fuzz系列之afl

afl 实战

前言

像 libFuzzer, afl 这类 fuzz 对于 从文件 或者 标准输入 获取输入的程序都能进行很好的 fuzz, 但是对于基于网络的程序来说就不是那么方便了。

这篇文章介绍用 afl 来 fuzz 网络应用程序。

介绍

afl 是一个非常厉害的 fuzz，最近几年炒的火热。它是基于代码插桩来生成测试用例，这样生成的样本就比较的好，而且针对 linux 做了许多性能优化使得速度也非常快。

使用 afl 的常规步骤

- 如果有源码，用 afl-gcc 或者 afl-clang-fast 编译源码，afl 会利用这些工具在编译期间对代码进行插桩，为后面的测试提供代码覆盖率，测试样本的变异则会基于代码覆盖率进行。无源码的话可以使用 qemu 进行插桩
- 搜集好 **初始样本集**，如果必要的话使用 afl-cmin 把样本集进行**精简**。
- 然后用 afl-fuzz 开始 fuzz.

下面对一些常见的命令给个示例

精简样本集

```
afl-cmin -i in/ -o out/  /path/to/program
```

in/ 是初始样本集目录

out/ 是 精简后的样本集存放的目录

开启 fuzz

```
afl-fuzz -i in/ -o out/ /path/to/program
```

in/ 是初始样本集目录

out/ 用于保存 fuzz 过程中的一些文件

afl-fuzz 默认是往 stdin 中写测试数据，它同时支持从文件喂 测试数据给目标程序，只要把设置文件的参数修改为 @@ ， fuzz 过程中 afl-fuzz 会把它替换成 文件名。

比如 ./a 这个程序的 第二个参数是要处理的文件的名称， 那么相应的 afl-fuzz 的命令就是

```
afl-fuzz -i in/ -o out/ ./a arg1 @@
```

更多内容请看

http://lcamtuf.coredump.cx/afl/QuickStartGuide.txt

http://lcamtuf.coredump.cx/afl/README.txt

Fuzz 网络程序

这里以 libmodbus 这个库为目标进行 fuzz 。

构建 Modbus TCP Server

库的官网地址如下

http://libmodbus.org/documentation/

这是一个用于 modbus 通讯的库， 通过这个库可以很方便的实现 modbus 服务器 和 客户端。 这里以 modbus tcp 的服务端作为 fuzz 的对象。

首先在官网下载好源码

http://libmodbus.org/releases/libmodbus-3.1.4.tar.gz

源码目录下的 tests 目录里面有一些示例程序， 其中 tests/bandwidth-server-one.c就实现了一个 modbus tcp server ， 把它做一些精简得到

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <modbus.h>

int main(int argc, char *argv[])
{
    int s = -1;
    modbus_t *ctx = NULL;
    modbus_mapping_t *mb_mapping = NULL;
    int rc;
    int use_backend;
    ctx = modbus_new_tcp("127.0.0.1", 1502);
    s = modbus_tcp_listen(ctx, 1);
    modbus_tcp_accept(ctx, &s);
    mb_mapping = modbus_mapping_new(MODBUS_MAX_READ_BITS, 0,
                                    MODBUS_MAX_READ_REGISTERS, 0);
    if (mb_mapping == NULL) {
        modbus_free(ctx);
        return -1;
    }
    uint8_t query[MODBUS_TCP_MAX_ADU_LENGTH];
    memset(query, 0, MODBUS_TCP_MAX_ADU_LENGTH);
    rc = modbus_receive(ctx, query); // 获取客户端的请求数据
    if (rc > 0) {
        modbus_reply(ctx, query, rc, mb_mapping); // 处理并响应之
    }
    modbus_mapping_free(mb_mapping);
    if (s != -1) {
        close(s);
    }
    /* For RTU, skipped by TCP (no TCP connect) */
    modbus_close(ctx);
    modbus_free(ctx);
    return 0;
}
```

代码逻辑简单理一下

- modbus_new_tcp 初始化 modbus_t 结构体
- modbus_tcp_accept 和 modbus_tcp_listen 就是调用 socket 监听端口
- modbus_mapping_new 初始化一个缓冲区，用于模拟寄存器信息
- 然后 modbus_receive 接收客户端的请求和输入
- 获取输入后就 通过 modbus_reply 处理 请求，以及构造响应数据包， 同时返回响应
- 然后就是释放掉分配的一些内存

利用 preeny 库辅助

afl 默认只能 fuzz 通过 stdin 和 文件 获取输入的程序， 要 fuzz 网络相关的程序，需要使用一个库

https://github.com/zardus/preeny

这个库利用 LD_PRELOAD 机制，重写了 很多库函数， 其中 desock.c 这个文件负责重写 socket 相关的函数，其实现的功能就是当应用从 socket 获取输入时，其实是从 stdin 获取输入。 （具体实现以后再看~_~）

首先下载编译下

