angr 学习笔记

前言

angr 是一个基于 符号执行 和 模拟执行 的二进制框架，可以用在很多的场景，比如逆向分析，漏洞挖掘等。本文对他的学习做一个总结。

安装

这里介绍 ubuntu 下的安装，其他平台可以看 官方文档

首先安装一些依赖包

```
sudo apt-get install python-dev libffi-dev build-essential virtualenvwrapper
```

然后使用

```
mkvirtualenv angr && pip install angr
```

即可安装

> 建议使用 virtualenv 来安装，因为 angr 用到的一些库和正常下的不一样，直接 pip 安装可能会安装不上去

angr常用对象及简单使用

使用 angr 的大概步骤

- 创建 project
- 设置 state
- 新建 符号量：BVS (bitvector symbolic) 或 BVV (bitvector value)
- 把符号量设置到内存或者其他地方
- 设置 Simulation Managers， 进行路径探索的对象
- 运行，探索满足路径需要的值
- 约束求解，获取执行结果

Project对象

介绍与简单使用

载入二进制文件使用 angr.Project 函数，它的第一个参数是待载入文件的路径，后面还有很多的可选参数，具体可以看 官方文档 。

```
p = angr.Project('./issue', load_options={"auto_load_libs": False})
```

> auto_load_libs 设置是否自动载入依赖的库，如果设置为 True 的话会自动载入依赖的库，然后分析到库函数调用时也会进入库函数，这样会增加分析的工作量，也有能会跑挂。

载入文件后，就可以通过 project 对象获取信息以及进行后面的操作

```
In [11]: proj = angr.Project('/bin/true')

In [12]: proj.loader.shared_objects
Out[12]: OrderedDict([('true', <ELF Object true, maps [0x400000:0x6063bf]>), (u'libc.so.6', <ELF Object libc-2.23.so, maps [0x1000000:0x13c999f]>), (u'ld-linux-x86-64.so.2', <EL

In [13]: proj = angr.Project('/bin/true', load_options={"auto_load_libs": False})

In [14]: proj.loader.shared_objects
Out[14]: OrderedDict([('true', <ELF Object true, maps [0x400000:0x6063bf]>)])

In [15]:
```

可以看到在使用 {"auto_load_libs": False} 后一些动态链接库没有被载入。

有两个小点还需要了解一下

- 如果 auto_load_libs 为 true, 那么程序如果调用到库函数的话就会直接调用 **真正的库函数** ，如果有的库函数逻辑比较复杂，可能分析程序就出不来了~~。同时 angr 使用 python 实现了很多的库函数（保存在 angr.SIM_PROCEDURES 里面），默认情况下会使用列表内部的函数来替换实际的函数调用，如果不在列表内才会进入到真正的 library.
- 如果 auto_load_libs 为 false， 程序调用函数时，会直接返回一个 不受约束的符号值。

hook

我们可以在 angr 中使用 hook 来把指定地址的二进制代码替换为 python 代码。angr 在模拟执行程序时，执行每一条指令前会检测该地址处是否已经被 hook ，如果是就不执

行这条语句，转而执行 hook 时指定的 python 处理代码。

下面看实例

目标程序地址

https://github.com/angr/angr-doc/tree/master/examples/sym-write

示例脚本

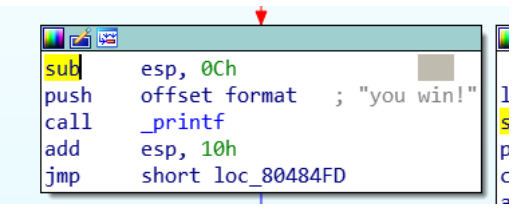
```
#!/usr/bin/env python
# coding=utf-8
import angr
import claripy


def hook_demo(state):
    state.regs.eax = 0
    state.regs.ebx = 0xdeadbeef

p = angr.Project("./examples/sym-write/issue", load_options={"auto_load_libs": False})
p.hook(addr=0x08048485, hook=hook_demo, length=2)
state = p.factory.blank_state(addr=0x0804846B, add_options={"SYMBOLIC_WRITE_ADDRESSES"})


u = claripy.BVS("u", 8)
state.memory.store(0x0804A021, u)
sm = p.factory.simgr(state)
sm.explore(find=0x080484DB)
st = sm.found[0]
print hex(st.se.eval(st.regs.ebx))
```

