1 一些基本概念

1.1 Basic Block

Basic Block 是满足以下条件的最大长度的序列:

- 只能在 *Basic Block* 的第一条指令进入该 *Basic Block*,
- 只能在 *Basic Block* 的最后一条指令走该 *Basic Block*.

1.2 Control Flow Graph

Control Flow Graph 是通过一下条件构成的:

- *Control Flow Graph* 的节点是 *Basic Block*
- 块 A 到块 B 有一条边当且仅当以下的某个条件成立,
  1. A 的结尾到 B 的开始有一个有条件跳转或者无条件跳转
  2. 在指令序列中 B 块直接跟着 A 块, 且 A 块的结尾不是一个无条件跳转
- 通常将 *jmp label* 指令改成 *jmp blockname*

2 插桩

在《深入理解计算机系统》中，介绍了在 Linux 链接器的库打桩机制，打桩面对的对象是共享库函数。它能够截获对共享库函数的调用，取而代之执行自己的代码。这里的自己的代码可以进行我们想要实现的功能，如追踪某个库函数的调用次数，验证和追踪它的输入和输出值等。下面对库函数中的 malloc 函数进行举例说明。

此处的测试文件为 *test.c*, 其文件内容为,

```
// filename is ./test.c

#include <stdio.h>
#include <malloc.h>
int main(void) {
    int *p = malloc(32);
    free(p);
    return 0;
}
```

对该文件进行编译并运行，使用如下指令，

```
linux> gcc -o test test.c
linux> ./test
linux>
```

可以发现程序没有输出。以下分别对该程序进行编译时打桩，连接时打桩，运行时打桩。

2.1 在编译时打桩

下面对该测试文件进行编译时打桩，

文件 *mymalloc.c*

```
// filename is ./mymalloc.c

#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>
void * mymalloc(size\_t size) {
    void *ptr = malloc(size);
    printf("malloc(\%d) = \%p\n", (int)size, ptr);
    return ptr;
}
#endif
```

文件 *mymalloc.h*

```
// filename is ./mymalloc.h

#define malloc(size) mymalloc(size)
Void *mymalloc(size\_t size);
```

文件 *test.c* 和初始文件一样。对打桩好的文件进行编译连接，

```
linux> gcc -DCOMPILETIME -c mymalloc.c # #1
linux> gcc -I. -o test test.c mymalloc.o # #2
linux> ./test
malloc(32) = 0x933010
linux>
```

在 #1 行中，对 *mymalloc.c* 文件进行编译为可重定位目标文件。在 #2 中，*test.c* 进行编译并与 *mymalloc.c* 链接，此处的*-I.* 用于打桩，它告诉 C 预处理器在搜索通常的系统目录前，先在当前的目录查找 *malloc.h*。

运行*./test*, 得到了我们想要知道的信息。

2.2 链接时打桩

文件 *test.c* 和初始文件一样，没有文件 *malloc.h*。下面给出 *mymalloc.c* 文件。

```
#ifdef LINKTIME
#include <stdio.h>

void *__real_malloc(size_t size);

void *__wrap_malloc(size_t size) {
    void *ptr = __real_malloc(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
#endif
```

在这个程序中，通过 Linux 静态链接器可以将 *__real_malloc* 解析成 *malloc*, 并且在 *test.c* 将 *malloc* 解析成 *__wrap_malloc*。

下面将源文件编译为可重定位目标文件，

```
linux> gcc -DLINKTIME -c mymalloc.c
linux> gcc -c test.c
```

然后将目标文件编译连接为可执行文件，

```
linux> gcc -Wl,--wrap,malloc -o test test.o mymalloc.o
linux> gcc -c test.c
```

其中*-Wl,-wrap,malloc* 就是把*-wrap malloc* 传递给链接器，做出上面所述的解析。

2.3 运行时打桩

运行时打桩是基于动态链接器的 **LD_PRELOAD** 环境变量。当加载和执行一个程序时，需要解析未定义的引用，动态链接器会先搜索 **LD_PRELOAD** 指定的库, 由此，可以对任意共享库中的任何函数进行打桩。

文件 *test.c* 和初始文件一样，没有文件 *malloc.h*。下面给出 *mymalloc.c* 文件。

```
// filename ./mymalloc.c
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

void *malloc(size_t size) {
    void *(*mallocp)(size_t size);
    char *error;
/* Get the address of libc malloc */
    mallocp = dlsym(RTLD_NEXT, "malloc");
    if ((error=dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

编译 *test.c* 文件，

```
linux> gcc -o test test.c
```

构建 *mymalloc.c* 共享库，

```
linux> gcc -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
```

在 *bash shell* 中运行 *test*，

