AFL源码分析笔记(一)

zoniony / 2019-04-05 08:39:00 / 浏览数 4906 安全技术 二进制安全 顶(0) 踩(0)

0x00 前言

总结师傅们笔记，主要源码分析。

0x01 代码覆盖率

代码覆盖率是fuzz中基本概念，先了解清这个概念后面的插装编译等概念才好理解。

代码覆盖率是一种度量代码的覆盖程度的方式，也就是指源代码中的某行代码是否已执行；对二进制程序，还可将此概念理解为汇编代码中的某条指令是否已执行。对fuzz来

计量方式主要为三种：函数，基本块，边界

插桩

插桩是为了覆盖率而实行的方法。

afl-gcc.c

afl-gcc是gcc的一个封装(wrapper)

主要三个功能

```
find_as(argv[0]);                    //■■gcc/clang/llvm■■■
 edit_params(argc, argv);            //■■■■
 execvp(cc_params[0], (char**)cc_params);//■■
```

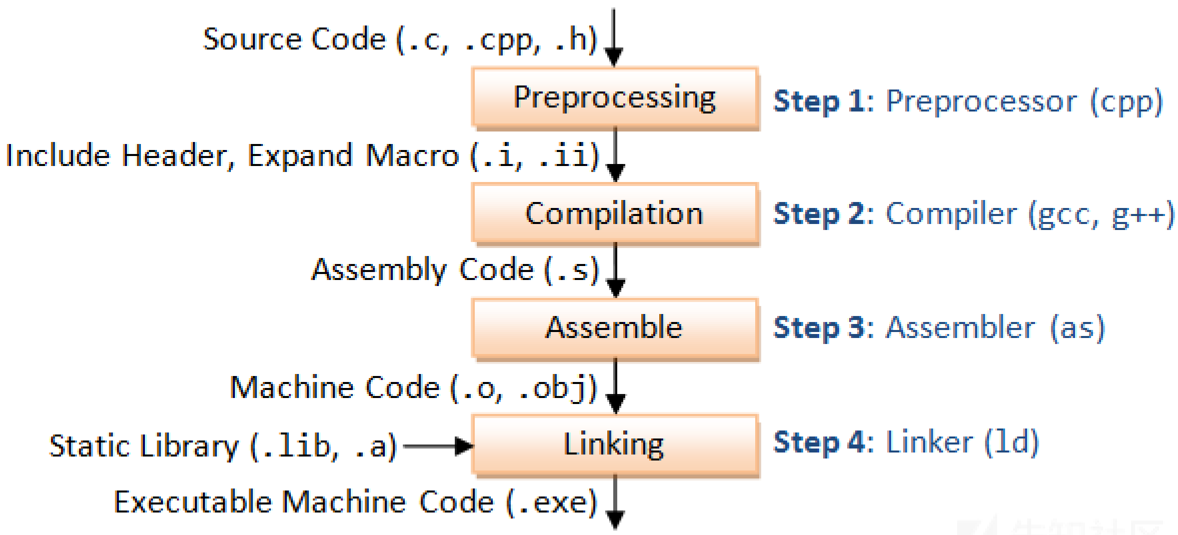
打印出cc_params,看看真正的参数是什么

```
gcc -o test test.c -B /usr/local/lib/afl -g -O3 -funroll-loops -D__AFL_COMPILER=1 -DFUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION=1
```

看看参数的意思。用了编译优化，指定了编译的标志，最终要的是-B指定了编译器(Assembler)

```
-funroll-loops      ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
-B <■■>             ■ <■■> ■■■■■■■■■■■■■
#ifdef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION     /* a flag also shared with libfuzzer) or */
#ifdef __AFL_COMPILER                                /* (this one is just for AFL). */
```