```
git clone https://github.com/zardus/preeny.git
cd preeny/
make
```

然后会在 x86_64-linux-gnu 目录下生成编译好的 lib 。

写个测试脚本，测试一下 (根据 tests 目录里面的 sock.c 改造)

```
#include <sys/socket.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int s = socket(AF_INET, SOCK_STREAM, 0);
    char buf[1024]={0};
    char send_msg[] = "hello, send by send() :\n";
    send(s, send_msg, strlen(send_msg), 0);
    recv(s, buf, 1024, 0);
    printf("recv from recv() : %s\n", buf);
}
```

编译运行

```
gcc sock_test.c -o sock_test
LD_PRELOAD="/home/haclh/vmdk_kernel/preeny/x86_64-linux-gnu/desock.so" ./sock_test
```

```
03:29 haclh@ubuntu:preeny $ LD_PRELOAD="/home/haclh/vmdk_kernel/preeny/x86_64-linux-gnu/desock.so" ./sock_test
hello, send by send() :
I am Tester
recv from recv() : I am Tester
```

往 socket 调用 send , 成功往 stdout 输出了 字符串。

从 stdin 输入 I am Tester ， 可以看到成功写入 buf 里面

所以我们就可以利用 preeny 来 fuzz modbus tcp server 了

编译 modbus server

首先使用 afl-gcc 编译 libmodbus ， 对 libmodbus 插桩。