介绍一下脚本的流程

- 首先 使用 angr.Project 载入文件， 设置 auto_load_libs 为 false 则不加载依赖的 lib
- 然后 使用 p.hook 把 0x08048485 处的 2 字节的指令 为 hook_demo，之后执行 0x08048485 就会去执行 hook_demo
- 然后创建一个 state，**因为要往内存里面设置 符号量 （BVS），设置** SYMBOLIC_WRITE_ADDRESSES
- 然后新建一个 8 位长度的符号量，并把它存到 0x0804A021 (全局变量 u 的位置)

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  _BOOL4 v3; // eax
  signed int i; // [esp+0h] [ebp-18h]
  int v6; // [esp+4h] [ebp-14h]
  int v7; // [esp+8h] [ebp-10h]
  unsigned int v8; // [esp+Ch] [ebp-Ch]

  v8 = __readgsdword(0x14u);
  v6 = 0;
  v7 = 0;
  for ( i = 0; i <= 7; ++i )
  {
    v3 = ((u >> i) & 1) != 0;                   // 根据 u 的值计算，来决定跳转方向
    ++*(&v6 + v3);
  }
  if ( v6 == v7 )
    printf("you win!");
  else
    printf("you lose!");
  return 0;
}
```

- 然后开始探索路径，最后求解出使得 程序执行到 you win 代码块的符号量的解。



这里主要讲 p.hook 的处理， 这里使用了 hook 函数的三个参数

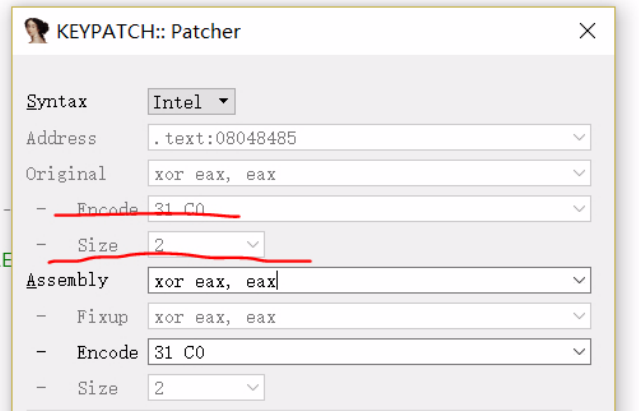
p.hook(addr=0x08048485, hook=hook_demo, length=2)

- addr 为待 hook 指令的地址

- `hook` 为 `hook` 的处理函数，在执行到 `addr` 时，会执行 这个函数，同时把 当前的 `state` 对象作为参数传递过去
- `length` 为 待 `hook` 指令的长度，在 执行完 `hook` 函数以后，`angr` 需要根据 `length` 来跳过这条指令，执行下一条指令

在上面的示例中， `hook` 了 `0x08048485` 处的指令

```
.text:0804847C                  mov     eax, large gs:14h
.text:08048482                  mov     [ebp+var_C], eax
.text:08048485                  xor     eax, eax
.text:08048487                  mov     [ebp+var_14], 0
.text:0804848E                  mov     [ebp+var_10], 0
.text:08048495                  mov     [ebp+var_18], 0
.text:0804849C                  jmp     short loc_80484CB
.text:0804849E ; ---------------------------------------------------------------------------
.text:0804849E
.text:0804849E loc_804849E:                             ; CODE XRE
.text:0804849E                  movzx   eax, ds:u
.text:080484A5                  movsx   edx, al
.text:080484A8                  mov     eax, [ebp+var_18]
.text:080484AB                  mov     ecx, eax
.text:080484AD                  sar     edx, cl
```



这是一条 `xor eax, eax` 的指令，长度为 `2` .