这一步正是汇编文件通过as进一步编译成二进制文件，这里替换了Assembler，当然为了插桩



afl-as.c和afl-as.h

反汇编刚才生成的test,会发现插了一些插入了额外的汇编指令

```
0x5555555547e0 <main>       lea    rsp, [rsp - 0x98]
0x5555555547e8 <main+8>     mov    qword ptr [rsp], rdx
0x5555555547ec <main+12>    mov    qword ptr [rsp + 8], rcx
0x5555555547f1 <main+17>    mov    qword ptr [rsp + 0x10], rax
0x5555555547f6 <main+22>    mov    rcx, 0x60b
0x5555555547fd <main+29>    call   __afl_maybe_log <0x555555554950>

0x555555554802 <main+34>    mov    rax, qword ptr [rsp + 0x10]
0x555555554807 <main+39>    mov    rcx, qword ptr [rsp + 8]
0x55555555480c <main+44>    mov    rdx, qword ptr [rsp]
0x555555554810 <main+48>    lea    rsp, [rsp + 0x98]
0x555555554818 <main+56>    lea    rsi, [rip + 0x5b5]
```

这两个文件被单独提出来可以来解释这里是怎么操作的

> The sole purpose of this wrapper is to preprocess assembly files generated by GCC / clang and inject the instrumentation bits included from afl-as.h. It is automatically invoked by the toolchain when compiling programs using afl-gcc / afl-clang.

主要是处理不同平台设置标志，处理参数等等.重要函数`add_instrumentation`

fprintf将插桩用的汇编用`fprintf`插如合适的地方

```
static  void  add_instrumentation■void■{
   ......
   while (fgets(line, MAX_LINE, inf)) {//■■■■
   ......
     fprintf(outf, use_64bit ? trampoline_fmt_64 : trampoline_fmt_32,
             R(MAP_SIZE));//■■■■■R(MAP_SIZE)
   ......
       //■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
   }
}
```

下面分别是32位和64位的，和调试看的一样

```
static const u8* trampoline_fmt_32 =

 "\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"       //■■__afl_maybe_log
 "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";

static const u8* trampoline_fmt_64 =

 "\n"
 "/* --- AFL TRAMPOLINE (64-BIT) --- */\n"
 "\n"
 ".align 4\n"
 "\n"
 "leaq -(128+24)(%%rsp), %%rsp\n"
 "movq %%rdx,  0(%%rsp)\n"
```

```
 "movq %%rcx,  8(%%rsp)\n"
 "movq %%rax, 16(%%rsp)\n"
 "movq $0x%08x, %%rcx\n"
 "call __afl_maybe_log\n"
 "movq 16(%%rsp), %%rax\n"
 "movq  8(%%rsp), %%rcx\n"
 "movq  0(%%rsp), %%rdx\n"
 "leaq (128+24)(%%rsp), %%rsp\n"
 "\n"
 "/* --- END --- */\n"
 "\n";
```

所以能看到，插桩是为了统计覆盖率。至于具体怎么实现，继续看后面

fork service

这是一种为了不使用`execve()`函数提高效率想出来的办法，省掉动态链接等过程，在lcamtuf的blog上也有详细的介绍。

afl-fuzz.c

```
EXP_ST void init_forkserver(char** argv) {

 int st_pipe[2], ctl_pipe[2];//■■■■■■■■■
   ......

execv(target_path, argv);   //■■fork server
}
```

有两个重点

- 怎么高效重复执行测试样例
- 记录样例的状态

开始fork service确认创建完毕

```
/* Close the unneeded endpoints. */
//■■■■■■■■
 close(ctl_pipe[0]);
 close(st_pipe[1]);

//■■■■■■■■
 fsrv_ctl_fd = ctl_pipe[1];
 fsrv_st_fd  = st_pipe[0];

......

 rlen = read(fsrv_st_fd, &status, 4);//■■■■■■■4■■■


 /* If we have a four-byte "hello" message from the server, we're all set.
    Otherwise, try to figure out what went wrong. */

 if (rlen == 4) {//■■■■■■■■
   OKF("All right - fork server is up.");
   return;
 }
```