```
unzip libmodbus-master.zip
cd libmodbus-master/
./autogen.sh
CC=afl-gcc CXX=afl-g++ ./configure --enable-static
make -j4
```

--enable-static： 用于生成静态库

然后在 src/.libs 下就可以看到编译好的库

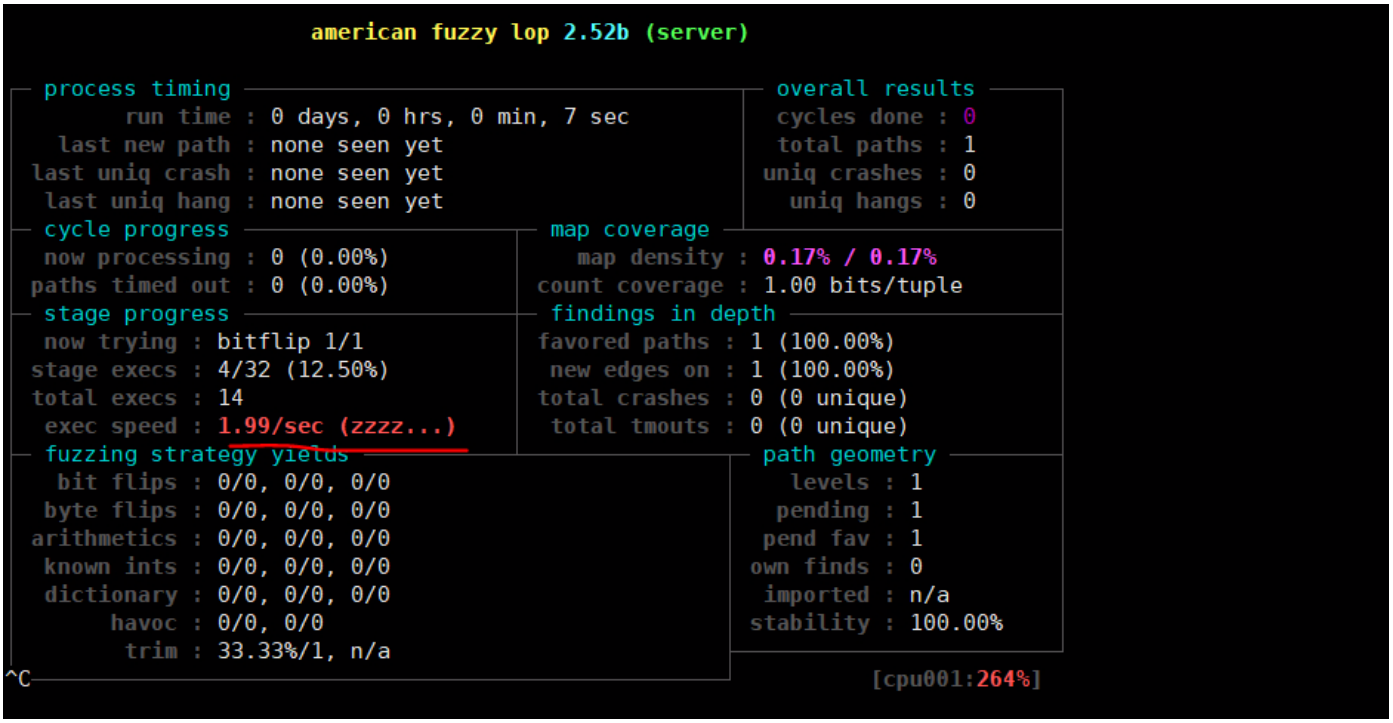
```
03:45 haclh@ubuntu:libmodbus-master $ ls src/.libs/
libmodbus.a  libmodbus.la  libmodbus.lai  libmodbus.so  libmodbus.so.5  libmodbus.so.5.1.0  modbus-data.o  modbus.o  modbus-rtu.o  modbus-tcp.o
```

libmodbus.a 就是编译好的静态库

然后使用我们修改过的 bandwidth-server-one.c 编译 和 fuzz

```
cd tests/
vim bandwidth-server-one.c
afl-gcc bandwidth-server-one.c -I../src ../src/.libs/libmodbus.a -o server
mkdir in
echo 11111 > in/1
LD_PRELOAD="/home/haclh/vmdk_kernel/preeny/x86_64-linux-gnu/desock.so" afl-fuzz -i in -o out ./server
```

这里 直接用 `echo` 生成了一个 测试文件，如果直接用这个去测的话会发现速度非常的慢。



获取样本数据

一组好的样本数据对 `fuzzer` 的影响还是非常大的，一般我们可以去网上搜索样本，比如图片，视频文件等。对于我们这次的目标 `libmodbus` ， 它自带了很多的测试程序，我们可以利用这些测试程序测试，然后用 `tcpdump` 抓包， 最后在把其中的请求数据保存下来，作为测试样本集。

首先使用 random-test-server 在 127.0.0.1:1502 起一个 modbus tcp 服务

```
04:09 haclh@ubuntu:libmodbus-master $ cd tests/
04:09 haclh@ubuntu:tests $ ./random-test-server
```

然后开启 tcpdump， 保存数据包到 ~/modbus.pcap

```
04:09 haclh@ubuntu:~ $ sudo tcpdump -i lo -w ~/modbus.pcap
[sudo] password for haclh:
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
```

最后使用 `random-test-client` 随机发送各种 `modbus` 请求到 `127.0.0.1:1502`