```
def hook_demo(state):
    state.regs.eax = 0
    state.regs.ebx = 0xdeadbeef
```

为了做示范，这里就是把 `eax` 设置为 `0` （ `xor eax,eax` 的作用）， 然后 设置 `ebx`为 `0xdeadbeef`， 因为后续不会用到 `ebx` , 修改它可以在路径探索完后查看这个值是否符合预期。

```
In [8]: %run hello_angr.py
0xdeadbeef

In [9]:
```

可以看到 `ebx` 被修改成了 `0xdeadbeef` 。

SimState对象

这个对象保存着程序运行到某一阶段的状态信息。

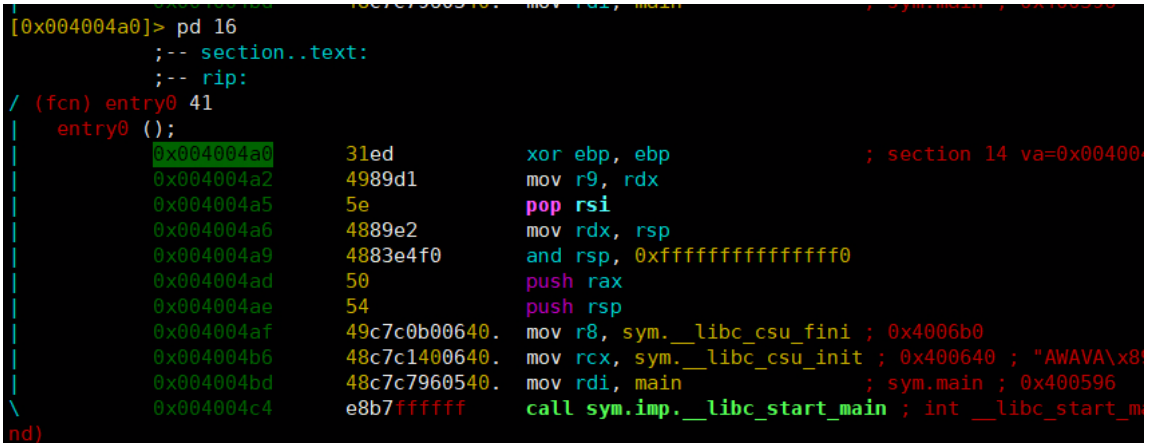
通过这个对象可以操作某一运行状态的上下文信息，比如内存，寄存器等

创建state

```
In [8]: p = angr.Project("./hello_angr")

In [9]: st = p.factory.entry_state()

In [10]: st.regs.rsp
Out[10]: <BV64 0x7ffffffffeff98>

In [11]: st
Out[11]: <SimState @ 0x4004a0>

In [12]:
```

首先加载二进制分析文件，创建 `project` 对象，然后创建一个 `entry_state` ， 之后就可以通过 这个 `state` 对象，获取或者修改此时程序的运行状态

`entry_state` ： 做一些初始化工作，然后在 程序的 入口停下



还有一个用的比较多的是

```
st = p.factory.blank_state(addr=0x4004a0)
```

这会创建一个 `blank_state` 对象，这个对象里面很多东西都是未初始化的，当程序访问未初始化的数据时，会返回一个不受约束的符号量

基本操作

`state` 对象一般是作为 **符号执行开始前** 创建用来 为 后续的执行 初始化一些数据，比如栈状态，寄存器值。

或者在 **路径探索结束后** 返回一个 `state` 对象供用户提取需要的值或进行 **约束求解** ，解出到达目标分支所使用的**符号量的值。**

访问寄存器

通过 `state.regs` 对象的属性访问以及修改寄存器的数据

```
In [12]: state.regs.r
state.regs.r10  state.regs.r14  state.regs.rax  state.regs.rdi  state.regs.rip
state.regs.r11  state.regs.r15  state.regs.rbp  state.regs.rdx  state.regs.rsi
state.regs.r12  state.regs.r8   state.regs.rbx  state.regs.register_default  state.regs.rsp
state.regs.r13  state.regs.r9   state.regs.rcx  state.regs.rflags

