
- afl-cc





- qemu mode
- frida mode
- nyx_mode
- coresight_mode

mutation
- custom mutator
- Mopt mutator
- And more

- GCC plugin
- CLANG
- afl-cc





- qemu mode
- frida mode
- nyx_mode
- coresight_mode

- custom mutator
- Mopt mutator
- And more

instrumentation
- GCC plugin
- CLANG
- afl-cc

./GNUmakefile

Build option & flag

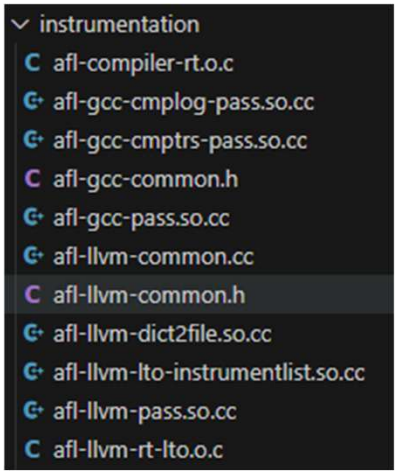
```
PROGS       = afl-fuzz afl-showmap afl-tmin afl-gotcpu afl-analyze
SH_PROGS    = afl-plot afl-cmin afl-cmin.bash afl-whatsup afl-addseeds afl-system-config afl-persistent-config afl-cc
```

생성되는 binary를 알 수 있다.

-> main함수가 존재하는 함수들이다.

---

instrumentation/afl-* (compiler,gcc,llvm)
src/afl-cc.c



./src/afl-fuzz.c

[1] initialize : afl-fuzz 실행 시점의 arg, env를 읽어와서 config 셋팅, env 셋팅, validation check

- `--version` : print VERSION

```
// c = release, a = volatile github dev, e = experimental branch
#define VERSION "++4.31a"
```

./include config.h

- `--help` : call `usage` → print manual

-a input mode 결정

- `-a` `text/ascii/txt/asc` → afl 객체의 input_mode를 1로 설정
- `-a` `bin/binary` → afl 객체의 input_mode를 2로 설정
- `-a` `def/default` → afl 객체의 input_mode 0으로 설정

-P fuzz mode 결정

- `-P` `explore/exploration` → afl 객체의 fuzz_mode를 0로 설정
- `-P` `exploit/exploitation` → afl 객체의 fuzz_mode를 1로 설정
- `-P` `<int>` → afl 객체의 switch_fuzz_mode를 <int>*1000으로 설정(그 외는 0)

-g/G minMAX 결정

- `-g/G` `<int>` → afl 객체의 min_length/max_length 결정

-Z : old_seed_selection = 1

-I <int> : afl->infoexec = <int>

arg 종류는 매우 다양하고 docs에 정리되어 있어 생략

./src/afl-fuzz.c

[1] initialize : afl-fuzz 실행 시점의 arg, env를 읽어와서 config 셋팅, env 셋팅, validation check

- `AFL_NO_COLOR` or `AFL_NO_COLOUR` 가 1이면 → print WARNING
  영향이 없다고 함(컴파일 시 설정)
- `AFL_DEBUG` 가 1이면 → afl 객체의 debug 멤버를 1로 설정
  기능은 뒤에서 서술
- `AFL_NO_FORKSRV` 가 1이면, afl->no_forkserver = 1
- `AFL_NO_CPU_RED` 가 1이면, afl->no_cpu_meter_red = 1
- `AFL_NO_ARITH` 가 1이면, afl->no_arith = 1
- `AFL_SHUFFLE_QUEUE` 가 1이면, afl->shuffle_queue = 1
- `AFL_EXPAND_HAVOC_NOW` 가 1이면, afl->expand_havoc = 1

env 종류는 매우 다양하고 docs에 정리되어 있어 생략

./src/afl-fuzz.c

[2] configuration

afl_state_t struct (./include/afl-fuzz.h) 가 fuzzer의 status를 관리한다.

./src/afl-fuzz.c 에서 afl 이라는 이름의 객체로 정의된다.

```
afl_state_t *afl = calloc(1, sizeof(afl_state_t));
```

```
typedef struct afl_state {

  /* Position of this state in the global states list */
  u32 _id;

  afl_forkserver_t fsrv;
  sharedmem_t      shm;
  sharedmem_t     *shm_fuzz;
  afl_env_vars_t   afl_env;

  char **argv;                                  /* argv if needed */
(생략)
```

./src/afl-fuzz.c

[2] configuration

<example>

```
u8 *in_dir,                       /* Input directory with test cases  */
   *out_dir,                      /* Working & output directory       */
   *tmp_dir,                      /* Temporary directory for input    */
   *sync_dir,                     /* Synchronization directory        */
   *sync_id,                      /* Fuzzer ID                        */
   *power_name,                   /* Power schedule name              */
   *use_banner,                   /* Display banner                   */
   *in_bitmap,                    /* Input bitmap                     */
   *file_extension,               /* File extension                   */
   *orig_cmdline,                 /* Original command line            */
   *infoexec;                    /* Command to execute on a new crash */
```

```
case 'i':                                          /* input dir */

        if (afl->in_dir) { FATAL("Multiple -i options not supported"); }
        if (optarg == NULL) { FATAL("Invalid -i option (got NULL)."); }
        afl->in_dir = optarg;

        if (!strcmp(afl->in_dir, "-")) { afl->in_place_resume = 1; }

        break;
```

./src/afl-fuzz.c

[2] configuration

```
  is_main_node,                   /* if this is the main node         */
  is_secondary_node,              /* if this is a secondary instance  */
  pizza_is_served,                /* pizza mode                       */
  input_mode,                     /* target wants text inputs         */
  fuzz_mode,          /* coverage/exploration or crash/exploitation mode */
  schedule,                       /* Power schedule (default: EXPLORE)*/
  havoc_max_mult,                 /* havoc multiplier                 */
  skip_deterministic,             /* Skip deterministic stages?       */
  use_splicing,                   /* Recombine input files?           */
  non_instrumented_mode,          /* Run in non-instrumented mode?    */
  score_changed,                  /* Scoring for favorites changed?   */
```

```
if (likely(afl->switch_fuzz_mode && afl->fuzz_mode == 0 &&
           !afl->non_instrumented_mode) &&
    unlikely(cur_time > (likely(afl->last_find_time) ? afl->last_find_time
                                                     : afl->start_time) +
                        afl->switch_fuzz_mode)) {

  if (afl->afl_env.afl_no_ui) {

    ACTF(
        "No new coverage found for %llu seconds, switching to exploitation "
        "strategy.",
        afl->switch_fuzz_mode / 1000);

  }

  afl->fuzz_mode = 1;

}
```

같은 로직의 반복 생략

./src/afl-fuzz.c

[3] prepare

afl->schedule : FAST/COE/EXPLOIT/LIN/QUAD/MMOPT/RARE/SEEK/EXPLORE 중 하나의 값

https://aflplus.plus/docs/power_schedules/

(다르파 결승에서 공개된 기술인 듯 하다. 2016 ACM 논문도 존재한다.)

./src/afl-fuzz.c

[3] prepare

퍼징해야 할 순서를 결정한다.

AFL flag	Power Schedule
`-p explore`	$p(i) = \frac{\alpha(i)}{\beta}$
`-p fast` (default)	$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)}}{f(i)}, M\right)$
`-p coe`	$p(i) = \begin{cases} 0 & \text{if } f(i) > \mu \\ \min\left(\frac{\alpha(i)}{\beta} \cdot 2^{s(i)}, M\right) & \text{otherwise.} \end{cases}$
`-p quad`	$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)^2}{f(i)}, M\right)$
`-p lin`	$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)}{f(i)}, M\right)$
`-p exploit` (AFL)	$p(i) = \alpha(i)$

α(i) 는 입력 i 에 대하여 얻은 performance score다

s(i) 는 입력 i 가 queue에서 선택된 횟 수다.

f(i) 는 입력 i 와 같은 실행 경로를 따르는 입력의 개수

μ: 이는 특정 실행 경로를 따르는 생성된 입력들의 평균 개수

프로그램의 다양한 실행 경로를 생성된 입력들이 얼마나 자주 탐색하는지를 평균적으로 측정한 값

β(i)>1인 상수

AFL flag	Power Schedule
`-p explore`	$p(i) = \frac{\alpha(i)}{\beta}$
`-p fast` (default)	$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)}}{f(i)}, M\right)$
`-p coe`	$p(i) = \begin{cases} 0 & \text{if } f(i) > \mu \\ \min\left(\frac{\alpha(i)}{\beta} \cdot 2^{s(i)}, M\right) & \text{otherwise.} \end{cases}$
`-p quad`	$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)^2}{f(i)}, M\right)$
`-p lin`	$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)}{f(i)}, M\right)$
`-p exploit` (AFL)	$p(i) = \alpha(i)$

Ref : Coverage-based Greybox Fuzzing as Markov Chain 논문 참고

./src/afl-fuzz.c

[3] prepare

power schedules 은 여러 seed를 효율적으로 사용하여 input을 생성하는 전략이다.

예외 핸들링으로 빠지는 경로를 높은 빈도로 접근하는 경우, 예외 처리되어 있는 seed에 많은 에너지를 쏟는 문제가 있다.

반면, 의심스럽고 흥미로운 경로를 낮은 빈도로 접근하는 경우, 유의미한 경로임에도 낮은 에너지를 쏟아 결과에 부정적인 영향을 주는 사례가 존재한다.

이를 해결하기 위한 것이 **power schedules**이다.
seed의 performance score 를 **`calculate_score`** 로 저장
일부 seed를 즐겨찾기로 등록 in **`update_bitmap_score`** func
다음에 선택 될 seed를 선택 in **`main`** func

./src/afl-fuzz.c

[3] prepare

*p(i)는 입력 i에 할당되는 에너지를 의미한다.

EXPLORE : seed의 score만을 반영한 전략
새로운 경로에 중점을 두고, 특정 케이스에 집중하지 않는다.

$$p(i) = \frac{\alpha(i)}{\beta}$$

FAST : 실행 빈도(f(i))가 낮은 경로에 높은 에너지를 부여한다.(귀한 input)
선택된 횟수(s(i))가 많을 수록 에너지가 지수적으로 늘어난다.
저빈도 경로에 집중하되, 상한 M을 통해 지나치게 집중하지 않도록 제한한다.

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)}}{f(i)}, M\right)$$

./src/afl-fuzz.c

[3] prepare

*p(i)는 입력 i에 할당되는 에너지를 의미한다.

COE :
f(i)>μ ← 빈도가 높음을 의미
빈도가 높은 경로를 더 이상 퍼징하지 않는다.
시드가 새로운 경로를 발견하며 반복적으로 선택되는 경우에 지수적으로 에너지가 늘어난다.
고빈도 경로를 무시하고 저빈도 경로에 집중하는 전략.

$$p(i) = \begin{cases} 0, & \text{if } f(i) > \mu \\ \min\left(\frac{\alpha(i)}{\beta} \cdot 2^{s(i)}, M\right), & \text{otherwise} \end{cases}$$

./src/afl-fuzz.c

[3] prepare

*p(i)는 입력 i에 할당되는 에너지를 의미한다.

QUAD :
seed가 선택된 횟 수(s(i)) 에 따라서 에너지가 지수적으로 늘어난다 ← 특정 경로에 집중한다는 의미
실행 빈도(f(i))가 높은 경로에 적은 에너지를 부여한다.
특정 경로를 집중적으로 탐색 할 때 사용되는 전략

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)^2}{f(i)}, M\right)$$

./src/afl-fuzz.c

[3] prepare

*p(i)는 입력 i에 할당되는 에너지를 의미한다.

LINEAR :
seed의 선택 빈도(s(i))에 비례하여 리니어하게 에너지가 늘어난다.
실행 빈도(f(i))가 높은 경로에 낮은 에너지를 부여한다.
균형잡힌 전략

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)}{f(i)}, M\right)$$

./src/afl-fuzz.c

[3] prepare

*p(i)는 입력 i에 할당되는 에너지를 의미한다.

EXPLOIT :
모든 seed에 동일한 에너지를 부여한다.
결국 seed의 score(실행 시간, 커버리지 등)에 따라 계산된다.

$$p(i) = \alpha(i)$$

./src/afl-fuzz.c

[3] prepare

```
enum {

  /* 00 */ EXPLORE, /* AFL default, Exploration-based constant schedule */
  /* 01 */ MMOPT,   /* Modified MOPT schedule                           */
  /* 02 */ EXPLOIT, /* AFL's exploitation-based const.  */
  /* 03 */ FAST,    /* Exponential schedule             */
  /* 04 */ COE,     /* Cut-Off Exponential schedule     */
  /* 05 */ LIN,     /* Linear schedul                   */
  /* 06 */ QUAD,    /* Quadratic sch                    */
  /* 07 */ RARE,    /* Rare edges                       */
  /* 08 */ SEEK,    /* EXPLORE that

  POWER_SCHEDULES_NUM

};
```

```
if (afl->schedule >= FAST && afl->schedule <= RARE) {
  afl->n_fuzz = ck_alloc(N_FUZZ_SIZE * sizeof(u32));
}
```

```
<in afl-fuzz.h>
#define N_FUZZ_SIZE (1 << 21)
  u32 *n_fuzz;
```

`afl->n_fuzz` 스케줄링을 위한 공간의 포인터, 2MB*4BYTE=8MB 할당

./src/afl-fuzz.c

[3] prepare

env load 관련 반복 로직

mode 관련된 로직은 생략

./src/afl-fuzz.c

[4] start

```
<afl-fuzz.h>
#define AFL_BUF_PARAM(name) ((void **)&afl->name##_buf)
```

```
  OKF("Generating fuzz data with a length of min=%u max=%u", afl->min_length,
      afl->max_length);
  u32 min_alloc = MAX(64U, afl->min_length);
  afl_realloc(AFL_BUF_PARAM(in_scratch), min_alloc);
  afl_realloc(AFL_BUF_PARAM(in), min_alloc);
  afl_realloc(AFL_BUF_PARAM(out_scratch), min_alloc);
  afl_realloc(AFL_BUF_PARAM(out), min_alloc);
  afl_realloc(AFL_BUF_PARAM(eff), min_alloc);
  afl_realloc(AFL_BUF_PARAM(ex), min_alloc);
```

`min_alloc` 크기로 `afl->in_scratch/afl->in/afl->out_scratch/afl->out/afl->eff/afl->ex` realloc

./src/afl-fuzz.c

[4] start

```
static inline void *afl_realloc(void **buf, size_t size_needed) {

  struct afl_alloc_buf *new_buf = NULL;

  size_t current_size = 0;
  size_t next_size = 0;

  if (likely(*buf)) {

    /* the size is always stored at buf - 1*size_t */
    new_buf = (struct afl_alloc_buf *)afl_alloc_bufptr(*buf);
    current_size = new_buf->complete_size;

  }

  size_needed += AFL_ALLOC_SIZE_OFFSET;
```

./include/alloc_inl.h 파일이며, 일반적인 realloc과 유사(2의 지수 승을 단위로 커진다는 차이점이 있음)

./src/afl-fuzz.c

[4] start

```
typedef struct afl_state { // <- afl obj
  /* Position of this state in the global states list */
  u32 _id;
  afl_forkserver_t fsrv;
```

```
<./include/forkserver.h>
typedef struct afl_forkserver {

  /* a program that includes afl-forkserver needs to define these */

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

  s32 fsrv_pid,                       /* PID of the fork server           */
      child_pid,                      /* PID of the fuzzed program        */
      child_status,                   /* waitpid result for the child     */
      out_dir_fd;                     /* FD of the lock file              */
```

forkserver 객체에 대해서 afl 객체와 동일하게 설정

```
  afl->fsrv.use_fauxsrv = afl->non_instrumented_mode == 1 || afl->no_forkserver;
  afl->fsrv.max_length = afl->max_length;
```

./src/afl-fuzz.c

[4] start

```
check_crash_handling();
check_cpu_governor(afl);
```

다음 슬라이드에서 각각의 함수 분석

./src/afl-fuzz.c

[4] start

check_crash_handling

./src/afl-fuzz-init.c

```c
s32 fd = open("/proc/sys/kernel/core_pattern", O_RDONLY);
u8  fchar;

if (fd < 0) { return; }

ACTF("Checking core_pattern...");

if (read(fd, &fchar, 1) == 1 && fchar == '|') {

  SAYF("\n" cLRD "[-] " cRST
       "Your system is configured to send core dump notifications to an\n"
       "    external utility. This will cause issues: there will be an "
       "extended delay\n"
       "    between stumbling upon a crash and having this information "
       "relayed to the\n"
       "    fuzzer via the standard waitpid() API.\n"
       "    If you're just experimenting, set "
       "'AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1'.\n\n"

       "    To avoid having crashes misinterpreted as timeouts, please \n"
       "    temporarily modify /proc/sys/kernel/core_pattern, like so:\n\n"

       "    echo core | sudo tee /proc/sys/kernel/core_pattern\n");

  if (!getenv("AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES")) {

    FATAL("Pipe at the beginning of 'core_pattern'");

  }
```

./src/afl-fuzz.c

[4] start

`/proc/sys/kernel/core_pattern` 에 접근하여 read-only 로 1바이트 읽어온다.

읽어온 바이트가 | 인지 확인한다.

core dump를 외부 유틸리티로 전달하기 때문에, crash 발견이 늦어질 수 있다고 경고를 발생한다.

```
whoami@whoami:~/AFLplusplus$ ls /proc/sys/kernel/core_pattern
/proc/sys/kernel/core_pattern
whoami@whoami:~/AFLplusplus$ cat /proc/sys/kernel/core_pattern
|/usr/share/apport/apport -p%p -s%s -c%c -d%d -P%P -u%u -g%g -- %E
whoami@whoami:~/AFLplusplus$
```

파이프 라인이 아닌, core라면 core dump를 파일로 저장하게 된다.

이를 기대하는 코드이며, `AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES` env 설정으로 bypass할 수 있다.