```
cd tests/
./random-test-client
```

然后写一个脚本把 `~/modbus.pcap` 中 由客户端发送的数据包 （也就是目的地为 `127.0.0.1:1502`的数据包） 的内容提取出来，每个数据包内容保存为一个单独的文件。

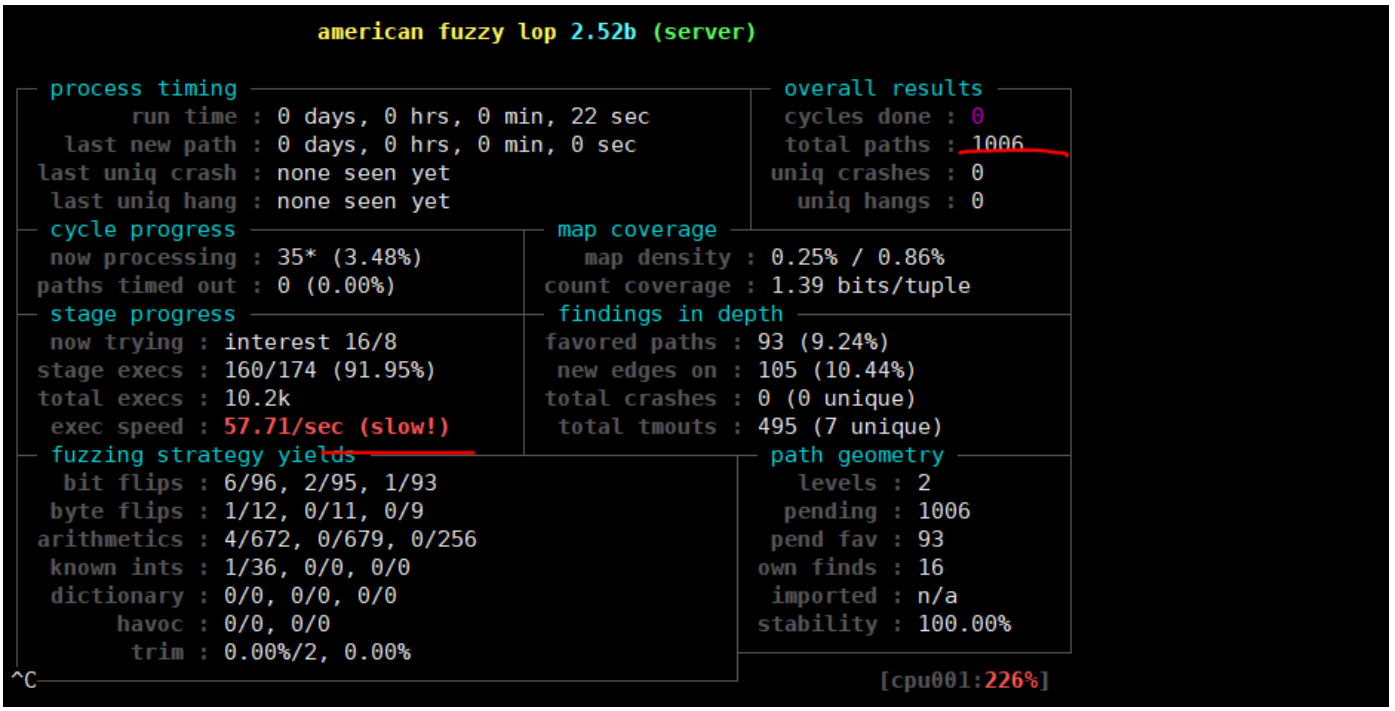
```
from scapy.all import *
save_path = "/tmp/seeds/"
uuid = 0

if not os.path.exists(save_path):
    os.system("mkdir %s" %(save_path))

def save_to_file(data):
```

```
    global uuid
    with open("{}{}".format(save_path, uuid), "w") as fp:
        fp.write(str(data))
    uuid += 1
    print "write test file: {}".format(uuid)