```
linux> LD\_PRELOAD="./mymalloc.so" ./test
malloc(32) = 0x1bf7010
linux>
```

2.4 AFL 中的代码插桩

2.4.1 有源代码代码的情况

当有源代码时，可以对源码进行编译的时候插入代码下面以 *afl-gcc* 为例，进行插桩的指令，

```
fprintf(outf, use_64bit ? trampoline_fmt_64 : trampoline_fmt_32,
    R(MAP_SIZE));
```

这里将汇编代码插入汇编文件中。

其中 R(MAP_SIZE) 是 (random() %(MAP_SEZE))

```
#define MAP_SIZE_POW2       16
#define MAP_SIZE            (1 << MAP_SIZE_POW2)
```

所以 MAP_SIZE 是 2^16，即 64KB

trampoline_fmt_32 具体的汇编代码如下所示。

```
static const u8* trampoline_fmt32 =

  "\n"
  "/* --- AFL TRAMPOLINE (32-BIT) --- */\n"
  "\n"
  ".align 4\n"
  "\n"
  "leal -16(%%esp), %%esp\n"
  "movl %%edi, 0(%%esp)\n"
  "movl \%\%edx, 4(%%esp)\n"
  "movl %%ecx, 8(%%esp)\n"
  "movl %%eax, 12(%%esp)\n"
  "movl \$0x%08x, %%ecx\n"
  "call __afl_maybe_log\n"
  "movl 12(%%esp), %%eax\n"
  "movl 8(%%esp), %%ecx\n"
  "movl 4(%%esp), %%edx\n"
  "movl 0(%%esp), %%edi\n"
  "leal 16(%%esp), %%esp\n"
  "\n"
```

```
  "/* ---- END ---- */\n"
  "\n";
```

"movl $0x%08x, %%ecx/n", R(MAP_SIZE),R(MAP_SIZE) 获得 0 到 MAP_SIZE 之间的一个随机数, 这个随机数用来标识这个代码块的 key。

这里 trampoline_fmt_32 实际上是为了调用 *___afl_maybe+_log* 这个函数。

这个 *call* 函数插入的位置在每个代码块之间。

3 突变 Mutation Engine

3.1 文件存放在队列中

3.1.1 和队列相关的一些数据结构

队列中的每一元素的数据结构如下,

```
struct queue_entry {
  u8* fname; /* 文件名字 */
  u32 len; /* 文件的长度 *//* 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,   \\
      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 */
};
```

队列使用链表表示,

```
static struct queue_entry
        *queue,      /* Fuzzing queue (linked list) */
        *queue_cur, /* Current offset within the queue  */
        *queue_top, /* Top of the list       */
        *q_prev100; /* Previous 100 marker  */
```

3.2 变异方法

在 *fuzz_one* 函数中实现了各种的变异方法

3.2.1 bitflip

bitflip 的实现函数

实现方法在 *FILP_BIT(_ar, _b)* 宏定义中，其函数的定义如下所示

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

_bf 表示当前的位,*_bf >>* 3 表示当前的字节,*_bf*&7 当前字节内位号,(128 >> (*_bf*&7)) 表示字节内的某一位置 1，= 表示字节内的翻转。

bitflip 1/1

其中的 1/1 中的 *1*/1 表示翻转 1 个比特，1/*1* 表示步长为 1。具体实现如下

```
for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
    ...
    FLIP_BIT(out_buf, stage_cur);
    ...


}
```

bitflip 2/1

其中的 2/1 中的 *2*/1 表示翻转相邻的 2 个比特，2/*1* 表示步长为 1。具体实现如下

```
for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
    ...
    FLIP_BIT(out_buf, stage_cur);
    FLIP_BIT(out_buf, stage_cur + 1);
    ...
}
```

bitflip 4/1

其中的 4/1 中的 *4*/1 表示翻转相邻的 4 个比特，4/*1* 表示步长为 1。具体实现如下

```
for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
    ...
    FLIP_BIT(out_buf, stage_cur);
    FLIP_BIT(out_buf, stage_cur + 1);
    FLIP_BIT(out_buf, stage_cur + 2);
    FLIP_BIT(out_buf, stage_cur + 3);
    ...
}
```

bitflip 8/8

其中的 8/8 中的 *8*/8 表示翻转相邻的 8 个比特，8/*8* 表示步长为 8。具体实现如下

```
for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
    ...
    out_buf[stage_cur] ^= 0xFF;
    ...
}
```

注意这里的 stage_cur 表示当前的字节为，而之前的 stage_cur 表示当前的比特位。

bitflip 16/8

其中的 16/8 中的 *16*/8 表示翻转相邻的 16 个比特，16/*8* 表示步长为 8。具体实现如下

```
for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
```

```
    ...
    *(u16*)(out_buf + stage_cur) ^= 0xFFFF;
    ...
}
```

这里的 *stage_cur* 表示当前的字节。

bitflip 32/8

其中的 32/8 中的 *32*/8 表示翻转相邻的 32 个比特，32/*8* 表示步长为 8。具体实现如下