./src/afl-fuzz.c

[4] start

check_cpu_governor

linux와 apple 중 linux 기준으로 분석하였다.

afl->afl_env.afl_skip_cpufreq 가 1 이면 아래 로직은 생략된다.

afl->cpu_aff 가 1 이상의 값으로 지정되었으면, 해당 값을 코어 넘버로 받아서 조사, 그 외에는 0번 코어를 조사한다.

`/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor"` 를 조사하거나, 0 자리에 afl->cpu_aff 가 들어간다.

```
whoami@whoami:~$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
schedutil
whoami@whoami:~$
```

조사 한 값 → `&tmp` → `fd(f)`

cpu freq의 min, max를 조사하고, 해당 값이 유효한지 체크한다.

./src/afl-fuzz.c

[4] start

ENV check (함수는 아님)

apple 환경이라면 DYLD_NO_PIE 환경변수가 1로 설정된다.

LD_PRELOAD 환경 변수를 로드하여 셋팅이 있는 지 조회한다. 있다면 WARNING 발생 → AFL_PRELOAD 말고 LD_PRELOAD를 쓴 것이 의도한 것인지 묻는다.

`afl->afl_env.afl_preload` 조사하여 AFL_PRELOAD 가 활성화 되었는 지 조사한다.

qemu_mode 또는 frida_mode 를 사용한 경우, 해당 mode에 대한 로직이 들어간다.

이는 생략하였다.

어떠한 모드도 선택되지 않았다면, 아래 2개 로직이 실행된다.

LD_PRELOAD 에다가 AFL_PRELOAD 값을 써준다.

DYLD_INSERT_LIBRARIES 에다가 AFL_PRELOAD 값을 써준다.

AFL_PRELOAD > LD_PRELOAD

LD_PRELOAD : 동적 lib 우선 지정 env

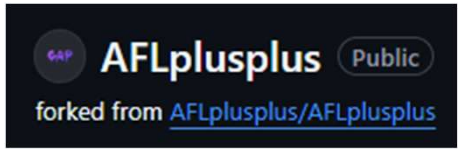
./src/afl-fuzz.c

[4] start



AFLplusplus contributer?

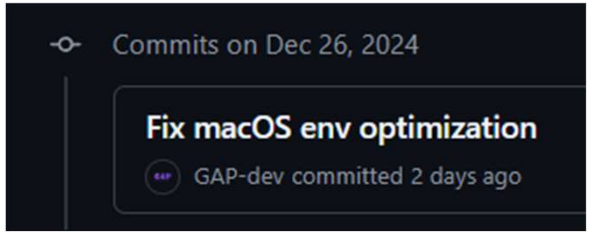
[1]



fork

---

[2]



$git checkout dev

$git pull

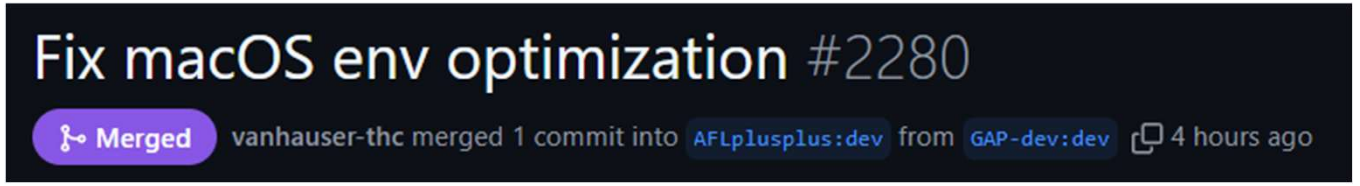
$git add .

$git commit –m "Fix macOS env optimization"

$git push

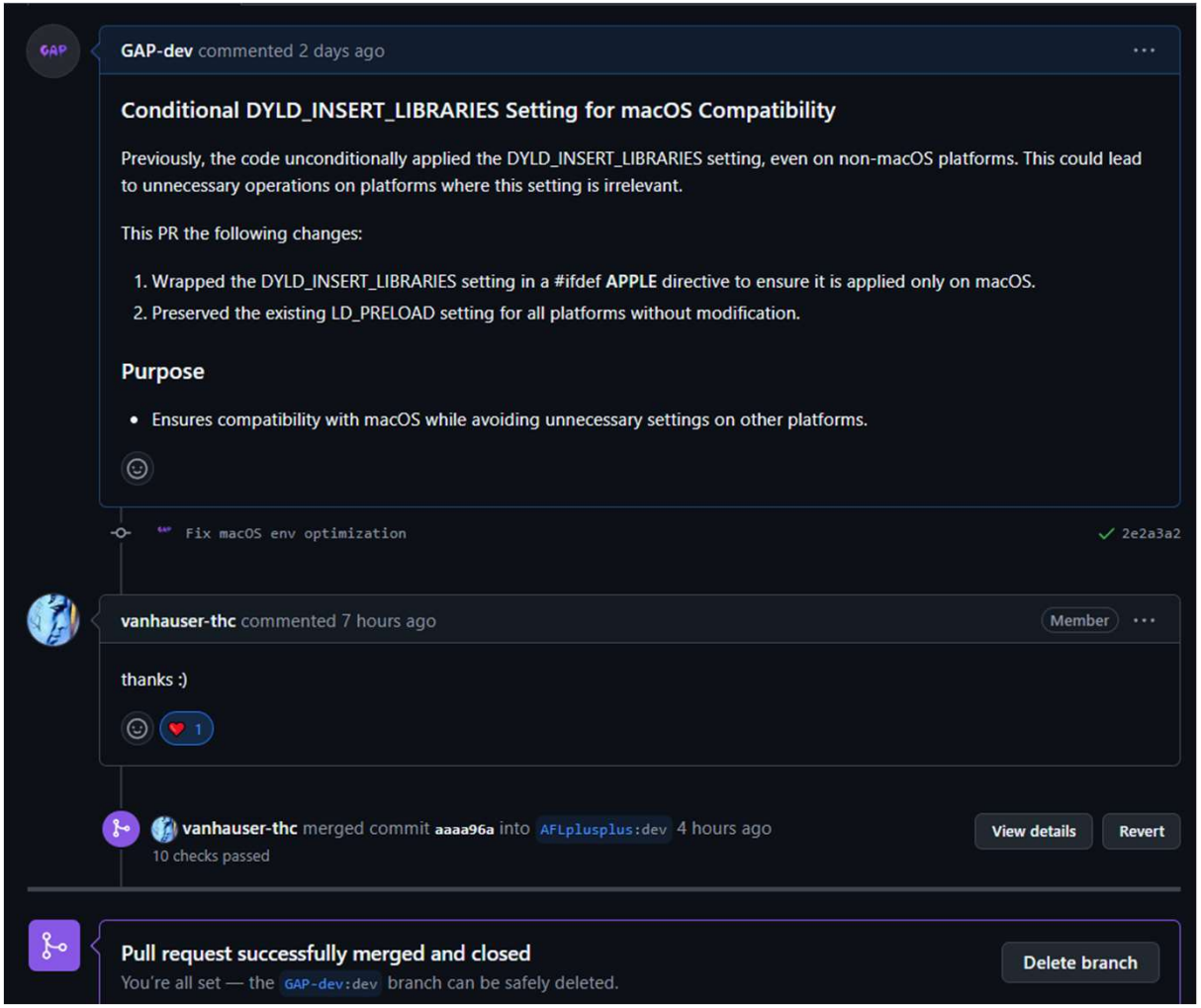
---

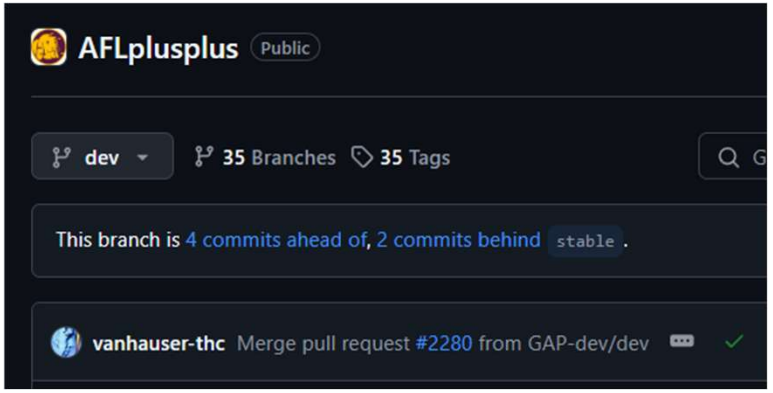
[3]



pull request

AFLplusplus contributer?





./src/afl-fuzz.c

[4] start

`AFL_LD_PRELOAD` 와 같은 잘못 된 환경변수를 사용한 경우 FATAL을 발생시키며, bad value가 전달 된 경우 FATAL을 발생시킨다.

`save_cmdline` 이 호출되어 argc, argv가 afl 객체에 저장된다. (./src/afl-init.c 에 존재)

`check_if_tty` 이 호출되어 `afl->afl_env.afl_no_ui` 이 활성화 된 경우, ui가 없는 환경으로 간주하여 `afl->not_on_tty` 이 1로 설정된다. (./src/afl-init.c 에 존재)

```
if (ioctl(1, TIOCGWINSZ, &ws)) {
```

ioctl로 stdout의 `TIOCGWINSZ` 을 요청하여 ws에 저장한다.

`TIOCGWINSZ` 은 창의 크기를 의미한다고 하며, errno == `ENOTTY` 를 조사한다. `ENOTTY` 는 no tty라는 뜻으로 stdout이 terminal 로 연결되지 않음을 의미한다고 한다.(file,pipe,etc.)

동일하게 `afl->not_on_tty` 이 1로 설정된다.

그러나 `afl->afl_env.afl_force_ui` 가 1인 경우에는 반드시 `afl->not_on_tty` 가 0 이다

./src/afl-fuzz.c

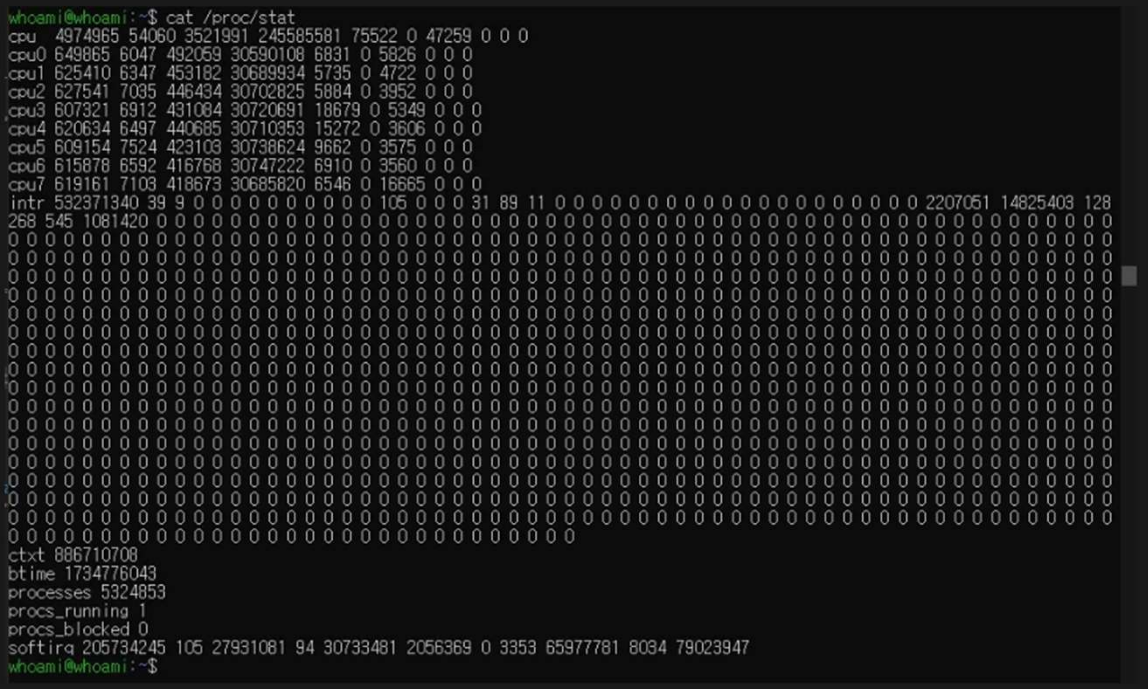
[4] start

다음으로는 `get_core_count` 가 호출된다.

```
#if defined(__APPLE__) || defined(__FreeBSD__) || defined(__OpenBSD__) || \
    defined(__DragonFly__)
```

환경에 따라 코드도 다양한데, 리눅스 기준으로는 아래와 같은 방법을 사용한다.

```
FILE *f = fopen("/proc/stat", "r");
u8    tmp[1024];
```



./src/afl-fuzz.c

[4] start

```
while (fgets(tmp, sizeof(tmp), f))
  if (!strncmp(tmp, "cpu", 3) && isdigit(tmp[3])) ++afl->cpu_core_count;
```

`cpu` 라는 3글자가 몇 번 등장하였는지 카운팅하여, 이를 `afl->cpu_core_count` 에 저장한다.

예외적인 시스템이 존재할 수 있겠다는 생각에 linux /proc/stat 을 조사해보았다. 🙏 **Linux Howtos: System -> /proc/stat explained**

`afl->cpu_core_count` 가 0인 경우 → 처리 불가 경고

`get_runnable_processes` 호출하여 /proc/stat 에서 blocked/running 을 ~~각각~~ 카운팅한다.
`cur_runnable` 에 저장된다.

./src/afl-fuzz.c

[4] start

```
FILE *f = fopen("/proc/stat", "r");
u8    tmp[1024];
u32   val = 0;

if (!f) { return 0; }

while (fgets(tmp, sizeof(tmp), f)) {

  if (!strncmp(tmp, "procs_running ", 14) ||
      !strncmp(tmp, "procs_blocked ", 14)) {

    val += atoi(tmp + 14);

  }

}

fclose(f);
```

./src/afl-fuzz.c

[4] start

apple, freebsd, openbsd, dragonfly에서는 자기자신을 더해준다.(+1)

```
#if defined(__APPLE__) || defined(__FreeBSD__) || defined(__OpenBSD__) || \
    defined(__DragonFly__)
    ++cur_runnable;
#endif
```

./src/afl-fuzz.c

[4] start

- CPU 코어 수 (`afl->cpu_core_count`).
- 실행 가능한 태스크 수 (`cur_runnable`).
- CPU 활용도 (`cur_runnable * 100.0 / afl->cpu_core_count`).
- CPU 코어 수가 1개일 때와 여러 개일 때의 출력 문법을 구분(`core` vs `cores`).

```
OKF("You have %d CPU core%s and %u runnable tasks (utilization: %0.0f%%).",
    afl->cpu_core_count, afl->cpu_core_count > 1 ? "s" : "", cur_runnable,
    cur_runnable * 100.0 / afl->cpu_core_count);
```

./src/afl-fuzz.c

[4] start

실행 가능한 태스크 수가 CPU 코어 수의 1.5배를 초과하는 경우, 시스템 부하가 높다고 판단

실행 가능한 태스크 수가 CPU 코어 수 이하인 경우(약간의 여유가 있는 경우), 병렬 작업을 시도하라고 권장

```
OKF("You have %d CPU core%s and %u runnable tasks (utilization: %0.0f%%).",
        afl->cpu_core_count, afl->cpu_core_count > 1 ? "s" : "", cur_runnable,
        cur_runnable * 100.0 / afl->cpu_core_count);

    if (afl->cpu_core_count > 1) {

      if (cur_runnable > afl->cpu_core_count * 1.5) {

        WARNF("System under apparent load, performance may be spotty.");

      } else if ((s64)cur_runnable + 1 <= (s64)afl->cpu_core_count) {

        OKF("Try parallel jobs - see "
            "%s/fuzzing_in_depth.md#c-using-multiple-cores",
            doc_path);
      }
    }
```

./src/afl-fuzz.c

[4] start – set exit

`ex_exit` 을 atexit 함수로 등록

```
atexit(at_exit);
```

atexit → exit 시 실행될 함수를 설정 (stdlib.h)

```
static void at_exit() {

  s32   i, pid1 = 0, pid2 = 0, pgrp = -1;
  char *list[4] = {SHM_ENV_VAR, SHM_FUZZ_ENV_VAR, CMPLOG_SHM_ENV_VAR, NULL};
  char *ptr;

  ptr = getenv("__AFL_TARGET_PID2");
  if (ptr && *ptr && (pid2 = atoi(ptr)) > 0) {

    pgrp = getpgid(pid2);
    if (pgrp > 0) { killpg(pgrp, SIGTERM); }
    kill(pid2, SIGTERM);

  }

  ptr = getenv("__AFL_TARGET_PID1");
  if (ptr && *ptr && (pid1 = atoi(ptr)) > 0) {

    pgrp = getpgid(pid1);
    if (pgrp > 0) { killpg(pgrp, SIGTERM); }
    kill(pid1, SIGTERM);

  }
```

./src/afl-fuzz.c

[4] start – set exit

`ex_exit` 을 atexit 함수로 등록

```
atexit(at_exit);
```

atexit → exit 시 실행될 함수를 설정 (stdlib.h)

```
static void at_exit() {

  s32   i, pid1 = 0, pid2 = 0, pgrp = -1;
  char *list[4] = {SHM_ENV_VAR, SHM_FUZZ_ENV_VAR, CMPLOG_SHM_ENV_VAR, NULL};
  char *ptr;

  ptr = getenv("__AFL_TARGET_PID2");
  if (ptr && *ptr && (pid2 = atoi(ptr)) > 0) {

    pgrp = getpgid(pid2);
    if (pgrp > 0) { killpg(pgrp, SIGTERM); }
    kill(pid2, SIGTERM);

  }

  ptr = getenv("__AFL_TARGET_PID1");
  if (ptr && *ptr && (pid1 = atoi(ptr)) > 0) {

    pgrp = getpgid(pid1);
    if (pgrp > 0) { killpg(pgrp, SIGTERM); }
    kill(pid1, SIGTERM);

  }
```

./src/afl-fuzz.c

[4] start – set exit

`__AFL_TARGET_PID2` 는 ./src/afl-forkserver.c 에서 setenv로 저장한 env다.

```
/* PARENT PROCESS */

char pid_buf[16];
sprintf(pid_buf, "%d", fsrv->fsrv_pid);
if (fsrv->cmplog_binary)
  setenv("__AFL_TARGET_PID2", pid_buf, 1);
else
  setenv("__AFL_TARGET_PID1", pid_buf, 1);
```

./src/afl-fuzz.c

[4] start – set exit

`fsrv->fsrv_pid` 를 의미하고 fsrv는 ./include/forkserver.h 에서 정의된 구조체이다.

```
typedef struct afl_forkserver {

  /* a program that includes afl-forkserver needs to define these */

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

  s32 fsrv_pid,                       /* PID of the fork server           */
      child_pid,                      /* PID of the fuzzed program        */
      child_status,                   /* waitpid result for the child     */
      out_dir_fd;                     /* FD of the lock file              */

  s32 out_fd,                         /* Persistent fd for fsrv->out_file */
      dev_urandom_fd,                 /* Persistent fd for /dev/urandom   */
```

./src/afl-fuzz.c

[4] start – set exit

PID of the fork server 라는 것을 주석을 통해 알 수 있다.

fork server의 pid를 읽어오고 NULL이 아닌지, 비었는지, 양수 인 지 검사한다.

pid(fork server)의 process group id 를 읽어와서 해당 group의 모든 process에 SIGTERM을 보낸다.
(종료)

pid(fork server)의 process 도 kill한다.