# 获取 rip 的值
In [12]: state.regs.rip
Out[12]: <BV64 0x400470>

# 获取 rsp 的值
In [13]: state.regs.rsp
Out[13]: <BV64 0x7fffffffeff78>

# 获取 rbp 的值
In [14]: state.regs.rbp
Out[14]: <BV64 reg_38_36_64{UNINITIALIZED}>

# 设置 rbp = rsp + 0x40
In [15]: state.regs.rbp = state.regs.rsp + 0x40

In [16]: state.regs.rbp
Out[16]: <BV64 0x7fffffffeffb8>

# 对于 BVV 和 BVS 都需要通过 solver 进行求解得到具体的值
In [26]: hex(state.se.eval(state.regs.rbp))
Out[26]: '0x7fffffffeffb8L'

In [27]: hex(state.solver.eval(state.regs.rbp))
Out[27]: '0x7fffffffeffb8L'
```

访问内存

有两种方式访问内存，一个是通过 `state.mem` 使用数组索引类似的方式进行访问

```
In [64]: state.mem[state.regs.rsp].qword
Out[64]: <uint64_t <BV64 0x2> at 0x7fffffffeff78>

In [65]: state.mem[state.regs.rsp].qword = 0xdeadbeefdeadbeef

In [66]: state.mem[state.regs.rsp].qword
```

```
Out[66]: <uint64_t <BV64 0xdeadbeefdeadbeef> at 0x7fffffffffeff78>

In [67]: m = state.mem[state.regs.rsp]

In [68]: m.
m.STRONGREF_STATE      m.double              m.int32_t             m.register_default    m.ssize               m.uint32_t            m.wstring
m.array                m.dword               m.int64_t             m.resolvable          m.ssize_t             m.uint64_t
m.byte                 m.example             m.int8_t              m.resolved            m.state               m.uint8_t
m.char                 m.float               m.long                m.set_state           m.store               m.uintptr_t
m.concrete             m.init_state          m.merge               m.set_strongref_state m.string              m.void
m.copy                 m.int                 m.ptrdiff_t           m.short               m.types               m.widen
m.deref                m.intl6_t             m.qword               m.size_t              m.uintl6_t            m.word
```

通过 得到的是一个 `SimMemView` 对象， 可以这个对象的属性决定按照什么方式进行内存访问。

```
In [71]: m.dword   # 按照 dword 进行访问， 4 字节
Out[71]: <uint32_t <BV32 0xdeadbeef> at 0x7fffffffffeff78>

In [72]: m.qword   # 按照 qword 进行访问， 8 字节
Out[72]: <uint64_t <BV64 0xdeadbeefdeadbeef> at 0x7fffffffffeff78>

In [73]: m.int
Out[73]: <int (32 bits) <BV32 0xdeadbeef> at 0x7fffffffffeff78>

In [74]: m.uin
m.uint16_t    m.uint32_t    m.uint64_t    m.uint8_t     m.uintptr_t

In [74]: m.uint64_t
Out[74]: <uint64_t <BV64 0xdeadbeefdeadbeef> at 0x7fffffffffeff78>
```

这些值如果需要把它转成 `python` 中的基本数据类型

```
# 通过 .resolved 转成 BVV 对象
In [75]: state.se.eval(m.qword.resolved)
Out[75]: 16045690984833335023L

# 通过求解器拿到具体值
In [76]: hex(state.se.eval(m.qword.resolved))
Out[76]: '0xdeadbeefdeadbeefL'
```

或者可以通过 `state.memory` 的 `load` 和 `store` 来读取和写入 数据到内存

```
In [90]: data = claripy.BVV(0xaaaaaaaaaabbbbbbbbbbbb, 0x20 * 8)