modbus_session = ''
pg = rdpcap("modbus.pcap")
session = pg.sessions()
for k in session.keys():
    if k.endswith("127.0.0.1:1502"):
        modbus_session = session[k]

for s in modbus_session:
    payload = s[TCP].payload
    if len(payload) > 4:
        save_to_file(payload)

print "Total: %d tests" %(uuid)
```

使用获取的样本再次 fuzz

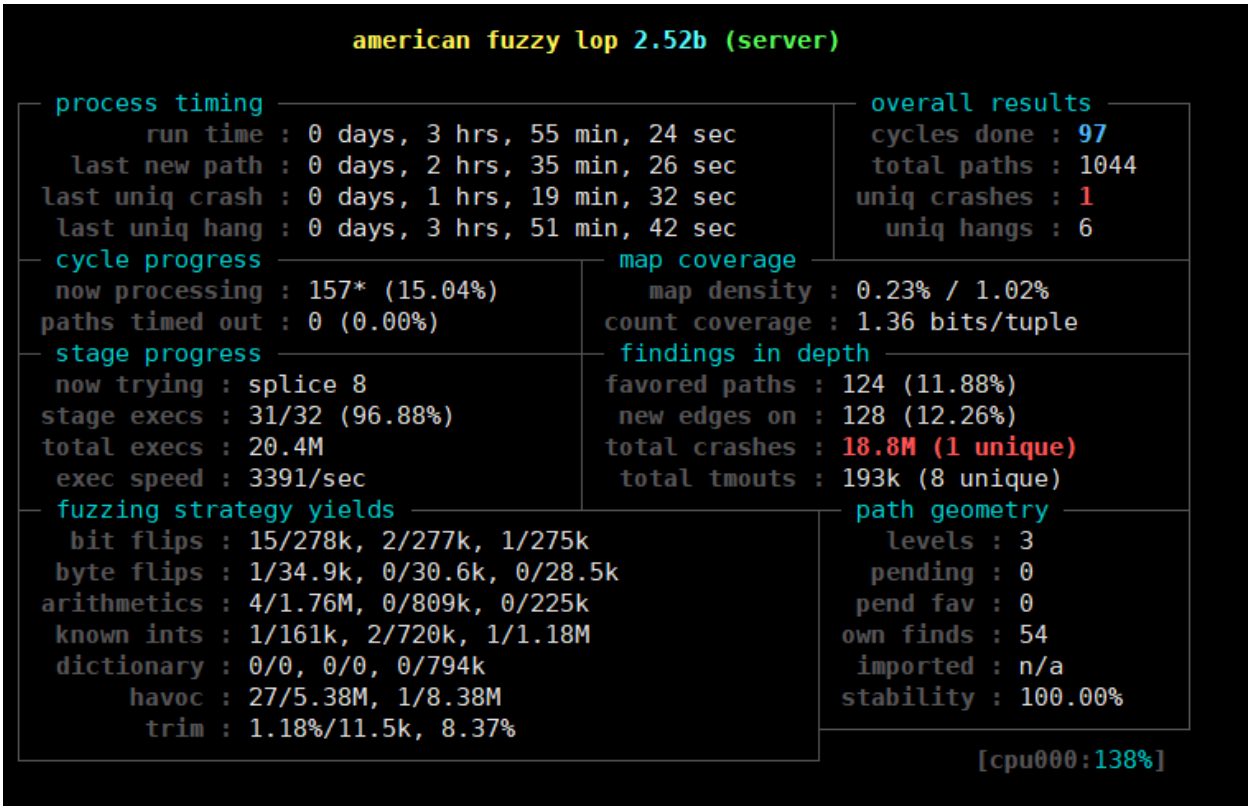
然后以生成的样本集作为初始样本集进行 `fuzz`