./src/afl-fuzz.c

[4] start – set exit

```
if (fsrv->cmplog_binary)
  setenv("__AFL_TARGET_PID2", pid_buf, 1);
else
  setenv("__AFL_TARGET_PID1", pid_buf, 1);
```

`fsrv->cmplog_binary` 활성화 여부에 따라서 `__AFL_TARGET_PID1` / `__AFL_TARGET_PID2` 사용을 결정한다.

이는 AFL++ 4.00 패치에서 수정 및 도큐먼트가 공개된 커밋으로, **docs/important_changes.md** AFLplusplus/AFLplusplus 에서 설명한다.

CMPLOG는 AFL에서 `strcmp`, `memcmp` 와 같은 비교 연산을 추적하고 뮤테이션을 효과적으로 하는 기능이다. 해당 기능이 활성화 되었다면 `fsrv->cmplog_binary` 가 활성화 되는 구조이다.

`at_exit` 에서는 `__AFL_TARGET_PID1` / `__AFL_TARGET_PID2` 모두 killgp,kill 한다.

./src/afl-fuzz.c

[4] start – set exit

```
<./include/config.h>
/* CPU Affinity lockfile env var */
  #define CPU_AFFINITY_ENV_VAR "__AFL_LOCKFILE"
```

```
  ptr = getenv(CPU_AFFINITY_ENV_VAR);
  if (ptr && *ptr) unlink(ptr);
```

`CPU_AFFINITY_ENV_VAR` 는 `__AFL_LOCKFILE` env를 의미하고, `CPU_AFFINITY_ENV_VAR` 는 ./src/afl-fuzz-init.c 에서 설정된다.

뒤에서 CPU_AFFINITY_ENV_VAR 가 무엇인지 설명한다

다시 `at_exit` 으로 돌아오면, 이러한 .affinity_lock 경로를 읽고 이를 unlink한다.

unlink 는 파일 시스템에서 pathname 이름을 가진 파일을 지운다.(unistd.h 함수)

./src/afl-fuzz.c

[4] start – what is CPU_AFFINITY_ENV

`CPU_AFFINITY_ENV_VAR` 는 `__AFL_LOCKFILE` env를 의미하고, `CPU_AFFINITY_ENV_VAR` 는 ./src/afl-fuzz-init.c 에서 설정된다.

```
<./src/afl-fuzz-init.c>
  if (afl->sync_id) {

    s32 lockfd, first = 1;

    snprintf(lockfile, sizeof(lockfile), "%s/.affinity_lock", afl->sync_dir);
    setenv(CPU_AFFINITY_ENV_VAR, lockfile, 1);

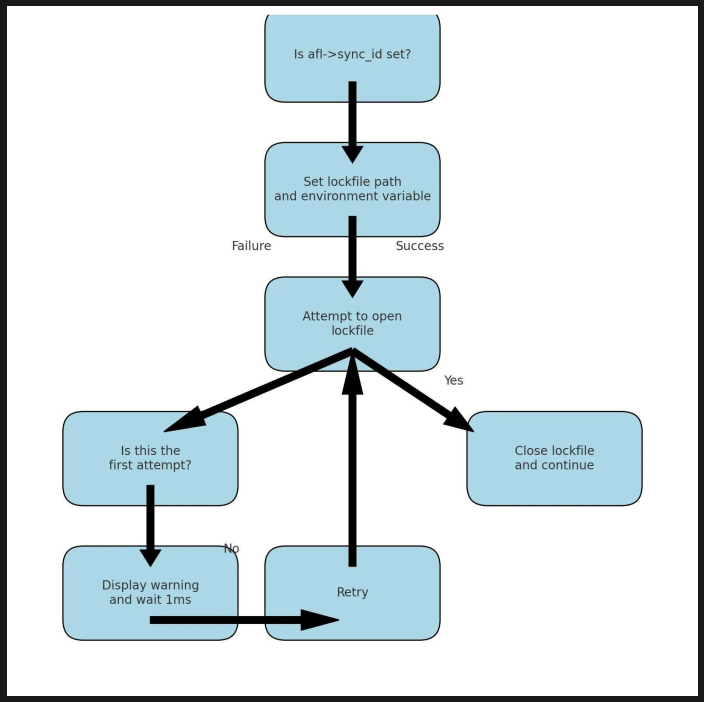
<중략>
lockfd = open(lockfile, O_RDWR | O_CREAT | O_EXCL, DEFAULT_PERMISSION)
```

`sync_id` 활성화 된 경우, `afl->sync_dir` 에 들어있는 경로에 .affinity_lock파일을 RDWR로 생성하며 재 생성을 방지한다.(race condition 방지로 보임)

해당 경로를 `CPU_AFFINITY_ENV_VAR` 에 저장한다.

./src/afl-fuzz.c

[4] start – what is CPU_AFFINITY_ENV



뒤에서 다시 정정해서 그림, 설명함

```
<./include/afl-fuzz.h>
*sync_dir,                          /* Synchronization directory        */
```

```
<./src/afl-fuzz-init.c>
if (afl->sync_id) {

   s32 lockfd, first = 1;

   snprintf(lockfile, sizeof(lockfile), "%s/.affinity_lock", afl->sync_dir);
   setenv(CPU_AFFINITY_ENV_VAR, lockfile, 1);

   do {

     if ((lockfd = open(lockfile, O_RDWR | O_CREAT | O_EXCL,
                        DEFAULT_PERMISSION)) < 0) {

       if (first) {

         WARNF("CPU affinity lock file present, waiting ...");
         first = 0;

       }

       usleep(1000);

     }

   } while (lockfd < 0);

   close(lockfd);

 }
```

./src/afl-fuzz.c

[4] start – what is CPU_AFFINITY_ENV

`sync_dir` 로 지정 된 디렉토리에 .affinity_lock 파일을 생성(접근) 시도한다.

이미 다른 프로세스가 디렉토리를 사용중이라면 .affinity_lock가 존재하며 open되어 있기에 접근 실패한다. 이를 성공할 때 까지 대기한다. (atomic하게 디렉토리를 이용하기 위함으로 보임)

./src/afl-fuzz.c

[4] start

```
  char *list[4] = {SHM_ENV_VAR, SHM_FUZZ_ENV_VAR, CMPLOG_SHM_ENV_VAR, NULL};
```

```
i = 0;
  while (list[i] != NULL) {
    ptr = getenv(list[i]);
    if (ptr && *ptr) {
#ifdef USEMMAP
      shm_unlink(ptr);
#else
      shmctl(atoi(ptr), IPC_RMID, NULL);
#endif
    }
    i++;
  }
```

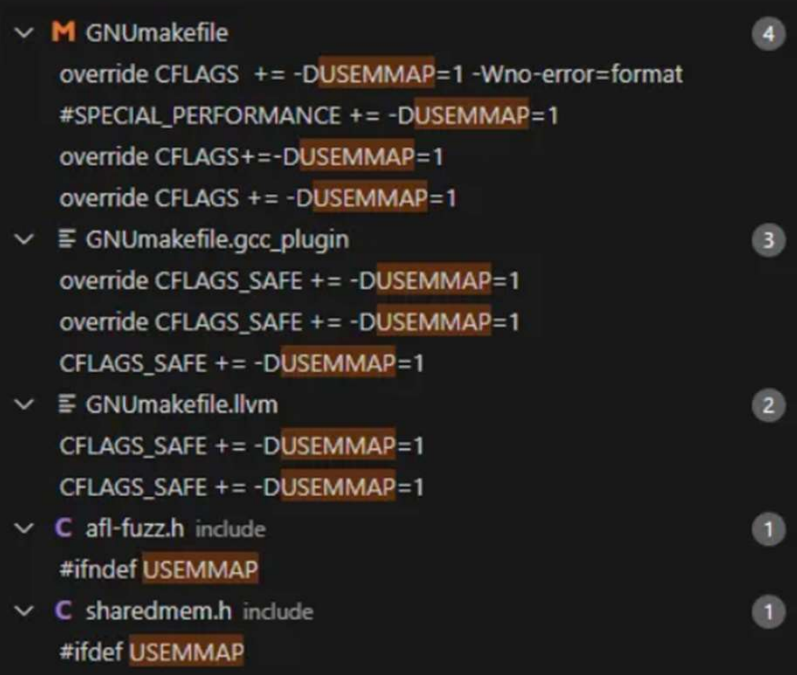
list 를 순회하면서 NULL이 아닐 때 까지 env를 검사한다.(3개 검사)

각 env가 유효하다면, 공유 메모리를 정리한다.

./src/afl-fuzz.c

[4] start

`USEMMAP` 의 컴파일 타임 정의에 따라서 코드는 다르게 적용되는데, 메모리 해제 방식을 결정한다.
(할당/관리도 다를 것으로 예상)



`USEMMAP` 가 정의된 경우, `shm_unlink(ptr);` 를 호출하여 공유 메모리 객체의 이름을 기반으로 삭제

`USEMMAP` 가 정의되지 않은 경우, `shmctl(atoi(ptr), IPC_RMID, NULL);` 를 호출하여 공유 메모리 세그먼트를 ID(`atoi(ptr)`) 기반으로 삭제합니다.

./src/afl-fuzz.c

[4] start

```
int kill_signal = SIGKILL;
/* AFL_KILL_SIGNAL should already be a valid int at this point */
if ((ptr = getenv("AFL_KILL_SIGNAL"))) { kill_signal = atoi(ptr); }
if (pid1 > 0) {
  pgrp = getpgid(pid1);
  if (pgrp > 0) { killpg(pgrp, kill_signal); }
  kill(pid1, kill_signal);
}
if (pid2 > 0) {
  pgrp = getpgid(pid1);
  if (pgrp > 0) { killpg(pgrp, kill_signal); }
  kill(pid2, kill_signal);
}
```

이전 sigterm

sigterm으로 kill되지 않은 경우, sigkill을 전달하여 강제 종료한다.

//at_exit 끝

./src/afl-fuzz.c

[4] start - setup_dirs_fds

```
<afl->out_dir>/
|
├── queue/
|   ├── .state/
|   |   ├── deterministic_done/
|   |   ├── auto_extras/
|   |   ├── redundant_edges/
|   |   └── variable_behavior/
├── crashes/
├── hangs/
├── plot_data
├── [plot_det_data] (INTROSPECTION이 활성화된 경우)
└── .synced/ (afl->sync_id가 설정된 경우)
```

`afl->out_dir` 디렉터리 기준으로 위와 같은 구조로 디렉터리가 생성된다.

이미 존재 할 경우 `handle_existing_out_dir` 가 호출된다.

./src/afl-fuzz.c

[4] start - setup_dirs_fds

```
See if the output directory is locked. If yes, bail out. If not,
     create a lock that will persist for the lifetime of the process
     (this requires leaving the descriptor open)
```

`afl->out_dir` 을 열고, file descriptor 를 `afl->fsrv.out_dir_fd` 에 저장한다.

open 실패 시 PFATAL 처리되며, 종료된다.

FATAL vs PFATAL : PFATAL 은 FATAL과 유사하지만 라이브러리나 시스템 콜에서 error가 발생했을 때 사용한다고 한다.(더 알아볼 내용)

./src/afl-fuzz.c

[4] start - setup_dirs_fds

```
#ifndef __sun
  if (flock(afl->fsrv.out_dir_fd, LOCK_EX | LOCK_NB) && errno == EWOULDBLOCK) {
    SAYF("\n" cLRD "[-] " cRST
         "Looks like the job output directory is being actively used by "
         "another\n"
         "    instance of afl-fuzz. You will need to choose a different %s\n"
         "    or stop the other process first.\n",
         afl->sync_id ? "fuzzer ID" : "output location");
    FATAL("Directory '%s' is in use", afl->out_dir);
  }
#endif                                                    /* !__sun */
  f = fopen(fn, "r");
  if (f) {
    u64 start_time2, last_update;
    if (fscanf(f,
               "start_time     : %llu\n"
               "last_update    : %llu\n",
               &start_time2, &last_update) != 2) {
      FATAL("Malformed data in '%s'", fn);
    }
    fclose(f);
```

./src/afl-fuzz.c

[4] start - setup_dirs_fds

`afl->fsrv.out_dir_fd` 을 flock으로 잠그기를 시도한다. LOCK_EX | LOCK_NB 플래그를 통해 잠그는데, `LOCK_EX` 는 배타적 잠금을 의미하며 다른 프로세스가 이 파일에 접근하지 못하게 한다. `LOCK_NB` 는 non-blocking mode로, 잠금이 안되는 경우,함수가 대기하지 않고 즉시 반환하게 된다.

flock은 실패 시 -1을 반환한다.

errno == `EWOULDBLOCK` 은 이미 다른 프로세스가 flock을 수행했다는 것을 의미한다.

./src/afl-fuzz.c

[4] start - setup_dirs_fds

#ifndef __sun 는 Solaris os 가 아닐 때를 의미한다. ifdef가 아닌, ifndef이다.

why?

Solaris 운영 체제는 파일 잠금을 지원하지만, 전통적으로 `flock`이 아닌 `fcntl` 기반의 잠금 방식을 사용한다.

`flock` :

- 주로 Linux 및 BSD 계열에서 사용
- 비교적 간단하게 파일 잠금을 구현

`fcntl` :

- Solaris에서 기본적으로 권장되는 파일 잠금 방식
- 더 세밀한 잠금 제어를 제공(예: 파일의 특정 부분만 잠금)

./src/afl-fuzz.c

[4] start - setup_dirs_fds

다른 프로세스가 out_dir를 잡고 있지 않다면, resume 절차로 진입하게 된다.

```
if (!afl->in_place_resume && afl->autoresume) {
  OKF("Detected prior run with AFL_AUTORESUME set. Resuming.");
  afl->in_place_resume = 1;
}
```

`afl->in_place_resume` 기본 값은 0이며, 기존 세션을 이어받을 지를 결정하는 플래그다.

(!afl->in_place_resume : 현재 세션이 재개 모드가 아닌 경우)

`afl->autoresume` 는 env 중 AFL_AUTORESUME=1 일 때 1의 값을 가진다.

이전 세션의 데이터가 있고 환경변수를 통해, resume을 희망할 때 in_place_resume 이 1로 트리거되며 진행된다.

./src/afl-fuzz.c

[4] start - setup_dirs_fds

```
if (!afl->in_place_resume && last_update > start_time2 &&
        last_update - start_time2 > OUTPUT_GRACE * 60) {
```

ast_update > start_time2 : out_dir의 마지막 수정 시점이 AFL 시작 시점보다 이전인 경우

(이번 실행에서 생긴 폴더는 아니라는 것)

last_update - start_time2 > OUTPUT_GRACE * 60 : 디렉터리에 기록된 데이터가 설정된 `OUTPUT_GRACE` (분) 이상의 작업 결과를 포함하는 경우

./src/afl-fuzz.c

[4] start - setup_dirs_fds

```
<./include/config.h>
#define OUTPUT_GRACE 25
```

이 때, last_update 와 start_time2 는 아래 코드에서 읽어온다.

```
u8       *fn = alloc_printf("%s/fuzzer_stats", afl->out_dir);
<중략>
f = fopen(fn, "r");
<중략>
fscanf(f,
            "start_time     : %llu\n"
            "last_update    : %llu\n",
            &start_time2, &last_update)
```

즉 해당 파트는, 사용자가 재개 모드로 실행하지 않았는데, 기존 데이터가 남아있는 경우에 기존 데이터가 덮어 쓰여지지 않도록 하기 위한 검증 파트이다.

경고 메시지를 통해, 이전 데이터를 이어서 사용하려면 `-i -` 옵션 또는 `AFL_AUTORESUME=1` 환경 변수를 설정하도록 권장한다.

./src/afl-fuzz.c

[4] start - setup_dirs_fds [end]

```
ck_free(fn);
```

리소스 해제하면서 마무리.

./src/afl-fuzz.c

[4] start - resume

```
if (afl->in_place_resume) {

  u8 *orig_q = alloc_printf("%s/queue", afl->out_dir);

  afl->in_dir = alloc_printf("%s/_resume", afl->out_dir);

  rename(orig_q, afl->in_dir);                    /* Ignore errors */

  OKF("Output directory exists, will attempt session resume.");

  ck_free(orig_q);

} else {

  OKF("Output directory exists but deemed OK to reuse.");

}
```

본격적으로 기존 fuzz state를 afl 객체로 읽어들인다.

./src/afl-fuzz.c

[4] start - resume

```
if (!afl->in_place_resume) {
  fn = alloc_printf("%s/.synced", afl->out_dir);
  if (delete_files(fn, NULL)) { goto dir_cleanup_failed; }
  ck_free(fn);
}
```

afl->out_dir/.synced/.../id:* 파일이 존재하면 제거한다.