__afl_maybe_log()

这里因为AFL自带的延时检测，所以没法调试看，这里只有看源码

这里先检测是否分配到公共内存，`__afl_area_ptr`里面就是地址，否则先调用`__afl_setup`初始化

```
.text:0000000000000950                 lahf
.text:0000000000000951                 seto    al
.text:0000000000000954                 mov     rdx, cs:__afl_area_ptr
.text:000000000000095B                 test    rdx, rdx
.text:000000000000095E                 jz      short __afl_setup
```

__afl_forkserver

写4个字节到状态管道`st_pipe[0]`，forkserver告诉fuzzer自己准备好了，而这正好是`rlen = read(fsrv_st_fd, &status, 4);`中等待的信息

```
.text:0000000000000ABB __afl_forkserver:
.text:0000000000000ABB                 push    rdx
.text:0000000000000ABC                 push    rdx
.text:0000000000000ABD                 mov     rdx, 4          ; n
.text:0000000000000AC4                 lea     rsi, __afl_temp ; buf
.text:0000000000000ACB                 mov     rdi, 0C7h       ; fd
.text:0000000000000AD2                 call    _write
.text:0000000000000AD7                 cmp     rax, 4
.text:0000000000000ADB                 jnz     __afl_fork_resume
```

__afl_fork_wait_loop

`fork server`直到从状态管道`read`到4个字节表明`fuzzer`准备好了

```
text:0000000000000AE1                  mov     rdx, 4          ; nbytes
.text:0000000000000AE8                 lea     rsi, __afl_temp ; buf
.text:0000000000000AEF                 mov     rdi, 0C6h       ; status
.text:0000000000000AF6                 call    _read
.text:0000000000000AFB                 cmp     rax, 4
.text:0000000000000AFF                 jnz     __afl_die
.text:0000000000000B05                 call    _fork
.text:0000000000000B0A                 cmp     rax, 0
.text:0000000000000B0E                 jl      __afl_die
.text:0000000000000B14                 jz      short __afl_fork_resume
```

记录子进程的pid，一旦子进程执行完了，通过状态管道发送到`fuzzer`继续执行

```
.text:0000000000000B16                 mov     cs:__afl_fork_pid, eax
.text:0000000000000B1C                 mov     rdx, 4          ; n
.text:0000000000000B23                 lea     rsi, __afl_fork_pid ; buf
.text:0000000000000B2A                 mov     rdi, 0C7h       ; fd
.text:0000000000000B31                 call    _write
.text:0000000000000B36                 mov     rdx, 0          ; options
.text:0000000000000B3D                 lea     rsi, __afl_temp ; stat_loc
.text:0000000000000B44                 mov     rdi, qword ptr cs:__afl_fork_pid ; pid
.text:0000000000000B4B                 call    _waitpid
.text:0000000000000B50                 cmp     rax, 0
.text:0000000000000B54                 jle     __afl_die
.text:0000000000000B5A                 mov     rdx, 4          ; n
.text:0000000000000B61                 lea     rsi, __afl_temp ; buf
.text:0000000000000B68                 mov     rdi, 0C7h       ; fd
.text:0000000000000B6F                 call    _write
.text:0000000000000B74                 jmp     __afl_fork_wait_loop
```

用伪代码更能看清楚逻辑

```
if ( write(0xC7, &_afl_temp, 4uLL) == 4 )
     {
       while ( 1 )
       {
         v25 = 0xC6;
         if ( read(0xC6, &_afl_temp, 4uLL) != 4 )
           break;
         LODWORD(v26) = fork();
         if ( v26 < 0 )
           break;
         if ( !v26 )
           goto __afl_fork_resume;
         _afl_fork_pid = v26;
         write(0xC7, &_afl_fork_pid, 4uLL);
         v25 = _afl_fork_pid;
         LODWORD(v27) = waitpid(_afl_fork_pid, &_afl_temp, 0);
         if ( v27 <= 0 )
           break;
         write(199, &_afl_temp, 4uLL);
       }
```

```
      _exit(v25);
    }
```