In [91]: data
Out[91]: <BV256 0xaaaaaaaaaabbbbbbbbbbbb>

In [92]: state.memory.load(state.regs.rsp, 0x40)
Out[92]: <BV512 0xdeadbeefefbeaddec0fffefffffffff07ccfffefffffffff07000000000000000000000000000000001900000000000000ddfffefffffffff070000000000000000>

# 存数据存的是 BVV 对象
In [93]: state.memory.store(state.regs.rsp,data)

In [94]: state.memory.load(state.regs.rsp, 0x40)
Out[94]: <BV512 0xaaaaaaaaaabbbbbbbbbbbb0000000000000000190000000000000ddfffefffffffff070000000000000000>
```

此外还可以往内存里面设置符号变量 (`BVS`)

```
In [96]: data = claripy.BVS("data", 0x20 * 8)

In [97]: data
Out[97]: <BV256 data_37_256>

In [98]: state.memory.store(state.regs.rsp,data)

In [99]: state.memory.load(state.regs.rsp, 0x40)
Out[99]: <BV512 data_37_256 .. 0x1900000000000000ddfffefffffffff070000000000000000#256>
```

此时还需在创建 `state` 时 设置 `SYMBOLIC_WRITE_ADDRESSES`， 例如

```
state = p.factory.blank_state(addr=0x0804846B, add_options={"SYMBOLIC_WRITE_ADDRESSES"})
```

模拟执行

用的的程序

https://github.com/angr/angr-doc/tree/master/examples/fauxware

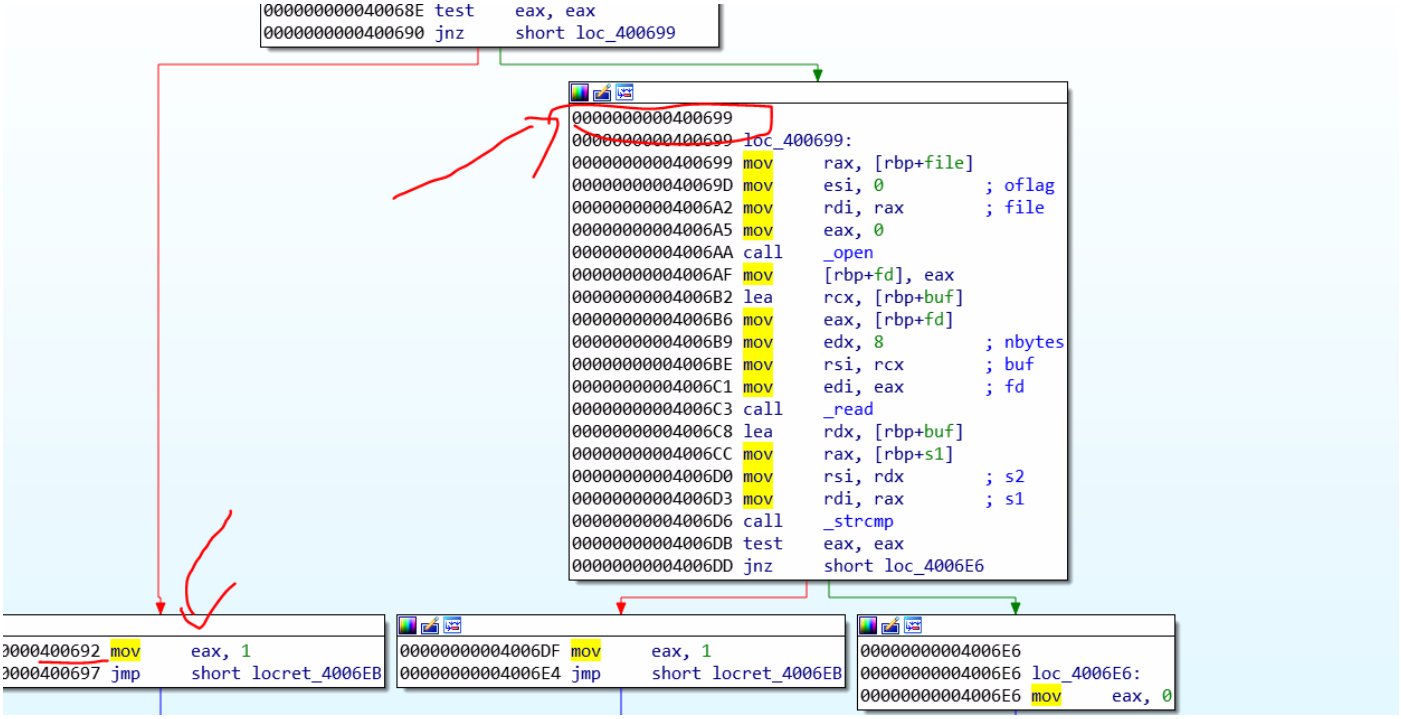
可以通过 `state` 对象来执行代码块