./src/afl-fuzz.c

[4] start - resume

```
fn = alloc_printf("%s/queue/.state/deterministic_done", afl->out_dir);
  if (delete_files(fn, case_prefix)) { goto dir_cleanup_failed; }
  ck_free(fn);

  fn = alloc_printf("%s/queue/.state/auto_extras", afl->out_dir);
  if (delete_files(fn, "auto_")) { goto dir_cleanup_failed; }
  ck_free(fn);

  fn = alloc_printf("%s/queue/.state/redundant_edges", afl->out_dir);
  if (delete_files(fn, case_prefix)) { goto dir_cleanup_failed; }
  ck_free(fn);

  fn = alloc_printf("%s/queue/.state/variable_behavior", afl->out_dir);
  if (delete_files(fn, case_prefix)) { goto dir_cleanup_failed; }
  ck_free(fn);
```

afl->out_dir/queue/.state/ 하위의 서브 디렉터리를 제거한다.

./src/afl-fuzz.c

[4] start - resume

```
fn = alloc_printf("%s/queue/.state", afl->out_dir);
if (rmdir(fn) && errno != ENOENT) { goto dir_cleanup_failed; }
ck_free(fn);
```

비어있는 afl->out_dir/queue/.state 디렉터리 제거

./src/afl-fuzz.c

[4] start - resume

```
fn = alloc_printf("%s/queue", afl->out_dir);
if (delete_files(fn, case_prefix)) { goto dir_cleanup_failed; }
ck_free(fn);
```

```
u8 *const case_prefix = afl->afl_env.afl_sha1_filenames ? "" : CASE_PREFIX;
```

```
#ifndef SIMPLE_FILES
  #define CASE_PREFIX "id:"
#else
  #define CASE_PREFIX "id_"
#endif
```

queue 안에 case_prefix에 맞는 파일 삭제. (이는 직접 실습을 통해 확인해보았다.)

./src/afl-fuzz.c

[4] start - resume

`SIMPLE_FILES` 은 AFL 1.45b 버전에서 추가된 기능으로, file name을 매우 단순하게 지정하는 flag라고 한다. (1.84b 에서 패치)

```
### Version 1.45b:

  - Added support for SIMPLE_FILES in config.h for folks who don't like
    descriptive file names. Generates very simple names without colons,
    commas, plus signs, dashes, etc.

    ------------------


### Version 1.84b:

  - Made SIMPLE_FILES behave as expected when naming backup directories for
    crashes and hangs.
```

```
<./include/config.h>
// #define SIMPLE_FILES
```

기본적으로는 disable 되어 있다. 이를 모르고 직접 체크해본 결과, disable되어있는 결과물을 확인할 수 있었다.

./src/afl-fuzz.c

[4] start - resume

```
if (!afl->in_place_resume) {

    fn = alloc_printf("%s/crashes/README.txt", afl->out_dir);
    unlink(fn);                                          /* Ignore errors */
    ck_free(fn);

  }
```

```
fn = alloc_printf("%s/crashes", afl->out_dir);
if (afl->in_place_resume && rmdir(fn)) {
    // 디렉터리가 비어 있지 않다면 백업 처리
    u8 *nfn = alloc_printf("%s_%04d%02d%02d%02d%02d%02d", fn, t.tm_year + 1900,
                           t.tm_mon + 1, t.tm_mday, t.tm_hour, t.tm_min, t.tm_sec);
    rename(fn, nfn);  // 디렉터리를 백업 이름으로 변경
    ck_free(nfn);
}
```

readme 지우고 crashes 를 백업한다.

./src/afl-fuzz.c

[4] start - resume

```
#ifndef SIMPLE_FILES

    u8 *nfn =
        alloc_printf("%s.%04d-%02d-%02d-%02d:%02d:%02d", fn, t.tm_year + 1900,
                     t.tm_mon + 1, t.tm_mday, t.tm_hour, t.tm_min, t.tm_sec);

#else

    u8 *nfn =
        alloc_printf("%s_%04d%02d%02d%02d%02d%02d", fn, t.tm_year + 1900,
                     t.tm_mon + 1, t.tm_mday, t.tm_hour, t.tm_min, t.tm_sec);

#endif
```

`SIMPLE_FILES` 활성화 여부에 따른 naming 차이가 있다.

이후에도 모든 임시파일들을 제거하는 로직이 진행되며, 불필요한 모든 레코드는 제거된다.(크래시는 백업)

./src/afl-fuzz.c

[4] start

다시 afl-fuzz.c로 돌아와서, 아래 코드로 복귀하게 된다.

```
#ifdef HAVE_AFFINITY
bind_to_free_cpu(afl);
#endif
```

```
<./include/afl-fuzz.h>
#if defined(__linux__) || defined(__FreeBSD__) || defined(__NetBSD__) || \
    defined(__DragonFly__) || defined(__sun)
  #define HAVE_AFFINITY 1
```

linux기준으로는 HAVE_AFFINITY 가 1임을 알 수 있다.

./src/afl-fuzz.c

[4] start

따라서 `bind_to_free_cpu` 함수가 호출되게 된다. (./src/afl-fuzz-init.c 에 존재)

```
if (afl->afl_env.afl_no_affinity && !afl->afl_env.afl_try_affinity) {
  if (afl->cpu_to_bind != -1) {
    FATAL("-b and AFL_NO_AFFINITY are mututally exclusive.");
  }
```

`-b` 옵션과 `AFL_NO_AFFINITY`가 동시에 설정되면 충돌을 방지하기 위해 에러를 발생
`AFL_NO_AFFINITY` 환경 변수가 설정된 경우 CPU 바인딩을 비활성화
-b 는 특정 코어를 bind하겠다는 의미로, 동적으로 바인딩 되는 것을 막는 것이 합리적이다.

./src/afl-fuzz.c

[4] start – bind cpu

```
<afl-fuzz.c>
case 'b': {                                          /* bind CPU core */

        if (afl->cpu_to_bind != -1) FATAL("Multiple -b options not supported");

        if (sscanf(optarg, "%d", &afl->cpu_to_bind) < 0) {

          FATAL("Bad syntax used for -b");

        }
```

afl-fuzz.c 의 arg 파싱 파트를 참고하면, `afl->cpu_to_bind` 는 -b <int> 의 int에 해당한다.

```
afl->cpu_to_bind = -1;
```

afl-fuzz-state.c 에서 afl init 시 `afl->cpu_to_bind` 기본 값이 -1로 초기화 되는 것을 볼 수 있다.

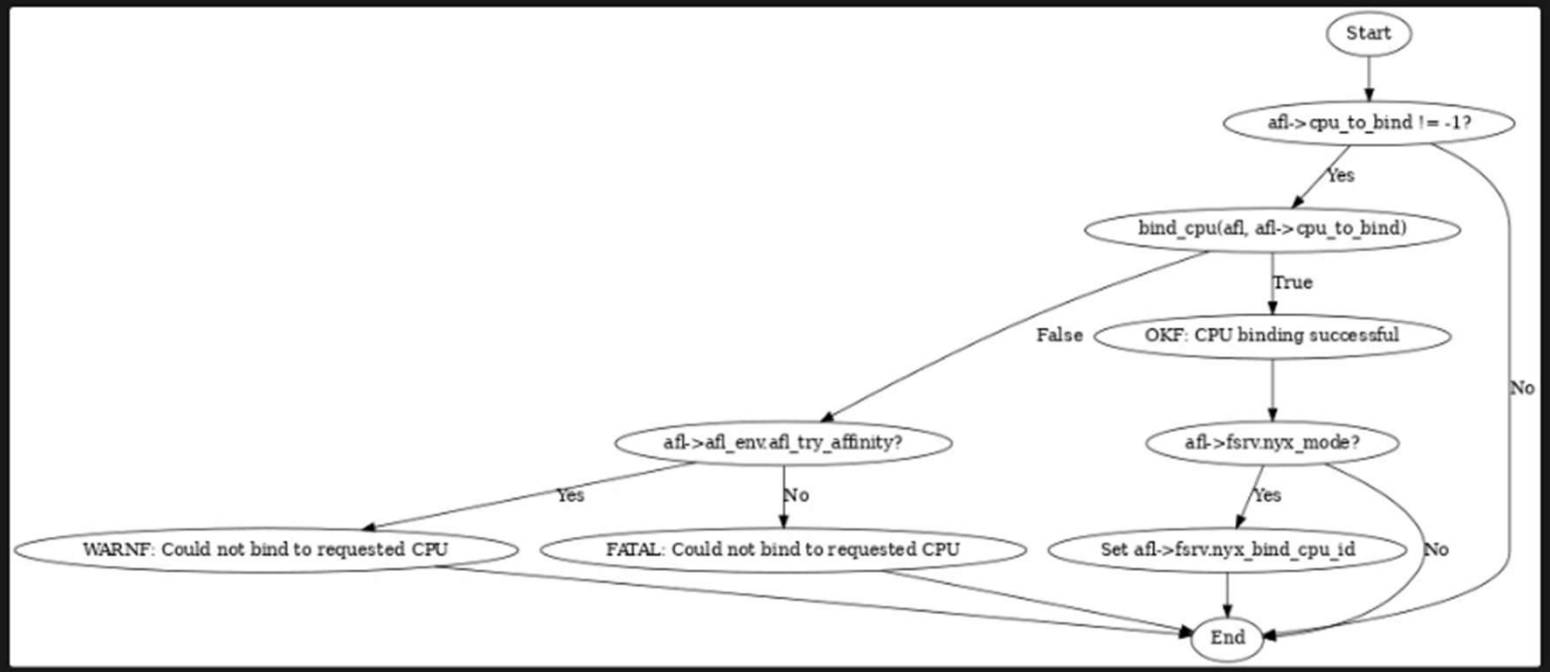
AFL_NO_AFFINITY 컨디션에서 Linux 환경이며 `nyx_mode` 가 활성화된 경우, `afl->fsrv.nyx_bind_cpu_id` 에 0을 저장한다.

./src/afl-fuzz.c

[4] start – bind cpu

`afl->cpu_to_bind` 가 -1이 아닌 정상 값인데, bind_cpu를 시도하고 실패 한 경우이면서 `afl->afl_env.afl_try_affinity` 가 활성화된 상황에서는 WARNING을 발생시킨다. (종료는 안 시킴)

bind_cpu를 실패하였음에도 `afl->afl_env.afl_try_affinity` 가 false인 경우에는 FATAL을 발생시킨다.(cpu_bind 실패 한 경우, -1)



./src/afl-fuzz.c

[4] start – bind cpu

`bind_cpu` 함수를 호출하여 cpu를 binding하는데, 이는 ./src/afl-fuzz-init.c 에서 정의된다.

```
static u8 bind_cpu(afl_state_t *afl, s32 cpuid) {
#if defined(__linux__) || defined(__FreeBSD__) || defined(__DragonFly__)
cpu_set_t c; // POSIX에서 사용하는 CPU 집합
#elif defined(__NetBSD__)
cpuset_t *c; // NetBSD에서 사용하는 CPU 집합 포인터
#elif defined(__sun)
psetid_t c; // Solaris/Illumos에서 사용하는 프로세서 세트 ID
#endif

  afl->cpu_aff = cpuid;
  #if defined(__linux__) || defined(__FreeBSD__) || defined(__DragonFly__
  CPU_ZERO(&c);
  CPU_SET(cpuid, &c);
  #elif defined(__NetBSD__)
```

./src/afl-fuzz.c

[4] start – bind cpu

AFL 상태 구조체에 `cpuid` 를 저장하고 cpu 집합을 초기화하는데, 플랫폼에 따라 적절하게 코딩되어 있다.

```
CPU_ZERO()
set을 비워서 어떤 CPU도 담지 않게 만들기.
CPU_SET()
CPU cpu를 set에 추가.
출처 : https://wariua.github.io/man-pages-ko/CPU_SET%283%29/
```

리눅스 기준으로는 CPU_ZERO, CPU_SET 을 사용하였다.

```
return (sched_setaffinity(0, sizeof(c), &c) == 0);
```

0 : 본인 프로세스, 0이 아니면 pid를 받는다고 한다.

c : 프로세서 셋

현재 프로세스를 c에 지정한 프로세서에 설정하겠다는 명령이다. 성공하면 0, 실패하면 1이 반환된다.

실제 return은 이와 반대되는 값이 전달 될 것이다.

./src/afl-fuzz.c

[4] start – bind cpu

```
if (afl->cpu_core_count < 2) { return; }
```

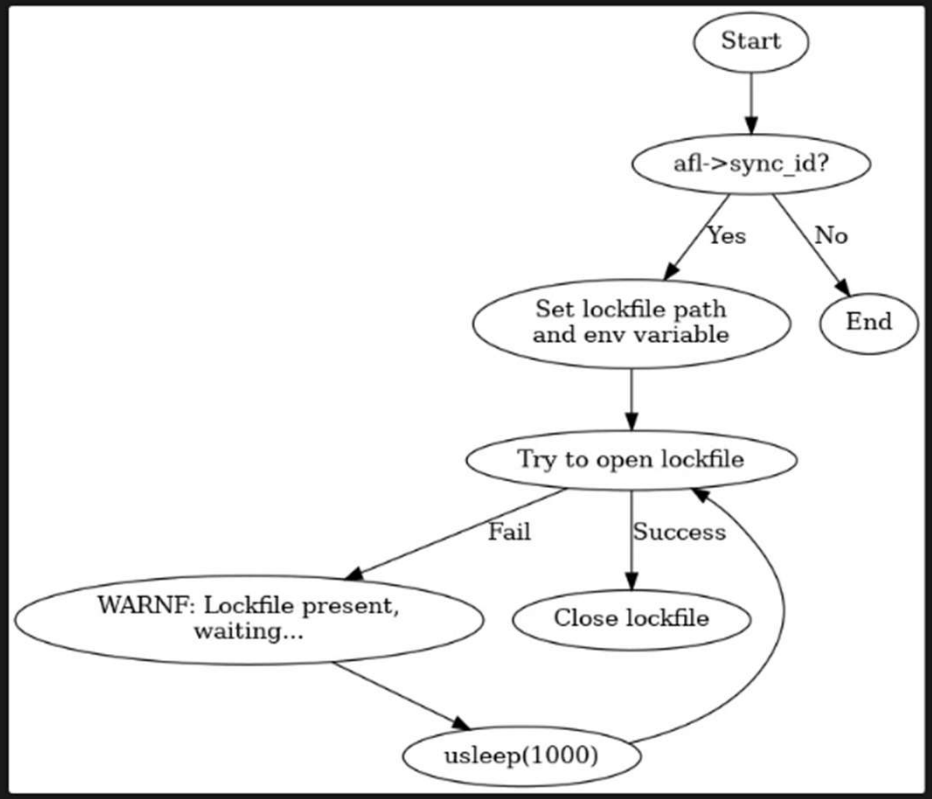
코어가 2개 미만이면 이하 `bind_to_free_cpu` 로직을 실행하지 않는다.

```
if (afl->sync_id) {
    s32 lockfd, first = 1;
    snprintf(lockfile, sizeof(lockfile), "%s/.affinity_lock", afl->sync_dir);
    setenv(CPU_AFFINITY_ENV_VAR, lockfile, 1);
    do {
      if ((lockfd = open(lockfile, O_RDWR | O_CREAT | O_EXCL,
                         DEFAULT_PERMISSION)) < 0) {
        if (first) {
          WARNF("CPU affinity lock file present, waiting ...");
          first = 0;
        }
        usleep(1000);
      }
    } while (lockfd < 0);
    close(lockfd);
  }
```

./src/afl-fuzz.c

[4] start – bind cpu

이전에 분석하였던 affinity_lock 코드가 여기서 사용된다.



`afl->sync_id` 활성화 된 경우, 특정 코어만 해당 자원(afl->sync_dir)을 점유할 때 까지 대기한다.

./src/afl-fuzz.c

[4] start – bind cpu

```
DIR *d;
struct dirent *de;
d = opendir("/proc");
if (!d) {
  if (lockfile[0]) unlink(lockfile);
  WARNF("Unable to access /proc - can't scan for free CPU cores.");
  return;
}
```

/proc 디렉터리 오픈, 실패 시 WARNING 발생 후 종료한다.

proc란?

> 리눅스에는 커널과 커널 모듈로부터 프로세스로 정보를 전송하기 위한 추가적인 메커니즘인 `/proc` 파일 시스템이 있습니다. 원래는 프로세스에 대한 정보를 쉽게 접근할 수 있도록 설계되었기 때문에 이름이 그렇게 붙었지만, 이제는 모듈 목록을 제공하는 `/proc/modules` 나 메모리 사용량 통계를 수집하는 `/proc/meminfo` 과 같이 커널의 모든 부분에 사용되고 있습니다.

> proc 파일 시스템을 사용하는 방법은 장치 드라이버를 사용하는 것과 매우 유사합니다. `/proc` 파일에 필요한 모든 정보를 포함하는 구조체를 만들고, 핸들러 함수(우리의 경우는 `/proc` 파일을 읽으려고 시도할 때 호출되는 하나만 있습니다)를 가리키는 포인터를 포함합니다. 그런 다음, `init_module` 은 커널에 구조체를 등록하고, `cleanup_module` 은 등록을 해제합니다.