在fuzzer这边来看,发出请求，接受状态，根据状态管道判断执行结果......

```
if ((res = write(fsrv_ctl_fd, &prev_timed_out, 4)) != 4); //■■fork server
if ((res = read(fsrv_st_fd, &child_pid, 4)) != 4)
   .......
/* Report outcome to caller. */
 if (WIFSIGNALED(status) && !stop_soon) {
   kill_signal = WTERMSIG(status);
   if (child_timed_out && kill_signal == SIGKILL) return FAULT_TMOUT;
   return FAULT_CRASH;

 }
```

分支记录

如何判断这条路径（代码）执行过，后面还要根据这些记录对后面变异有帮助。既要节约空间又要有效率，那单链表之类的肯定不能用,AFL用的是二元tuple(跳转的源地址和

例如：

  A->B->C->D->A-B

可以用[A,B] [B,C] [C,D] [D,A]四个二元组表示，只需要记录跳转的源地址和目标地址。并且[A,B]执行了两次，其余执行了一次，这里用hash映射在一张map中。

接下来代码具体讲讲。

之前在`__afl_maybe_log`后面还有`_afl_store`这个函数

```
.text:0000000000000960 __afl_store:                            ; CODE XREF: __afl_maybe_log+4F↓j
.text:0000000000000960                                         ; __afl_maybe_log+309↓j
.text:0000000000000960                 xor     rcx, cs:__afl_prev_loc
.text:0000000000000967                 xor     cs:__afl_prev_loc, rcx
.text:000000000000096E                 shr     cs:__afl_prev_loc, 1
.text:0000000000000975                 inc     byte ptr [rdx+rcx]
```

对应的伪代码。`COMPILE_TIME_RANDOM`就是`add_instrumentation`中`fprintf`中`R(MAP_SIZE)`,也是在执行`call __afl_maybe_log`汇编前`rcx`中保存的随机数，这个随机数代表分支

```
cur_location = <COMPILE_TIME_RANDOM>;       //■■■■■■■
shared_mem[cur_location ^ prev_location]++;  //■■■■■■■■■■■■■■■■■■■■■■tuple■■map
prev_location = cur_location >> 1;          //■■■■■■■■
```

为什么当前分支最后需要向右移一位？比如`A->A`或者`A->B->A`这种不右移异或为0

并且共享内存的`MAP_SIZE=64K`碰撞概率缩小很多。下面是官方给的

```
Branch cnt | Colliding tuples | Example targets
------------+------------------+-----------------
     1,000 | 0.75%            | giflib, lzo
     2,000 | 1.5%             | zlib, tar, xz
     5,000 | 3.5%             | libpng, libwebp
    10,000 | 7%               | libxml
    20,000 | 14%              | sqlite
    50,000 | 30%              | -
```

分支信息处理

共享内存还有个变量`trace_bits`来记录分支执行次数

```
classify_counts((u32*)trace_bits);
```

fuzzer主要将每个分支处理次数归入下面这个表中

```
static const u8 count_class_lookup8[256] = {

 [0]           = 0,
 [1]           = 1,
 [2]           = 2,
```

```
 [3]           = 4,
 [4 ... 7]     = 8,
 [8 ... 15]    = 16,
 [16 ... 31]   = 32,
 [32 ... 127]  = 64,
 [128 ... 255] = 128
};
```

比如执行了4-7次的其计数为8，最后用一个hash还判断新测试用例分支数增加没有

```
u32 cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);
```

参考链接

https://paper.seebug.org/496/
http://lcamtuf.coredump.cx/afl/technical_details.txt
https://www.inforsec.org/wp/?p=2678
https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html

上一篇：通过三道CTF学习反馈移位寄存器 下一篇：通过三道CTF学习反馈移位寄存器