```
for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
    ...
    *(u32*)(out_buf + stage_cur) ^= 0xFFFFFFFF
    ...
}
```

这里的 *stage_cur* 表示当前的字节。

3.2.2 arithmetic

arth 8/8

其中的 8/8 中的 *8*/8 表示翻转相邻的 32 个比特，32/*8* 表示步长为 8。具体实现如下

```
for (i = 0; i < len; i++) {
    u8 orig = out_buf[i];
    ...
    for (j = 1; j <= ARITH_MAX; j++) {
        ...
        out_buf[i] = orig + j;
        ...
        out_buf[i] = orig - j;
    }
    ...
}
```

表示对原来的位进行加减 j 运算。注意如果之前 bitflip 已经生成过的变异：如果加/减某个数后，其效果与之前的某种 bitflip 相同，那么这次变异肯定在上一个阶段已经执行过了，此次便不会再执行。

类似的有 **arth 16/8, arth 32/8, arth**。这里不在赘述。

3.2.3 interest

interest 表示把一些特殊内容替换到原文件中。

interest 8/8

每次对 8 个 bit 进替换，按照每 8 个 bit 的步长从头开始，即对文件的每个 byte 进行替换

具体如何实现，略。

interest 16/8

每次对 16 个 bit 进替换，按照每 8 个 bit 的步长从头开始，即对文件的每个 word 进行替换

具体如何实现，略。

interest 32/8

每次对 32 个 bit 进替换，按照每 8 个 bit 的步长从头开始，即对文件的每个 dword 进行替换

具体如何实现，略。

其中 interest value 的值在 configh 已经设定好。可以看到，用于替换的基本都是可能会造成溢出的数；

3.2.4 dictionary

dictionary 表示把自动生成或用户提供的 token 替换/插入到原文件中。

user extras(over)

从头开始，将用户提供的 tokens 依次替换到原文件中。

具体如何实现，略。

user extras(insert)

从头开始，将用户提供的 tokens 依次插入到原文件中。

具体如何实现，略。

auto extras(over)

从头开始，将自动检测的 tokens 依次替换到原文件中。

3.2.5 havoc

有以下一些思路，关于如何具体实现，此处略。

- 随机选取某个 bit 进行翻转

- 随机选取某个 byte，将其设置为随机的 interesting value
- 随机选取某个 word，并随机选取大、小端序，将其设置为随机的 interesting value
- 随机选取某个 dword，并随机选取大、小端序，将其设置为随机的 interesting value
- 随机选取某个 byte，对其减去一个随机数
- 随机选取某个 byte，对其加上一个随机数
- 随机选取某个 word，并随机选取大、小端序，对其减去一个随机数
- 随机选取某个 word，并随机选取大、小端序，对其加上一个随机数
- 随机选取某个 dword，并随机选取大、小端序，对其减去一个随机数
- 随机选取某个 dword，并随机选取大、小端序，对其加上一个随机数
- 随机选取某个 byte，将其设置为随机数
- 随机删除一段 bytes
- 随机选取一个位置，插入一段随机长度的内容，其中 75% 的概率是插入原文中随机位置的内容，25% 的概率是插入一段随机选取的数
- 随机选取一个位置，替换为一段随机长度的内容，其中 75% 的概率是替换成原文中随机位置的内容，25% 的概率是替换成一段随机选取的数
- 随机选取一个位置，用随机选取的 token（用户提供的或自动生成的）替换
- 随机选取一个位置，用随机选取的 token（用户提供的或自动生成的）插入

3.2.6 splice

拼接，2 个 seed 进行拼接，并进行 havoc, 具体如何实现此处忽略。

4 fork server

相比 *fuzzer* 进程，目标程序以不同的进程运行着，他们相互隔离，因此，类似目标程序的错误导致进程崩溃不会影响 *fuzzer* 程序的运行。

由于 *execve(), linker, all of the library initialization* 需要消耗一定的时间，如何能够最小化花费在这些工作上的时间，*Jann Horn* 给出了一个很好的方案, 他归结为将一小段代码

注入到模糊的二进制文件中,这一壮举可以通过 LD_PRELOAD,PTRACE_POKETEXT,编译时工具-或只是通过提前重写 ELF 二进制文件

injected shim 的目的是让 execve () 发生，经过链接器，然后在实际程序中尽早停止，然后再处理模糊器生成的任何输入或进行其他感兴趣的操作。

参考

https://thepatrickstar.github.io/afl-white-paper/
https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html
http://rk700.github.io/2018/01/04/afl-mutations/
https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html
https://bbs.pediy.com/thread-254705.htm
http://rk700.github.io/2017/12/28/afl-internals/
https://paper.seebug.org/841/
https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html