> 일반적인 파일 시스템은 메모리(`/proc` 가 있는 곳)가 아닌 디스크에 위치하며, 이 경우 인덱스 노드(index-node, 줄여서 inode) 번호는 파일의 inode가 있는 디스크 위치를 가리키는 포인터입니다. inode는 파일의 권한과 같은 파일에 관한 정보를 담고 있으며, 파일 데이터가 저장된 디스크 위치를 가리키는 포인터도 포함하고 있습니다.

https://wikidocs.net/196798

./src/afl-fuzz.c

[4] start – bind cpu

```
root@whoami:/# ls /proc/
1     2572  282   394  56        bus         dma         kallsyms     locks        partitions   timer_list
1992  2573  283   434  6         cgroups     driver      kcore        mdstat       schedstat    tty
1993  2580  286   435  81        cmdline     execdomains key-users    meminfo      self         uptime
1995  2599  303   441  8678      config.gz   filesystems keys         misc         softirqs     version
2     2600  314   463  8679      consoles    fs          kmsg         modules      stat         vmallocinfo
2071  266   322   464  8686      cpuinfo     interrupts  kpagecgroup  mounts       swaps        vmstat
2073  267   3405  465  8999      crypto      iomem       kpagecount   mtrr         sys          zoneinfo
2074  275   3406  526  acpi      devices     ioports     kpageflags   net          sysvipc
2079  276   3413  527  buddyinfo diskstats   irq         loadavg      pagetypeinfo thread-self
```

```
root@whoami:/# cat /proc/1/status
Name:   systemd
Umask:  0000
State:  S (sleeping)
Tgid:   1
Ngid:   0
Pid:    1
PPid:   0
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 128
Groups:
NStgid: 1
NSpid:  1
NSpgid: 1
NSsid:  1
VmPeak:   231344 kB
VmSize:   165960 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:     11152 kB
VmRSS:     11152 kB
RssAnon:            3060 kB
RssFile:            8092 kB
RssShmem:              0 kB
VmData:    19080 kB
VmStk:       132 kB
VmExe:       896 kB
VmLib:      9056 kB
VmPTE:        92 kB
VmSwap:        0 kB
HugetlbPages:          0 kB
CoreDumping:    0
THP_enabled:    1
Threads:        1
SigQ:   0/127900
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 7be3c0fe28014a03
SigIgn: 0000000000001000
SigCgt: 00000001000004ec
CapInh: 0000000000000000
CapPrm: 000001ffffffffff
CapEff: 000001ffffffffff
CapBnd: 000001ffffffffff
CapAmb: 0000000000000000
NoNewPrivs:     0
Seccomp:        0
Seccomp_filters:        0
```

```
SpeculationIndirectBranch:
Cpus_allowed:   ffffffff
Cpus_allowed_list:      0-31
Mems_allowed:   1
Mems_allowed_list:      0
```

```
Seccomp:        0
Seccomp_filters:        0
Speculation_Store_Bypass:       thread vulnerable
SpeculationIndirectBranch:      conditional enabled
Cpus_allowed:   ffffffff
Cpus_allowed_list:      0-31
Mems_allowed:   1
Mems_allowed_list:      0
```

```
TH_MAX, "/proc/%s/status", de->d_name);

rocesses without vmsize are probably kernel

!strncmp(tmp, "VmSize:\t", 8)) { has_vmsize

!strncmp(tmp, "Cpus_allowed_list:\t", 19) &
!strchr(tmp, ',') && sscanf(tmp + 19, "%u",
hval < sizeof(cpu_used) && has_vmsize) {
```

./src/afl-fuzz.c

[4] start – bind cpu

proc 내부 폴더를 순회하면서 status를 읽고 VmSize, Cpus_allowed_list를 파싱한다.

```
u8  cpu_used[4096] = {0};
```

`VmSize`가 없는 프로세스는 커널 태스크로 간주되며, 이후 로직에서 제외

CPU 번호가 단일 숫자로만 표시 (`,` `-` 포함 시 제외).

CPU 번호(`hval`)가 `cpu_used` 배열 크기보다 작음.

위 두 조건에 맞는 경우, `cpu_used[hval]` 에 기록한다.

```
fclose(f);
```

닫고 마무리(아래는 다 플랫폼 코드 존재 + free한 cpu core에 bind, 생략)

./src/afl-fuzz.c

[4] start

다시 afl-fuzz.c로 돌아와서, 흐름을 따라가 보면, `init_count_class16` 함수가 호출 되는 것을 확인할 수 있다.

```
u16 count_class_lookup16[65536];
void init_count_class16(void) {
  u32 b1, b2;
  for (b1 = 0; b1 < 256; b1++) {
    for (b2 = 0; b2 < 256; b2++) {
      count_class_lookup16[(b1 << 8) + b2] =
          (count_class_lookup8[b1] << 8) | count_class_lookup8[b2];
    }
  }
}
```

./src/afl-fuzz-bitmap.c 에서 정의된 것을 확인할 수 있고, Lookup Table (LUT) 를 설정하는 것을 확인할 수 있다. LUT란 최적화를 위해 미리 계산된 값을 테이블 형태로 저장하는 것으로 `count_class_lookup16` 에 `b1` , `b2` 두 개의 8비트 변수로, 각각 0부터 255까지의 값을 반복하며 b1은 상위 8비트, b2는 하위 8비트를 결정하여 저장한다. 총 16비트의 값을 `count_class_lookup16` 에 저장해둔다.

./src/afl-fuzz.c

[4] start

```
if (afl->is_main_node && check_main_node_exists(afl) == 1) {

    WARNF("it is wasteful to run more than one main node!");
    sleep(1);

  } else if (!auto_sync && afl->is_secondary_node &&

             check_main_node_exists(afl) == 0) {

    WARNF(
        "no -M main node found. It is recommended to run exactly one main "
        "instance.");
    sleep(1);

  }
```

-M 인자로 시작한 master node가 없는 상태에서 -S (또는 인자 없이)실행한 경우, WARNING을 발생시킨다.

이후 인자 저장하는 로직과 디버깅용 플래그, 검증 단계들이 존재하며, `setup_custom_mutators` 함수가 호출되어 afl객체의 mutaion관련 셋팅을 적용하게 된다.

./src/afl-fuzz.c

[4] start

이어서 `check_binary` 라는 함수가 실행되고, afl-fuzz-init.c 에서 정의된 `check_binary` 함수가 binary의 실행 가능성을 검토한다.

```
if (stat(afl->fsrv.target_path, &st) || !S_ISREG(st.st_mode) ||
        !(st.st_mode & 0111) || (f_len = st.st_size) < 4) {

      FATAL("Program '%s' not found or not executable", fname);

    }
```

`stat` 함수가 실패하면 파일이 존재하지 않거나 exec 권한이 없는 경우이다.

`S_ISREG` 는 일반 파일인지 확인한다. 디렉터리나 symbolic 등은 일반 파일이 아니다. 일반 파일이 아니면 true이기에 조건문에 !가 붙는다.

`st.st_mode` 는 파일의 실행 권한을 확인한다. 0111은 실행 권한 비트를 의미하며, 이를 마스크로 and 연산하여 결과가 0이면 실행 권한이 없다는 것이다. 이 경우 true가 리턴되기에 조건문에 ! 가 붙는다.

`st.st_size` 로 파일의 사이즈가 4보다 작으면 true가 리턴된다. 이 경우에는 실행 파일의 최소한의 헤더가 포함되지 않기에 실행 파일이 아닌 것으로 간주된다.

이러한 로직을 통하여 실행 파일이 아니거나 불가능 한 상태일 때 FATAL을 발생시키게 된다.

./src/afl-fuzz.c

[4] start

Core Part!

AFL != AFL plus plus

./src/afl-fuzz.c

afl-fuzz.c

```
read_testcases
      |
      v
pivot_inputs
```

./src/afl-fuzz.c

afl-fuzz.c

`read_testcases`

```
if (!S_ISREG(st.st_mode) || !st.st_size || strstr(fn2, "/README.txt")) {
```

조건에 맞는 파일을 읽어옴

```
if (subdirs && S_ISDIR(st.st_mode) && nl[i]->d_name[0] != '.') {
  free(nl[i]);
  read_testcases(afl, fn2);
  ck_free(fn2);
  goto next_entry;
}
```

재귀함수로 sub dir까지 읽어옴

./src/afl-fuzz.c

afl-fuzz.c

read_testcases

```
add_to_queue(afl, fn2, st.st_size >= MAX_FILE ? MAX_FILE : st.st_size,
passed_det);
```

queue에 추가, MAX_FILE 보다 클 경우, 짤라서 읽음

./src/afl-fuzz.c

afl-fuzz.c

`read_testcases -> add_to_queue`

```
add_to_queue(afl, fn2, st.st_size >= MAX_FILE ? MAX_FILE : st.st_size,
passed_det);
```

queue 자료형을 구현하고 afl구조체에 업데이트

```
++afl->queued_items;
++afl->active_items;
++afl->pending_not_fuzzed;
afl->cycles_wo_finds = 0;
```

queued_items: 총 큐 항목 수를 증가시킵니다.
active_items: 활성화된 항목 수를 증가시킵니다.
pending_not_fuzzed: 아직 퍼징되지 않은 항목 수를 증가시킵니다.
cycles_wo_finds: 최근 발견 이후의 사이클 수를 초기화합니다.

./src/afl-fuzz.c

afl-fuzz.c

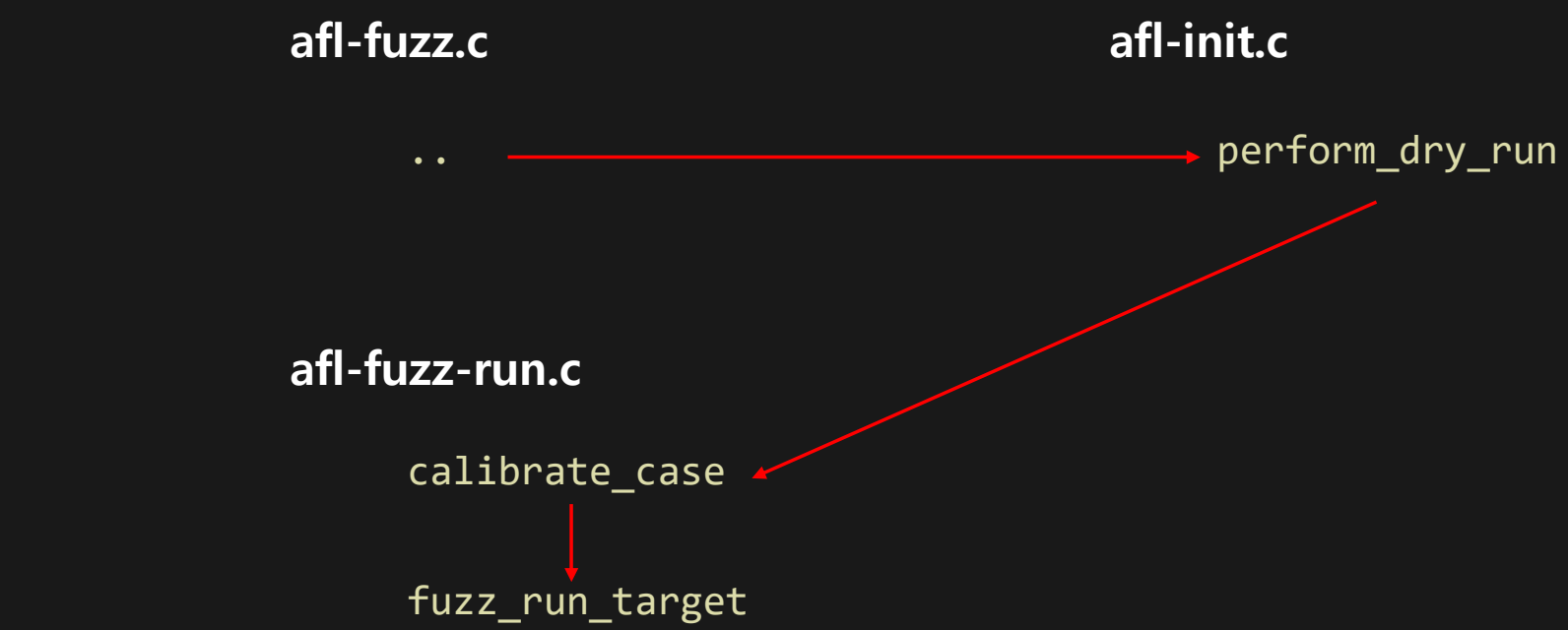
`pivot_inputs`

```
for (i = 0; i < afl->queued_items && likely(afl->queue_buf[i]); i++) {
  q = afl->queue_buf[i];
  if (unlikely(q->disabled)) { continue; }
}
```

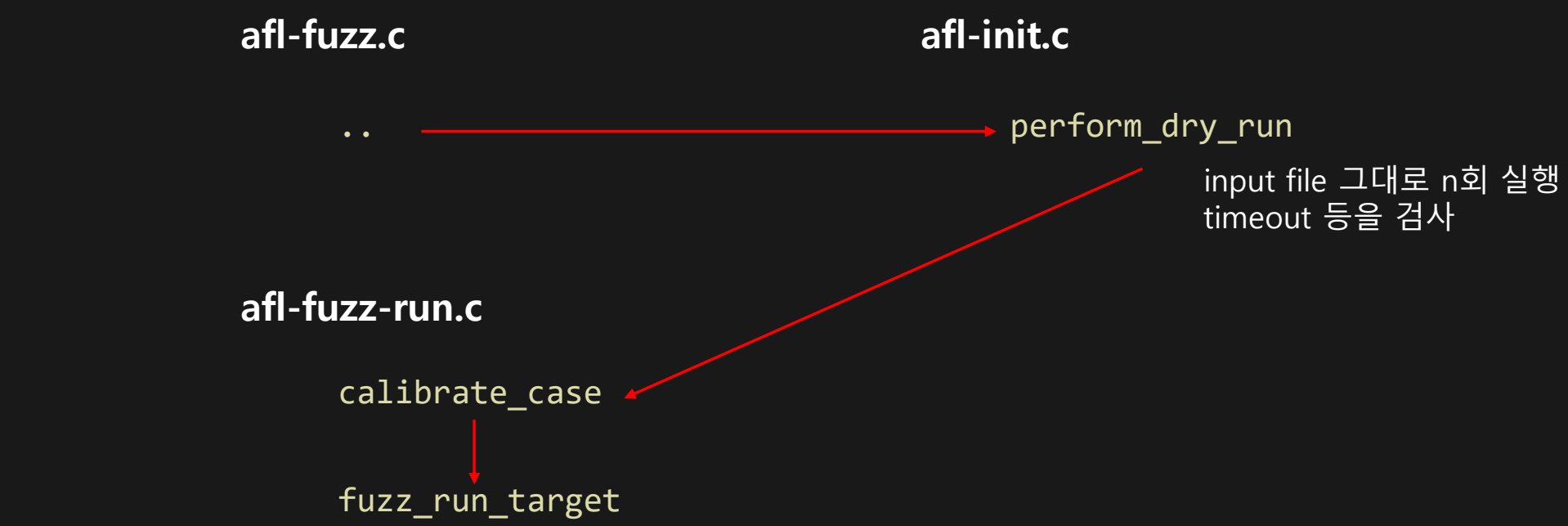
queue 선회하며 disable 된 데이터 skip

*기존 테스트 케이스를 queue폴더로 가져가는 함수

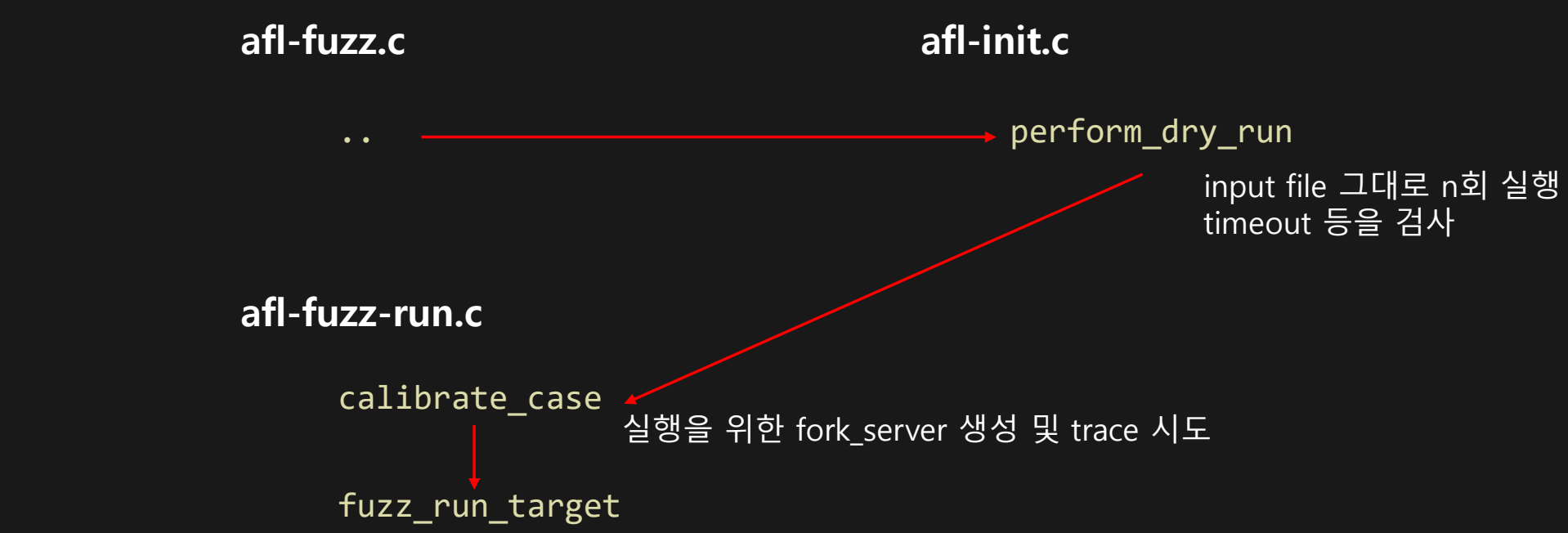
./src/afl-fuzz.c



./src/afl-fuzz.c



./src/afl-fuzz.c



./src/afl-fuzz.c

afl-fuzz-run.c

fuzz_run_target

```
fsrv_run_result_t fuzz_run_target(afl_state_t *afl, afl_forkserver_t *fsrv, u32 timeout) {

fsrv_run_result_t res = afl_fsrv_run_target(fsrv, timeout, &afl->stop_soon);
```