```
proj = angr.Project('examples/fauxware/fauxware')
state = proj.factory.entry_state()
while True:
    succ = state.step()
    if len(succ.successors) == 2:
        break
    state = succ.successors[0]

state1, state2 = succ.successors
state1
state2
```

上面的代码就是一直执行直到出现两个分支时停下

```
In [9]: proj = angr.Project('examples/fauxware/fauxware')

In [10]: state = proj.factory.entry_state()

In [11]: while True:
   ....:         succ = state.step()
   ....:         if len(succ.successors) == 2:
   ....:                 break
   ....:         state = succ.successors[0]
   ....:

In [12]: state1, state2 = succ.successors

In [13]: state1
Out[13]: <SimState @ 0x400692>

In [14]: state2
Out[14]: <SimState @ 0x400699>

In [15]:
```

这两个分支位于 `authenticate` 函数里面



然后使用

```
In [7]: state1.posix.dumps(0)
Out[7]: '\x00\x00\x00\x00\x00\x00\x00\x00\x00SOSNEAKY\x00'

In [8]: state2.posix.dumps(0)
Out[8]: '\x00\x00\x00\x00\x00\x00\x00\x00\x00S\x00\x80N\x00\x00 \x00\x00'
```

获取进入特定分支，需要往 `stdin` 输入的数据。

可以看到这里如果要进入返回 1 的分支(state1)，只要往 stdin 输入 SOSNEAKY，从而认证通过，这是一个后门密码。

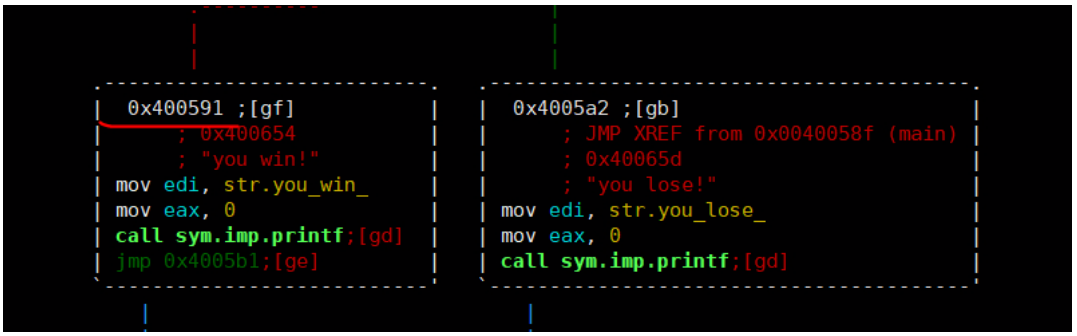
angr 重写了一些 libc 的函数，比如获取 stdin 数据，会返回符号量，用于符号执行，在某个状态下可以使用 state1.posix.dumps(0) 获取进入该状态时 stdin需要输入的数据 （0 表示的就是 stdin, 1 则是 stdout)。

传入命令行参数

创建 state 时还可以设置 命令行参数为 符号量 .下面用一个简单的例子