```
LD_PRELOAD="/home/haclh/vmdk_kernel/preeny/x86_64-linux-gnu/desock.so" afl-fuzz -i /tmp/seeds/  -o out ./server
```



速度有一定的提升，而且 总路径数 直接 `1000+`

最后 `fuzz` 了两个多小时



那个唯一的 `crash` 还是误报 (~_~)

总结

`afl + preeny` 来 `fuzz` 网络应用 速度还行， 关键的还是要找到好的样本，从程序自带的测试用例中抓取也是一个不错的思路。

参考

Fuzzing nginx - Hunting vulnerabilities with afl-fuzz

AFL Persistent Mode

在介绍一个 在 `fuzz` 一些网络程序时可能用到的特性， `AFL` 的 `persistent` 模式。

`persistent` 模式就是在程序的某个代码位置不断喂**生成的变异数据** 进行 `fuzz`，而不用每次喂数据都得重新 `fork` 一个程序。

要使用这个特性，首先得编译 `llvm_mode`

```
cd afl-2.52b/
cd llvm_mode/
make
cd ..
sudo make install
```

此时就会有 `afl-clang-fast` 和 `afl-clang-fast++` 两个命令， 要使用这个模式，就要用这两个命令来编译目标应用。

`afl` 的 作者有一篇 文章 介绍这个特性。

下面还是用 `afl` 自带的 测试文件 `experimental/persistent_demo/persistent_demo.c` 来看看。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
int main(int argc, char** argv) {
  char buf[100];
  while (__AFL_LOOP(1000)) {
    memset(buf, 0, 100);
```

```
    read(0, buf, 100);
    if (buf[0] == 'f') {
      printf("one\n");
      if (buf[1] == 'o') {
        printf("two\n");
        if (buf[2] == 'o') {
          printf("three\n");
          if (buf[3] == '!') {
            printf("four\n");
            abort();
          }
        }
      }
    }
  } // end of while (__AFL_LOOP(1000))
  return 0;
}
```

最关键的 就是 AFL_LOOP(1000) 这个宏,其中的参数指定**循环的次数。**

每一次循环 afl 都会生成 测试数据，然后喂到 stdin ,这样 fuzzer 就可以在 AFL_LOOP 宏 包围的内部，通过 read(0,buf, size) 来获取测试数据，然后喂给目标程序的数据处理的代码，这样可以减少 fork 等操作的开销。

对应到上面的程序，就是 afl 会在

```
while (__AFL_LOOP(1000)) {
........
........
}
```

里面 fuzz 1000 次，即生成 1000 次测试数据， 然后会 return 0 . 程序结束，然后 afl 会重新起一个程序。继续这样的 fuzz .

被 while (__AFL_LOOP(1000)) 包围的代码，就是不断的 从 stdin 获取测试数据，然后进入下面的 if 判断逻辑。

编译 然后用 afl-fuzz 它

```
afl-clang-fast persistent_demo.c -o persistent_demo
afl-fuzz  -i in/ -o out/ ./persistent_demo
```

一会就能拿到 crash 了。（ abort 会被 afl 检测为 crash）。

对于 libmodbus ， 不会用这种方式进行 fuzz。（如果有人成功，望悉知，感激不尽）

不过网上还是有一些案例

```
https://www.fastly.com/blog/how-fuzz-server-american-fuzzy-lop
https://sensepost.com/blog/2017/fuzzing-apache-httpd-server-with-american-fuzzy-lop-%2B-persistent-mode/
```

总结

如果使用 afl 来 fuzz 网络应用，有两种方式

- 利用 preeny 把从 socket 获取数据，转变为 从 stdin 获取数据
- 利用 afl 的 persistent 模式

其实还有第三种，网上有个修改版的 afl 可以用来 fuzz 网络应用，不过版本比较老，貌似也没啥人使用（:~

```
https://github.com/jdbirdwell/afl
```

此外， afl 还有各种扩展模式，比如 利用 qemu 可以无源码 fuzz。 17 年 还有一个 afl-unicorn ， 貌似可以 fuzz 任意架构的代码（:没来得及看~。

```
https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf
https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5
```

来源： https://www.cnblogs.com/hac425/p/9416917.html