./src/afl-fuzz.c

afl-fuzz-run.c

fuzz_run_target

```
fsrv_run_result_t fuzz_run_target(afl_state_t *afl, afl_forkserver_t *fsrv, u32 timeout) {
```

call

```
fsrv_run_result_t res = afl_fsrv_run_target(fsrv, timeout, &afl->stop_soon);
```

mode 별 실행 로직

__linux__ : child process생성, pid 읽어오기, crash 탐지, timeout 탐지

./src/afl-fuzz.c

afl-fuzz-run.c

fuzz_run_target

```
fsrv_run_result_t fuzz_run_target(afl_state_t *afl, afl_forkserver_t *fsrv, u32 timeout) {
```

call

```
fsrv_run_result_t res = afl_fsrv_run_target(fsrv, timeout, &afl->stop_soon);
```

mode 별 실행 로직

__linux__ : child process생성, pid 읽어오기, crash 탐지, timeout 탐지

./src/afl-fuzz.c

afl-fuzz-run.c

fuzz_run_target

```
fsrv_run_result_t fuzz_run_target(afl_state_t *afl, afl_forkserver_t *fsrv, u32 timeout) {
```

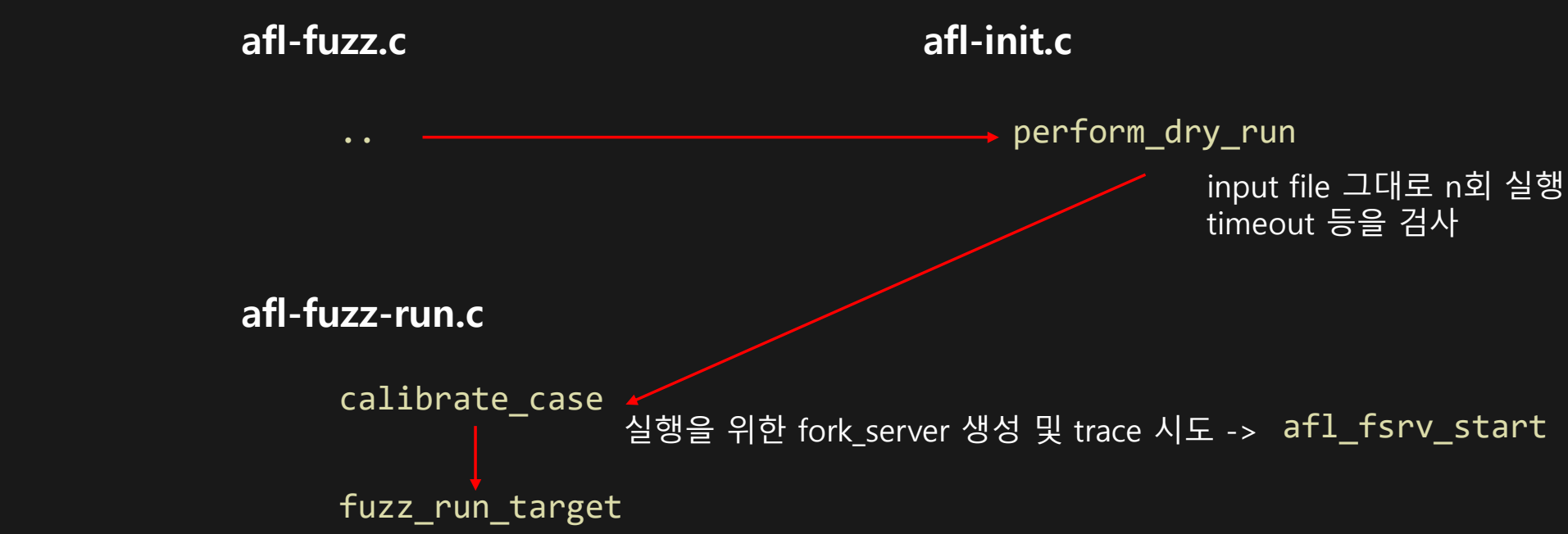



```
fsrv_run_result_t res = afl_fsrv_run_target(fsrv, timeout, &afl->stop_soon);
```

coverage 측정 로직도 포함되어 있다.

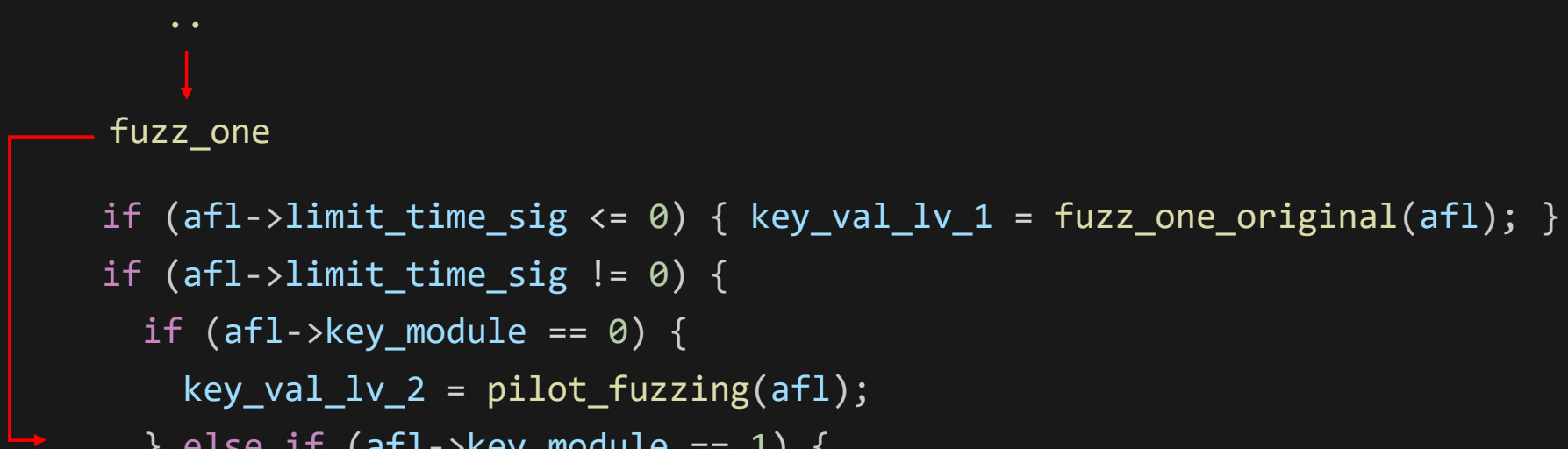
뒤에서 따로 다룰 예정

./src/afl-fuzz.c



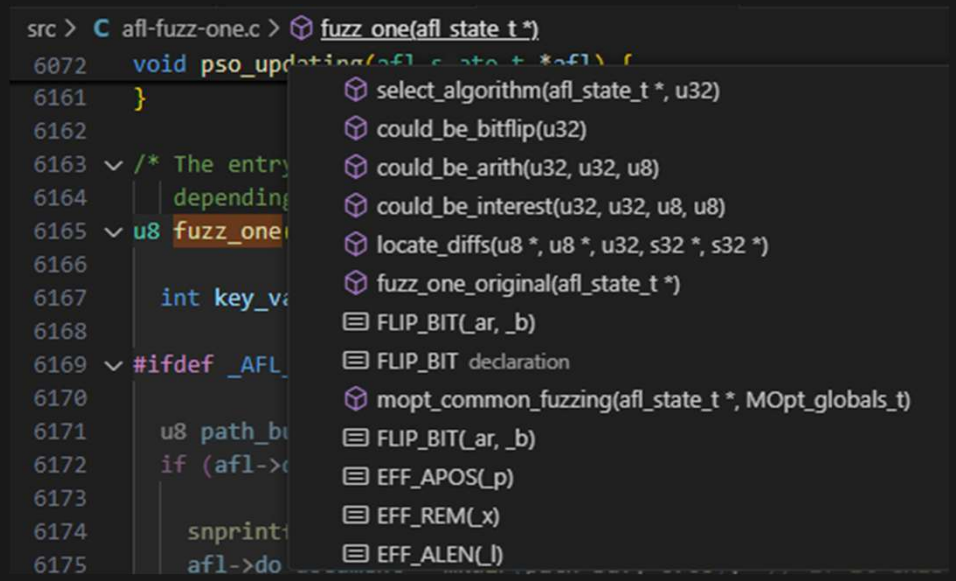
./src/afl-fuzz.c

afl-fuzz.c



./src/afl-fuzz.c

afl-fuzz.c



./src/afl-fuzz.c

afl-fuzz.c

```
#define FLIP_BIT(_ar, _b)                          \
  do {                                             \
                                                   \
    u8 *_arf = (u8 *)(_ar);                        \
    u32 _bf = (_b);                                \
    _arf[(_bf) >> 3] ^= (128 >> ((_bf) & 7));      \
                                                   \
  } while (0)

  /* Single walking bit. */

  afl->stage_short = "flip1";
  afl->stage_max = len << 3;
  afl->stage_name = "bitflip 1/1";

  afl->stage_val_type = STAGE_VAL_NONE;

  orig_hit_cnt = afl->queued_items + afl->saved_crashes;

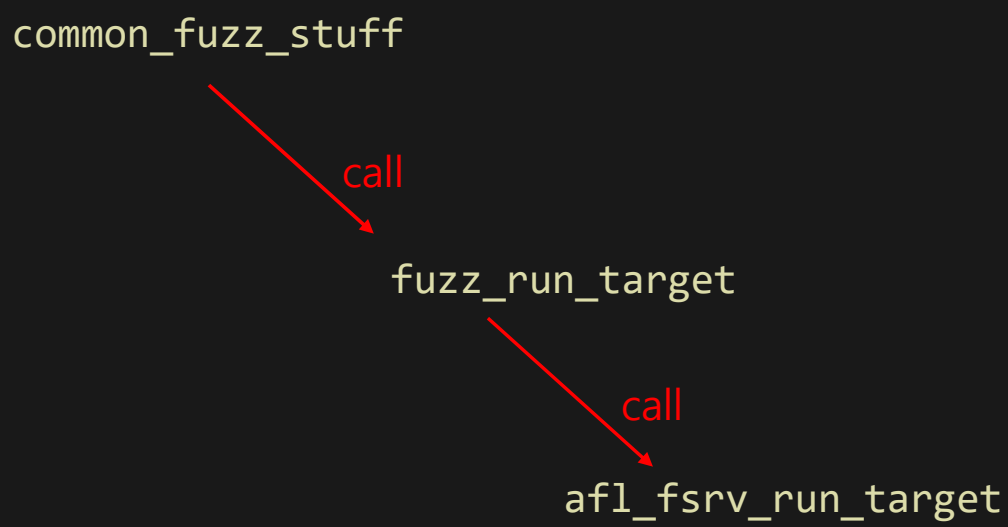
  /* Get a clean cksum. */

  if (common_fuzz_stuff(afl, out_buf, len)) { goto abandon_entry; }
```

afl-fuzz-run.c

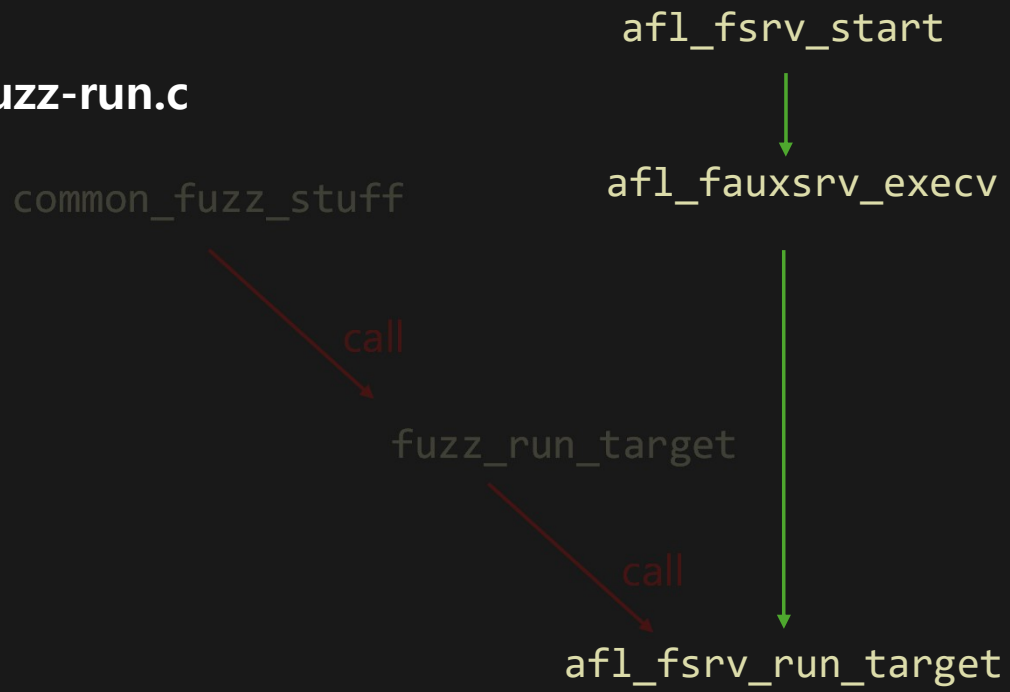
./src/afl-fuzz.c

afl-fuzz-run.c



./src/afl-fuzz.c

afl-fuzz-run.c



./src/afl-fuzz.c

afl-fuzz-run.c

afl_fauxsrv_execv

```
if (pipe(st_pipe) || pipe(ctl_pipe)) { PFATAL("pipe() failed"); }

fsrv->last_run_timed_out = 0;
fsrv->fsrv_pid = fork();
```

./src/afl-fuzz.c

afl-fuzz-run.c

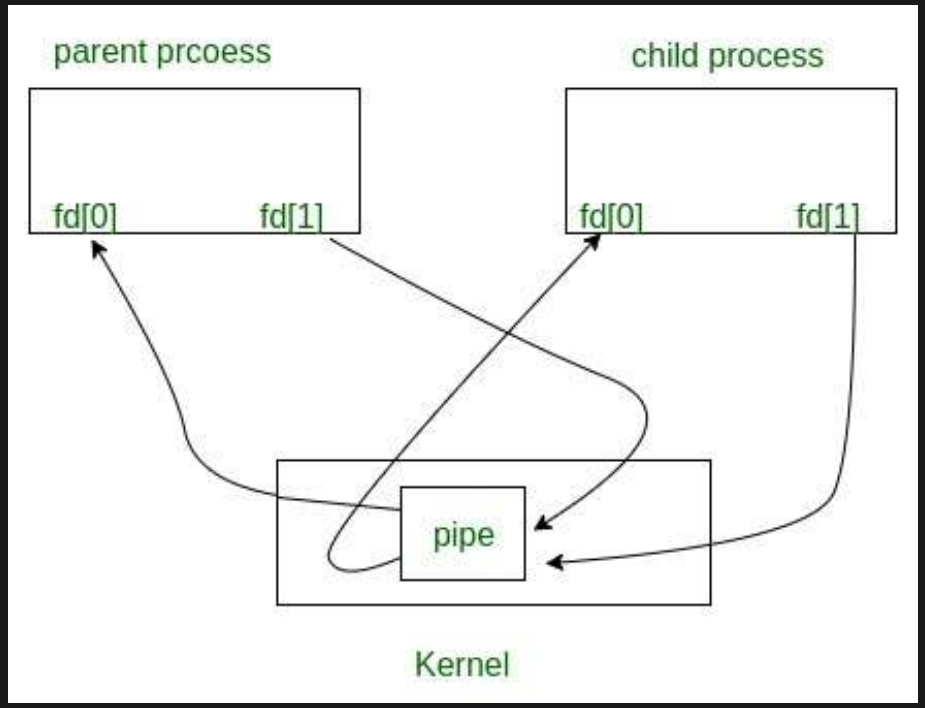
ctl_pipe :

- AFL++가 포크 서버에 제어 명령 전달
- 새 프로세스 요청
- 타임아웃 또는 데이터 전달

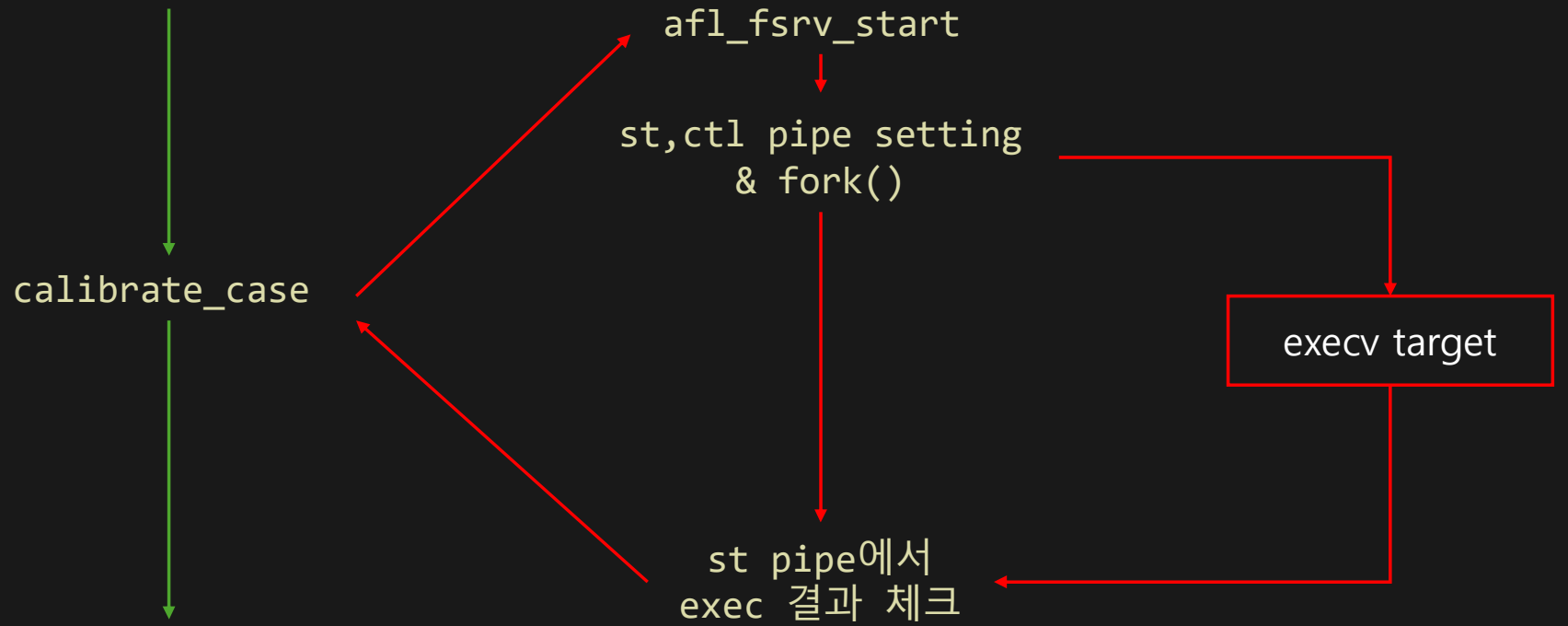
st_pipe :