```
#include <stdio.h>
#include <string.h>
int main(int argc, char** argv)
{

    if (!strcmp(argv[1], "hello args test")) {
        printf("you win!");
    }
    else {
        printf("you lose!");
    }
    return 0;
}
```

这里需要传入一个命令行参数，参数值如果为 hello args test 就会进入 you win 分支



you win 分支所在代码块的地址为 0x400591. 所以我们就需要通过符号执行让层序执行到 0x400591。

脚本如下

```
#!/usr/bin/env python
# coding=utf-8
import angr
import claripy

p = angr.Project("./args_test")
args = claripy.BVS("args", 8 * 16)
state = p.factory.entry_state(args=['./args_test', args])

sm = p.factory.simgr(state)
sm.explore(find=0x400591)
st = sm.found[0]
print st.se.eval(args,cast_to=str)
```

- 首先加载文件，然后设置 args 符号量，长度为 16 字节 （ 8*16 位）
- 然后创建一个 entry_state 同时把 args 作为第一个参数传给程序。
- 之后创建 simgr 对象进行路径探索， 指定要走到的目标代码块 （ 0x400591 ） 即 win 所在的代码块
- 然后使用 求解对象 st.se.eval(args,cast_to=str) 得到进入到该代码块用到的 符号量的值

cast_to=str， 用于把结果转成 字符串， 否则就是 16 进制字符串

建议使用 ipython 来执行脚本，执行完后脚本中的对象还会存在 ipython 的上下文中，可以方便做些其他的操作

```
In [9]: %run args_test.py
hello args test

In [10]: st.se.eval(args)
Out[10]: 138766332635613996189760731817327162368L
```

SimulationManager对象

这个对象用于具体的路径探索。

以一个简单的例子开始

程序来自

https://github.com/angr/angr-doc/tree/master/examples/fauxware

解决的代码

```
#!/usr/bin/env python
import angr
p = angr.Project('fauxware')
state = p.factory.entry_state()
sm = p.factory.simgr(state)
sm.explore(find=0x04007BD)
st = sm.found[0]
print st.posix.dumps(0).strip("\x00")
```

- 创建一个 `entry_state`
- 然后创建 `SimulationManager` 对象 `sm` 进行路径探索
- 指定我们想要走到的位置是 `0x04007BD` （即认证通过的分支）
- 找到以后从 `sm.found[0]` 拿到此时的 `state`, 然后获取 `stdin` 输入的数据，就可以知道走到该分支需要从 `stdin` 输入的数据

```
In [10]: %run my.py
SOSNEAKY

In [11]:
```

其他更多请看

https://docs.angr.io/docs/

实例分析

`angr` 还搜集了许多使用 `angr` 解出来的 题目。下面就以其中的一些题来介绍 `angr`的使用， 用几遍就知道大概流程了。

csaw_wyvern

简单分析

程序位于

https://github.com/angr/angr-doc/tree/master/examples/csaw_wyvern

通过这个题可以了解到怎么往 `stdin` 里面放置符号量 以及 设置约束条件

先看看大概逻辑, 首先调用 `fgets` 获取输入保存到 `s`

```
.text:000000000040E1D8                lea     rcx, [rbp-110h]
.text:000000000040E1DF                mov     esi, 101h       ; n
.text:000000000040E1E4                mov     rdi, rcx        ; s
.text:000000000040E1E7                mov     [rbp-180h], rax
.text:000000000040E1EE                mov     [rbp-188h], rcx
.text:000000000040E1F5                call    _fgets
.text:000000000040E1FA                lea     rcx, [rbp-120h]
```

然后进入 `start_quest` ， 对输入进行处理

```
    std::vector<int,std::allocator<int>>::push_back(&hero, &secret_2575);
    v7 = std::string::length(v12) - 1LL != legend >> 2;
    if ( y26 < 10 || ((x25 - 1) * x25 & 1) == 0 )
      break;
LABEL_13:
```

这里判断输入字符串的长度是不是 `28` （ `fgets` 会把 输入的字符串 + `\n` 保存到缓冲区， `legend>>2` 为 `28`）

所以要求输入的字符串的长度应该为 `28` 个字节，且 每个字节都不是 `\x00` 或者 `\n`，第 `29` 个字节为 `\n`， 表示输入完成。

解答

最后的脚本为

