linux 符号执行,Manticore：符号执行

转载

深井冰323 于 2021-05-16 20:50:10 发布 140 收藏 1
文章标签： linux 符号执行

“感谢打赏”

8种机械键盘轴体对比

本人程序员，要买一个写代码的键盘，请问红轴和茶轴怎么选？

简介

Manticore是一个用python开发的用来进行动态二进制分析的开源工具，可以帮助我们快速地利用符号执行、污点分析和插桩来分析二进制。

Manticore有一个命令行工具，可以用符号执行来产生程序的测试用例，当运行程序时，每个测试用例会产生不同的输出结果，例如一个普通的退出或一个崩溃。

命令行工具可以满足一些应用场景，但是实际的使用更加灵活，可以使用它提供的Python API来自定义分析和根据应用进行优化。Manticore API可以：舍弃无关紧要的状态

在任意执行点运行自定义分析函数

具体化符号内存

内省和修改模拟的机器状态

安装

1.安装系统依赖1

2$ sudo apt-get update && sudo apt-get install z3 python-pip -y

$ python -m pip install -U pip

2.安装manticore1

2$ git clone https://github.com/trailofbits/manticore.git && cd manticore

$ sudo pip install .

3.例子1

2$ cd examples/linux

$ make

命令行工具：1

2

3

4

5$ manticore basic

$ cat mcore_8PZ_Lo/test_00000001.stdin| ./basic

Message: It is less than or equal to 0x41

$ cat mcore_8PZ_Lo/test_00000002.stdin| ./basic

Message: It is greater than 0x41

Manticore API：1

2

3$ cd ../script

python count_instructions.py ../linux/helloworld

Executed 6897 instructions.

实验例子

下面是multiple-styles的writeup。

用IDA反汇编multiple-styles的main函数如下：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int result; // eax@3
  unsigned __int64 v4; // rax@5
  __int64 v5; // rcx@7
  int i; // [sp+28h] [bp-58h]@1
  char buf[32]; // [sp+30h] [bp-50h]@1
  char v8[18]; // [sp+50h] [bp-30h]@1
  __int64 canary; // [sp+68h] [bp-18h]@1

  canary = *MK_FP(__FS__, 40LL);
  read(0LL, buf, 17LL);
  strcpy(v8, "myvnvsuowsxs}ynk)");
  for ( i = 0; ; ++i )
  {
    LODWORD(v4) = strlen(v8);
    if ( i >= v4 )
      break;
    if ( v8[i] != buf[i] + 10 )
    {
      puts("sounds fake but ok");
      result = 1;
      goto LABEL_7;
    }
  }
  puts("you got it!");
  result = 0;
LABEL_7:
  v5 = *MK_FP(__FS__, 40LL) ^ canary;
  return result;
}
```

看到程序可能输出”sounds fake but ok”或”you got it!”，猜测后者是正确输入后的结果。代码逻辑比较简单，目的是对manticore有一个初步了解。

如果我们执行命令manticore multiple-styles，manticore将会开始自动分析二进制文件，最终会得出到达所有代码路径的输入，但是这会花费大量时间。通过部分人工分析，可以写出优化的脚本。

插桩解决方法

解决方法脚本如下：1

2

3

4

5

6

7

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28from manticore import Manticore
m = Manticore('./multiple-styles')
m.context['flag'] = "" # 在manticore的context字典中创建一个flag条目，在这里存储flag，可以在不同的hook之间访问
# 下面创建一些hook，hook告诉manticore在执行指定地址的指令之前挂起程序，先执行我们的代码，然后继续执行
def (state):
cpu = state.cpu
m.context['flag'] += chr(cpu.AL - 10) # flag的值存在EAX的最低字节AL中
@m.hook(0x400a3e)
def hook2(state):
cpu = state.cpu
```

cpu.ZF = True # 设置zero flag为True，使得不管字符是否匹配都能到下一轮

下面两个hook只是打印flag

@m.hook(0x400a40)

def hookf(state):

print("Failed")

m.terminate()

@m.hook(0x400a6c)

def hooks(state):

print("Success!")

print(m.context['flag'])

m.terminate()

设置m.concrete_data为17字节的填充，传入stdin

m.concrete_data = "12345678" * 2 + "n"

m.run()

运行脚本：1

2

3$ python concrete.py

Success!

coldlikeminisodas

符号执行解决方法

理想的状态是不需要我们自己弄明白如何计算输入到stdin的值，这里的逆向非常容易，但是真实的问题不会这么简单，manticore的符号执行的能力使得我们可以让它来做这些工作。

解决方法脚本如下：1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17from manticore import Manticore

m = Manticore('./multiple-styles')

m.context['flag'] = ""

@m.hook(0x400a6c) # success的地址

def (state):

cpu = state.cpu

transform_base = cpu.RBP - 0x50

for i in range(17):

solved = state.solve_one(cpu.read_int(transform_base + i, 8))

print(solved)

m.context['flag'] += chr(solved)

print(m.context['flag'])

m.terminate()

m.run()

这种解决方法为manticore指定一个目的地址，然后帮它找到如何到达这里。通过分析二进制，它会从stdin读取值到位于RBP-0x50的buffer，我们把这个地址保存在变量transform_base中。然后循环buffer里的每一个字节，告诉manticore给这个字节找到一个值，能让我们到达当前地址。将找到的值一个个加入到flag中。1

2

3python symbolic.py

...

coldlikeminisodas

我们像可编程调试器一样使用manticore，修改循环所以它不会出错。我们为manticore标记hook的指令为success状态，state.solve_one操作使用z3来解决限制。限制的建立是通过检查程序剩余部分，找出需要填在内存transform_base+i中的值，使得我们能够到达当前状态。

reference

Manticore: Symbolic execution for humans

Multiple Styles: The Writeup