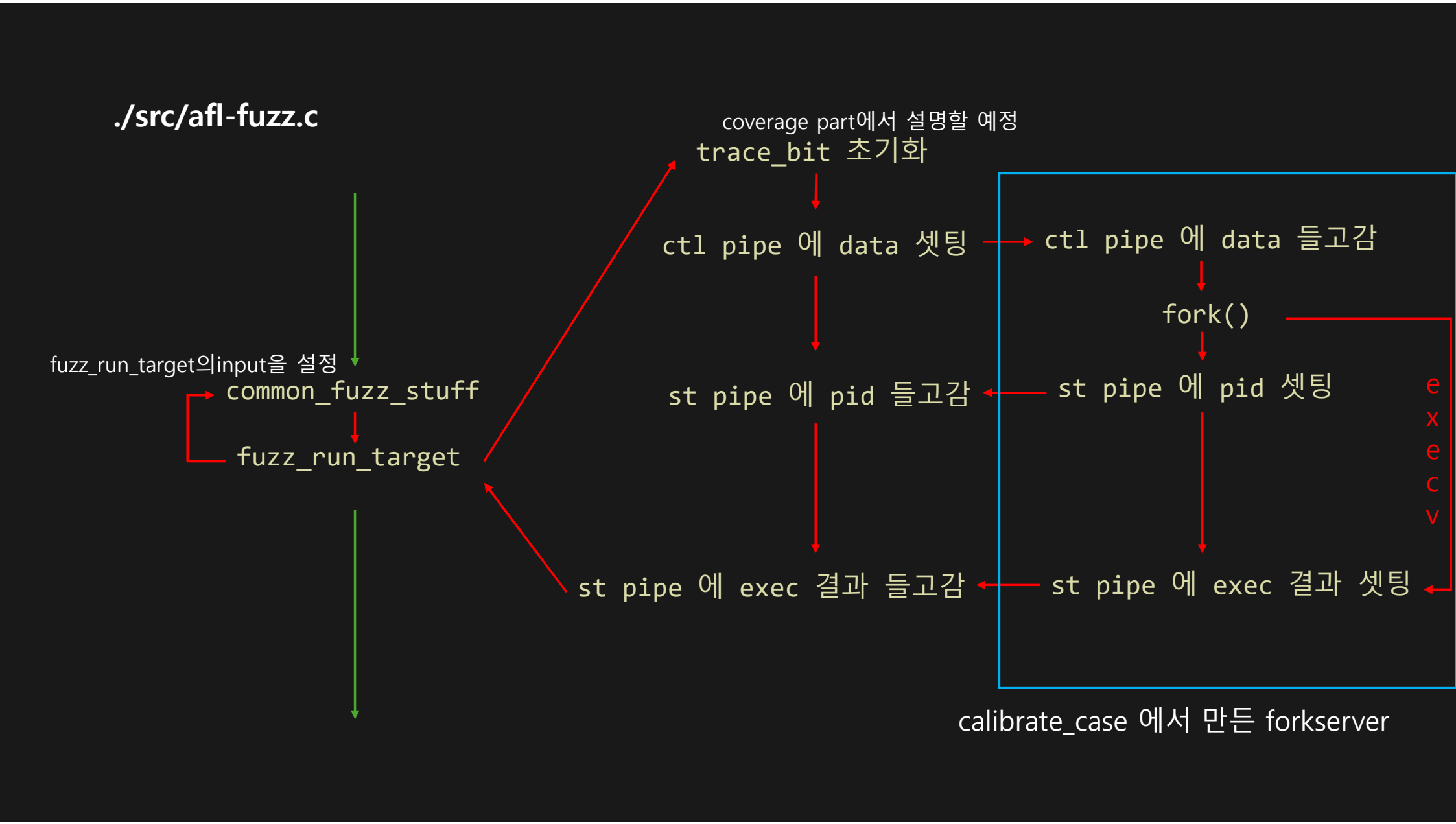
- 포크 서버가 AFL++로 상태 전달
- 자식 프로세스 PID 및 종료 상태 전달

./src/afl-fuzz.c



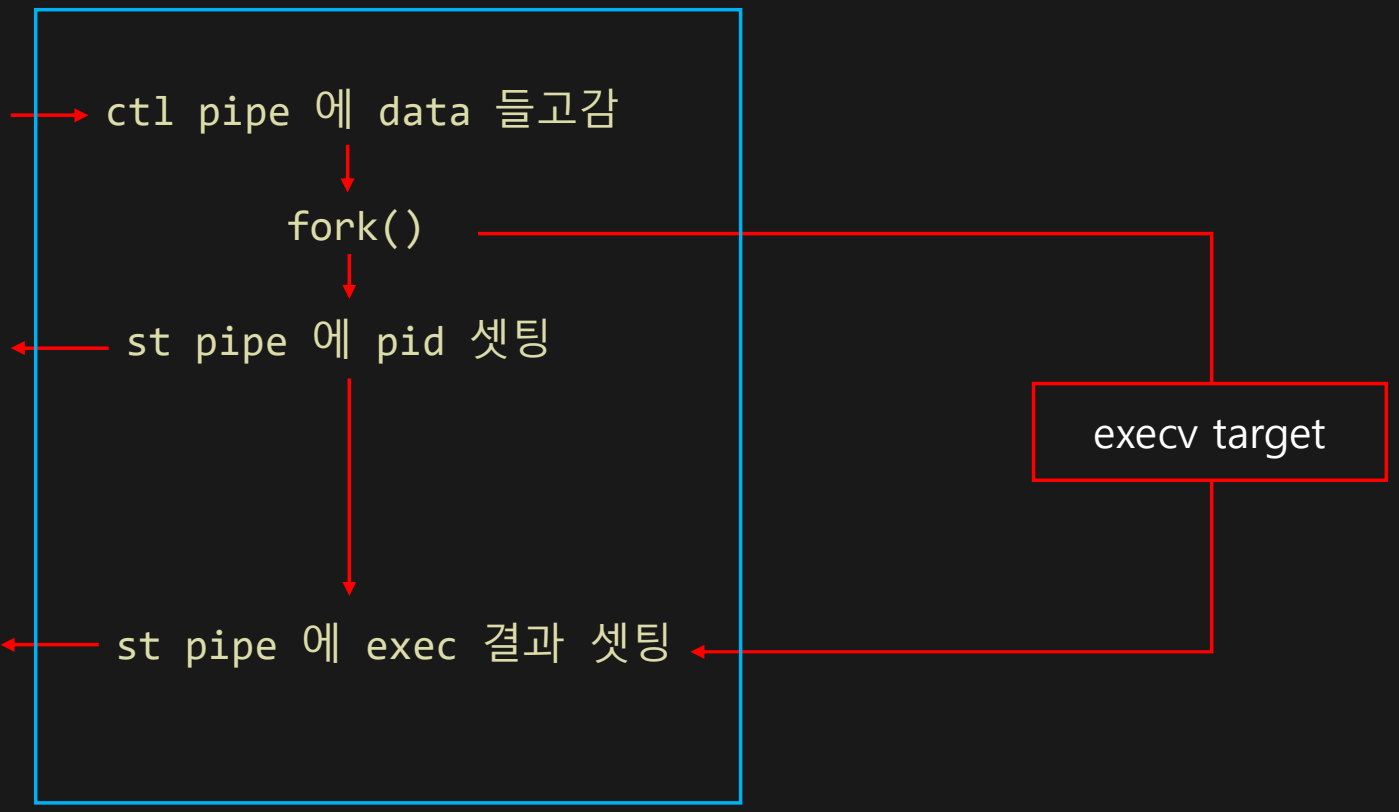
./src/afl-fuzz.c




./src/afl-fuzz.c
coverage part에서 설명할 예정
trace_bit 초기화
ctl pipe 에 data 셋팅
ctl pipe 에 data 들고감
fork()
fuzz_run_target의input을 설정
common_fuzz_stuff
fuzz_run_target
st pipe 에 pid 들고감
st pipe 에 pid 셋팅
execv
st pipe 에 exec 결과 들고감
st pipe 에 exec 결과 셋팅
calibrate_case 에서 만든 forkserver

coverage

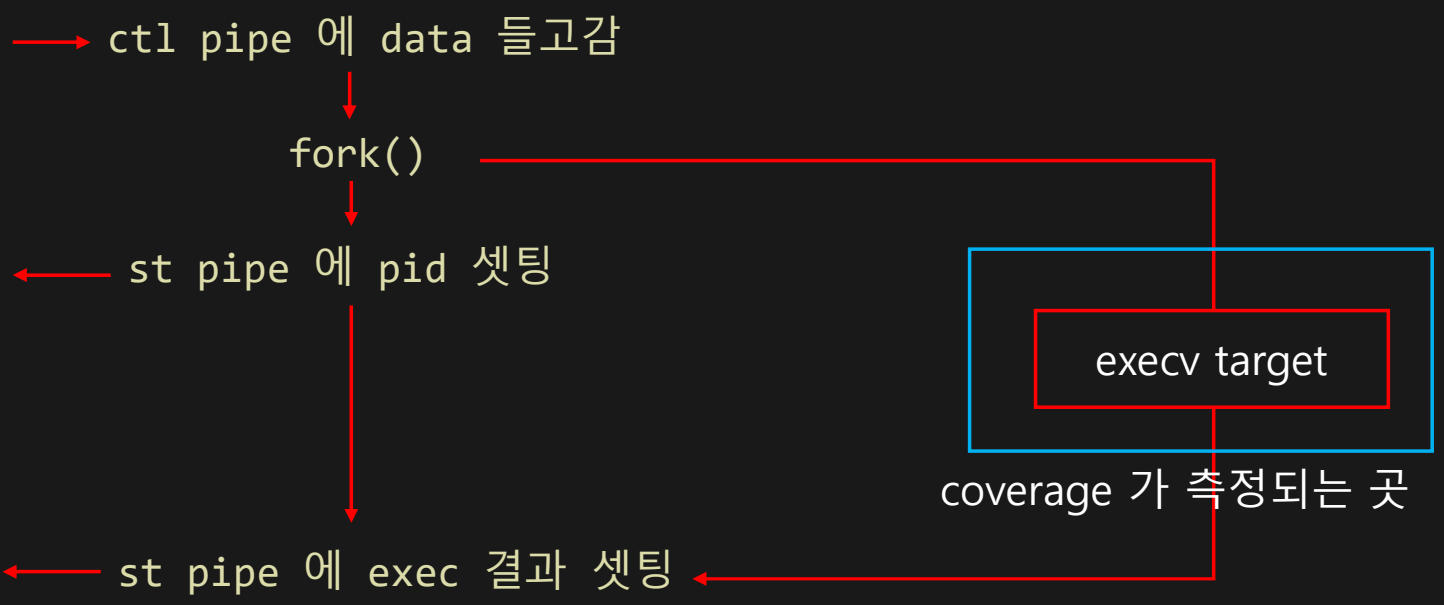
trace_bit 초기화



calibrate_case 에서 만든 forkserver

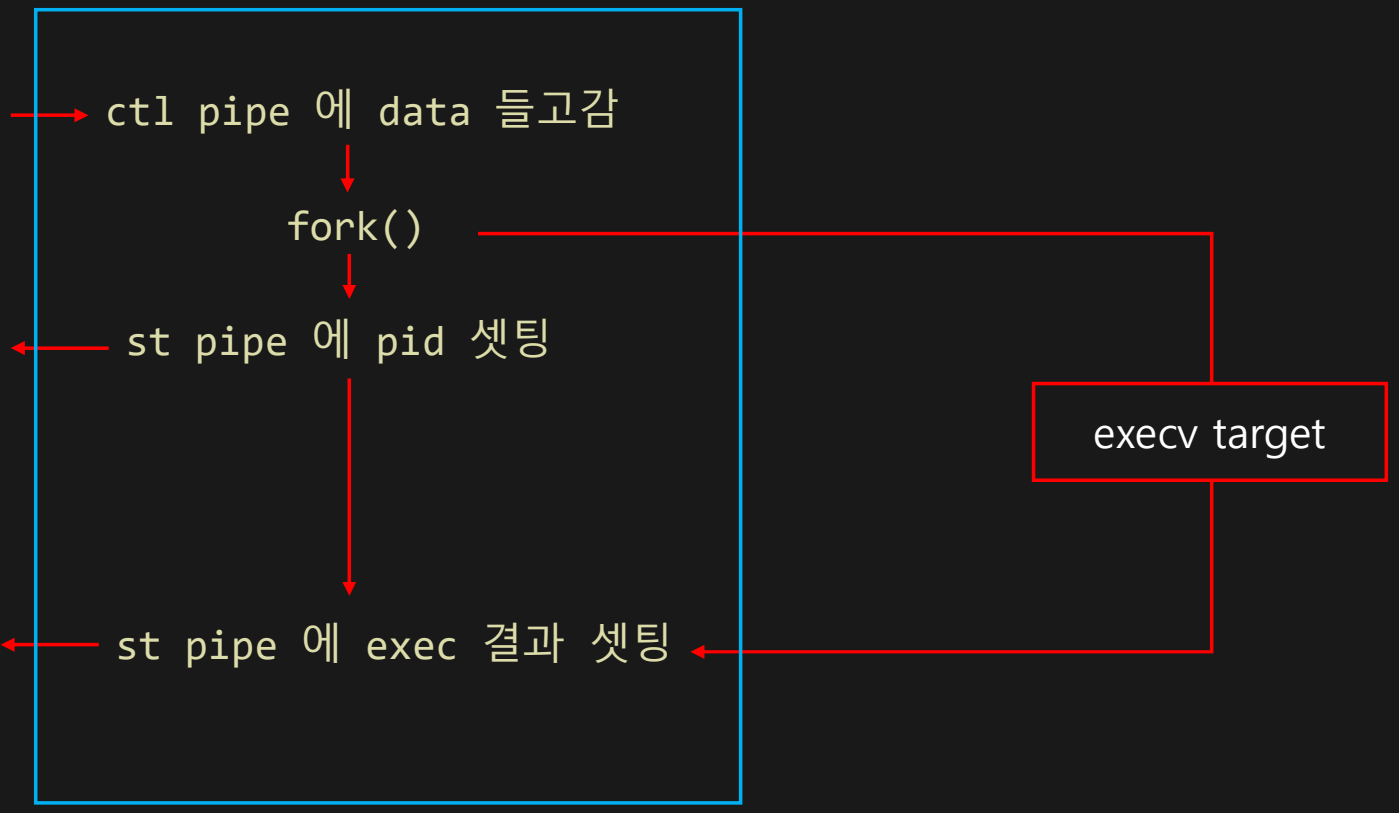
coverage

`trace_bit 초기화`



coverage

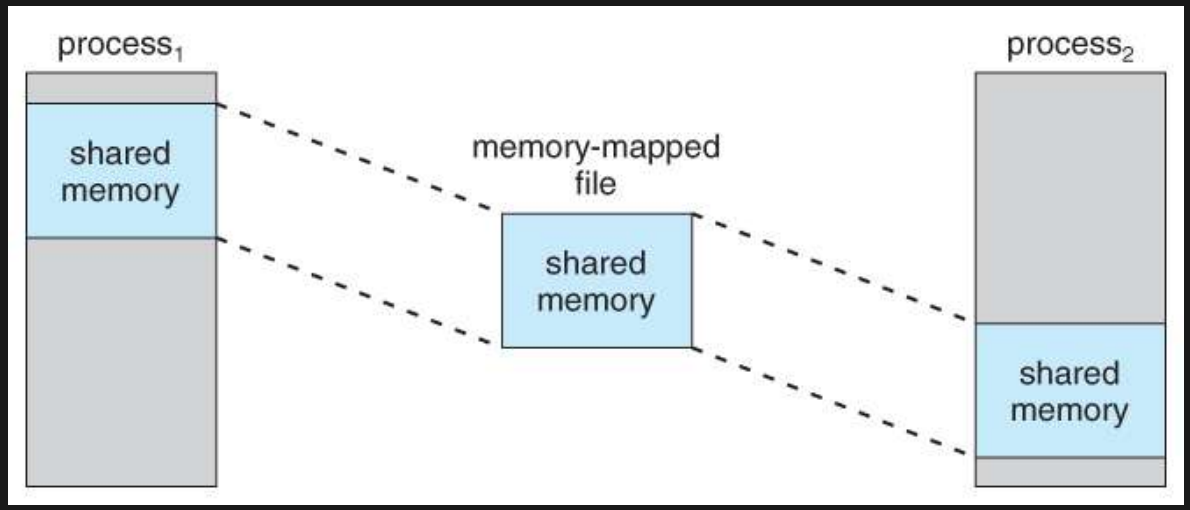
trace_bit 초기화

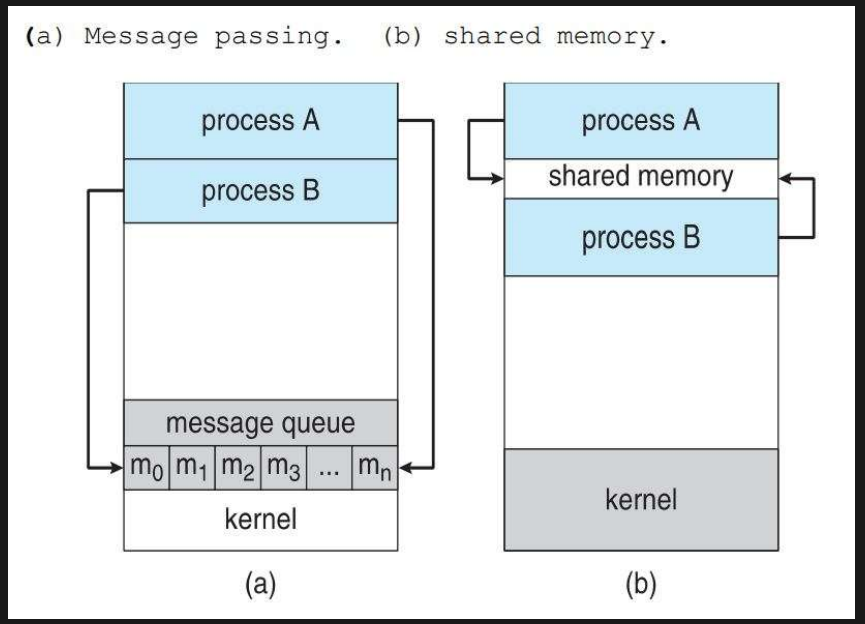


coverage 를 알아야 하는 곳

coverage

`trace_bit` 초기화

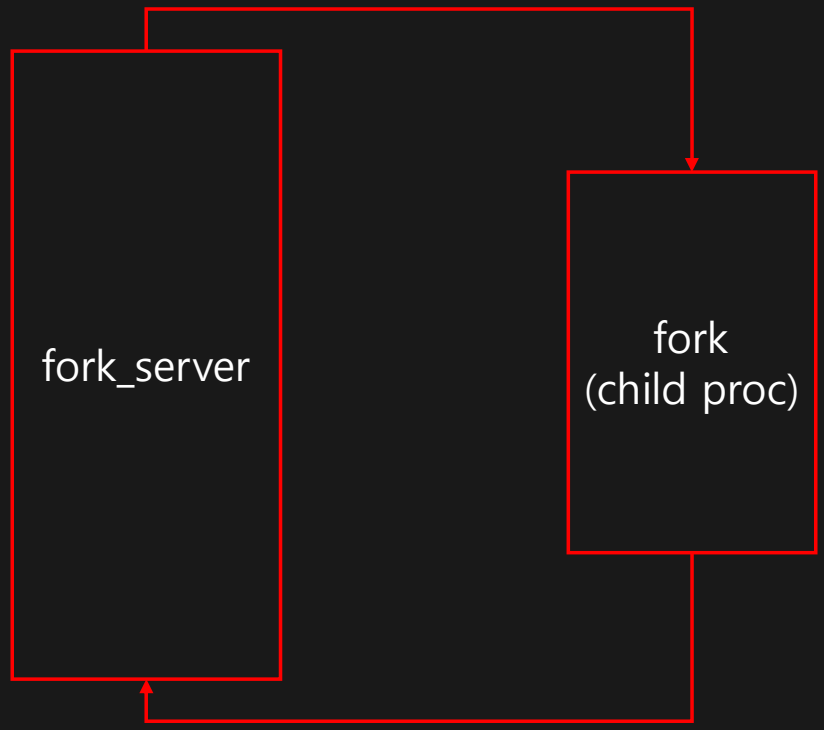




Ref : 시스템 보안 ppt...

coverage

`trace_bit` 초기화



coverage

`trace_bit` 초기화



coverage

trace_bit 초기화

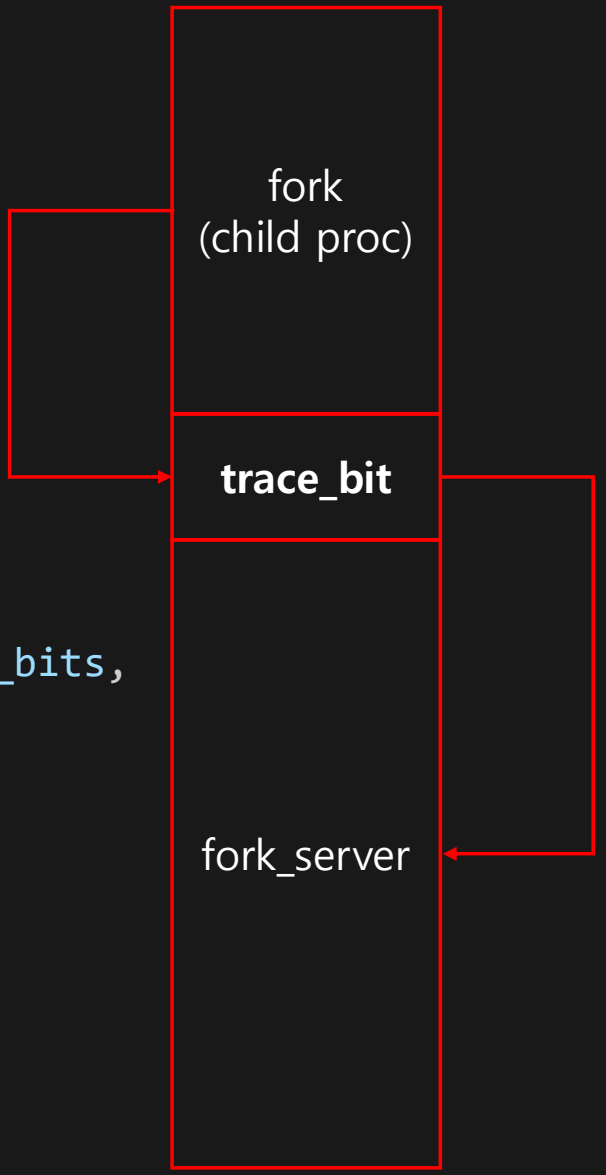
```
afl->fsrv.shmem_fuzz_len = (u32 *)map;
afl->fsrv.shmem_fuzz = map + sizeof(u32);
```



coverage

`trace_bit 초기화`

?



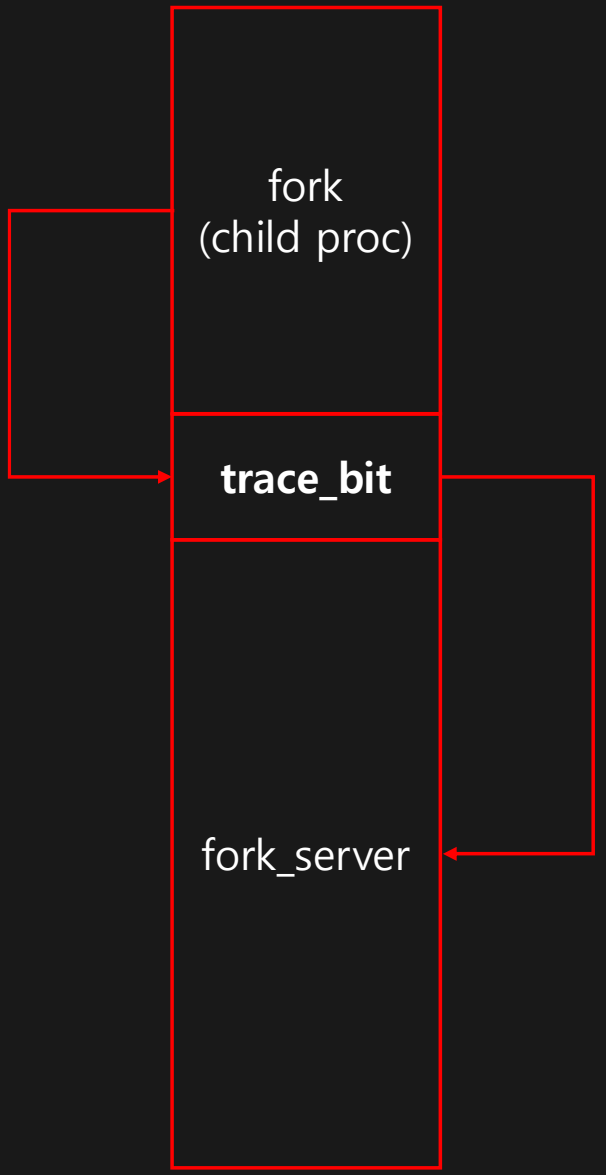

```
memcpy(afl->first_trace, afl->fsrv.trace_bits,
                afl->fsrv.map_size);
```

coverage

`trace_bit 초기화`

LLVM, afl-cc, ETC.

binary에 코드가 삽입되어 있다



coverage

Instrumentation code audit

coverage

LLVM sanitizer / LLVM coverage

말고도 많은데, 시간 상 LLVM 선택하여 분석

coverage

LLVM sanitizer

```
Value *GuardPtr = IRB.CreateIntToPtr(
            IRB.CreateAdd(
                IRB.CreatePointerCast(FunctionGuardArray, IntptrTy),
                ConstantInt::get(IntptrTy, (++special + AllBlocks.size()) * 4)),
            Int32PtrTy);

        LoadInst *Idx = IRB.CreateLoad(IRB.getInt32Ty(), GuardPtr);
        ModuleSanitizerCoverageAFL::SetNoSanitizeMetadata(Idx);

        callInst->setOperand(1, Idx);
```

./instrumentation/SanitizerCoveragePCGUARD.so.cc

coverage

LLVM sanitizer

```
LoadInst *CurLoc = IRB.CreateLoad(IRB.getInt32Ty(), GuardPtr);
ModuleSanitizerCoverageAFL::SetNoSanitizeMetadata(CurLoc);

/* Load SHM pointer */

LoadInst *MapPtr = IRB.CreateLoad(PointerType::get(Int8Ty, 0), AFLMapPtr);
ModuleSanitizerCoverageAFL::SetNoSanitizeMetadata(MapPtr);
```

./instrumentation/SanitizerCoveragePCGUARD.so.cc

coverage

LLVM sanitizer

AFLMapPtr는 __afl_area_ptr라는 글로벌 변수를 참조하여
AFL++가 사용하는 공유 메모리 영역

InjectCoverageAtBlock:
trace-pc-guard 방식으로 커버리지 정보를 삽입

instrumentFunction:
특정 함수의 기본 블록들에 대해 커버리지 관련 코드를 삽입하는 작업

./instrumentation/SanitizerCoveragePCGUARD.so.cc

coverage

LLVM coverage

calcCyclomaticComplexity:

llvm::Function(타겟 함수)의 복잡도를 측정하기 위해 Cyclomatic Complexity (CC)를 계산

V(G)=E−N+2P

E: 엣지(Edge) 개수 (기본 블록의 후속 블록 개수 포함)

N: 노드(Node) 개수 (기본 블록 개수)

P: 연결된 컴포넌트의 개수 (일반적으로 함수 내부는 1로 고정됨)

Ref : https://en.wikipedia.org/wiki/Cyclomatic_complexity

./instrumentation/afl-llvm-common.cc

coverage

LLVM coverage

calculateCollisions:

balls in bins problem 알고리즘으로 구현

공(balls)은 엣지(edge) 개수.
통(bins)은 AFL++의 맵 크기(MAP_SIZE).

충돌 가능성을 출력

./instrumentation/afl-llvm-common.cc

coverage

LLVM coverage

balls in bins problem:

"**Balls in bins problem**"은 확률론에서 많이 사용되는 문제로, 주어진 개수의 공(**balls**)을 일정한 수의 통(**bins**)에 무작위로 배치했을 때 특정 상황이 발생할 확률을 분석하는 문제

Ref : https://en.wikipedia.org/wiki/Balls_into_bins_problem

./instrumentation/afl-llvm-common.cc

LLVM coverage

AFL-whitelist, blacklist 등의 구현은 ppt에서 생략

./instrumentation/afl-llvm-common.cc

coverage

LLVM coverage

```
GlobalVariable *AFLMapPtr =
    new GlobalVariable(M, PointerType::get(Int8Ty, 0), false,
                       GlobalValue::ExternalLinkage, 0, "__afl_area_ptr");
```

./instrumentation/afl-llvm-pass.so.cc

mutation

Mutation code audit

mutation

~~Custom Mutation~~ / Default Mutation

mutation

logic

afl-fuzz.c 에 존재

```
while (likely(!afl->stop_soon)) {
아래 360줄 정도 mutation logic

}

stop_fuzzing:
퍼징 종료 로직
```

mutation

```
time_t      cursec = time(NULL);
struct tm *curdate = localtime(&cursec);
if (unlikely(!afl->afl_env.afl_pizza_mode)) {
  if (unlikely(curdate->tm_mon == 3 && curdate->tm_mday == 1)) {
    afl->pizza_is_served = 1;
  } else {
    afl->pizza_is_served = 0;
  }
}
```

4/1 이면 pizza mode가 실행된다...
*C언어는 tm_mon==0이 1월이다

mutation

```
  if (afl->cycle_schedules) {
          /* we cannot mix non-AFLfast schedules with others */
          switch (afl->schedule) {
            case EXPLORE:
              afl->schedule = EXPLOIT;
              break;
            case EXPLOIT:
              afl->schedule = MMOPT;
              break;
            case MMOPT:
              afl->schedule = SEEK;
              break;
            case SEEK:
              afl->schedule = EXPLORE;
              break;
```

mutation

```
switch (afl->expand_havoc) {

            case 0:
              // this adds extra splicing mutation options to havoc mode
              afl->expand_havoc = 1;
              break;
            case 1:
```

0:Splicing 기반의 추가 mutation 옵션을 havoc 모드에 추가

1:Mopt 활성화

2:Havoc 모드에서 시도 당 최대 mutation 횟수 증가

3.2단계 1번 더/

4.아직 구현되지 않은 코드라고 주석만 존재

mutation

splicing은 기존 입력 데이터를 조합하여 새로운 테스트 케이스를 생성

Mutation Optimization은 기존 mutation의 결과를 통해 다음 mutation의 확률을 동적으로 설정

0:Splicing 기반의 추가 mutation 옵션을 havoc 모드에 추가
1:Mopt 활성화
2:Havoc 모드에서 시도 당 최대 mutation 횟수 증가
3.2단계 1번 더/
4.아직 구현되지 않은 코드라고 주석만 존재

mutation

core

afl-fuzz-one.c

```
/*
    -L command line paramter => limit_time_sig value
      limit_time_sig == 0 then run the default mutator //이거고
      limit_time_sig  > 0 then run MOpt
      limit_time_sig  < 0 both are run
*/
```

fuzz_one → fuzz_one_original

```
327  > u8 fuzz_one_original(afl_state_t *afl) { ···
3454
```

mutation

Deterministic Stages - Bit-Flips

flip 1
입력의 각 비트를 뒤집어본다(bit ^= 1)

flip 2
연속된 2개 비트를 뒤집어본다

flip 4
연속된 4개 비트를 뒤집어본다

flip 8
각 바이트를 뒤집어본다(^= FF)

flip 16
연속된 2개 바이트를 뒤집어본다(^= FFFF)

flip 32
연속된 4개 바이트를 뒤집어본다(^= FFFFFFFF)

각 시드 당 1번 실행, 새로운 커버리지가 발견되면 queue에 등록하고, 그 외의 경우는 원상복구 시킨다

플립 시 커버리지 영향이 없는 바이트는 건너뛴다

mutation

Deterministic Stages - Arithmetic Mutations

arith 8
각 바이트에 대해 최대 ARITH_MAX(기본값=35)까지 덧셈/뺄셈 수행

arith 16
2바이트 워드(리틀엔디언/빅엔디언 모두) 기준으로 덧셈/뺄셈

arith 32
4바이트 워드(리틀엔디언/빅엔디언 모두) 기준으로 덧셈/뺄셈

비트 플립과 같은 결과가 나타나면 skip

비트 플립과 동일하게, 특정 오프셋에 영향이 안가는 경우 넘어간다

mutation

Deterministic Stages - Interesting Mutations

interesting 8 16 32
미리 지정해둔 값으로 대입해본다

```
s8  interesting_8[] = {INTERESTING_8};
s16 interesting_16[] = {INTERESTING_8, INTERESTING_16};
s32 interesting_32[] = {INTERESTING_8, INTERESTING_16, INTERESTING_32};
```

확실하진 않지만, 해당 배열이 interesting 한 값을 저장한 배열로 보임(레퍼러 분석안해봄)

비트 플립, arith에서 해봤으면 skip

mutation

Deterministic Stages - Interesting Mutations

```
#define INTERESTING_8                                       \
  -128,    /* Overflow signed 8-bit when decremented  */ \
      -1,  /*                                         */ \
      0,   /*                                         */ \
      1,   /*                                         */ \
      16,  /* One-off with common buffer size         */ \
      32,  /* One-off with common buffer size         */ \
      64,  /* One-off with common buffer size         */ \
      100, /* One-off with common buffer size         */ \
      127                     /* Overflow signed 8-bit when i

#define INTERESTING_8_LEN 9

#define INTERESTING_16                                      \
  -32768,   /* Overflow signed 16-bit when decremented */ \
      -129, /* Overflow signed 8-bit                   */ \
      128,  /* Overflow signed 8-bit                   */ \
      255,  /* Overflow unsig 8-bit when incremented   */ \
      256,  /* Overflow unsig 8-bit                    */ \
      512,  /* One-off with common buffer size         */ \
      1000, /* One-off with common buffer size         */ \
```

config.h

mutation

Deterministic Stages - Dictionary-based Mutations

1. 미리 지정해둔 값으로 대입해본다
2. 실행하며 반복되는 토큰을 탐지하여 dictionary를 동적으로 생성한다

결과가 계속 같으면 skip
세부 로직이 매우 많다.(분석 생략)

mutation

Random / Havoc Stage - Havoc Mutations

1. Flipping bits/bytes
2. Arithmetic changes (add/sub).
3. Inserting blocks of repeated or random bytes
4. Deleting blocks of bytes
5. Overwriting blocks with random data or data from the same input
6. Splicing in content from other queue entries

코드가 너무 방대하여 document 참고

함수 select_algorithm()은 실행할 연산자 그룹을 무작위로 선택

실행 결과를 바탕으로 동적으로 확률을 업데이트