```
#!/usr/bin/env python

import angr

p = angr.Project('wyvern')
st = p.factory.full_init_state()
```

```
# 设置 stdin 的约束条件， 使其 前 28 个字节 不能为 \x00 或者 \n
for _ in xrange(28):
    k = st.posix.files[0].read_from(1)
    st.se.add(k != 0)
    st.se.add(k != 10)

# 设置第 29 个字节为终止符，即为 \n
k = st.posix.files[0].read_from(1)
st.se.add(k == 10)

# 使得 文件指针指向文件的开头
st.posix.files[0].seek(0)
st.posix.files[0].length = 29


sm = p.factory.simgr(st)
sm.run()

# 因为是用的 sm.run() 所以要在 sm.deadended 里面寻找结果
for st in sm.deadended:
    out = st.posix.dumps(1)
    if "flag" in out:
        print out
```

- 首先创建一个 `full_init_state`， 因为这里是 `c++` 代码， 而 `angr` 只是 实现了一些常用的 `c` 函数，所以得加载所有的库， `c++` 的函数在底层才会去调用 `c` 的函数。创建 `full_init_state` 后 `angr` 就会跟进 `c++` 函数里面。
- 然后对 `stdin` 里面的前 `29` 个字节做约束条件， 前面已经分析过，要求输入的字符串长度为 `28`， 最后以 `\n` 结束
- 最后创建 `SimulationManager`，然后调用 `.run()` 跑到不能进行跑为止，然后遍历 `sm.deadended` 查看 `flag`， 因为如果输入正确就会打印出 `flag`。

运行示例

```
In [1]: %run my.py
WARNING | 2018-05-22 21:58:34,760 | angr.analyses.disassembly_utils | Your verison of capstone does not support MIPS instruction groups.
WARNING | 2018-05-22 21:58:41,288 | angr.manager | No completion state defined for SimulationManager; stepping until all states deadend
WARNING | 2018-05-22 22:17:50,610 | angr.state_plugins.symbolic_memory | Concretizing symbolic length. Much sad; think about implementing.
+-----------------------+
|    Welcome Hero       |
+-----------------------+

[!] Quest: there is a dragon prowling the domain.
    brute strength and magic is our only hope. Test your skill.

Enter the dragon's secret: success

[+] A great success! Here is a flag{dr4gOn_or_p4triclan_it5_LLVM}
```

通过打印的日志，可以看到大概运行了 `20` 分钟。下面介绍两种加速的方法

使用 pypy

`pypy` 是一个 `python` 的版本，采用 jit 的方法来提升 `python` 脚本的运行速度。

首先安装

```
sudo apt install pypy
```

然后安装 `pip`

```
wget https://bootstrap.pypa.io/get-pip.py
sudo pypy get-pip.py
```

然后使用 `pip` 安装 `angr`

```
sudo pip install angr
```

然后使用 `pypy` 来执行脚本即可

```
23:52 haclh@ubuntu:csaw_wyvern $ pypy my.py
WARNING | 2018-05-22 23:52:09,645 | angr.analyses.disassembly_utils | Your verison of capstone does not support MIPS instruction groups.
WARNING | 2018-05-22 23:52:17,667 | angr.manager | No completion state defined for SimulationManager; stepping until all states deadend
WARNING | 2018-05-22 23:58:23,344 | angr.state_plugins.symbolic_memory | Concretizing symbolic length. Much sad; think about implementing.
+-----------------------+
|    Welcome Hero       |
+-----------------------+

[!] Quest: there is a dragon prowling the domain.
```

```
    brute strength and magic is our only hope. Test your skill.

Enter the dragon's secret: success

[+] A great success! Here is a flag{dr4gOn_or_p4triclan_it5_LLVM}
```

此时只用了大概 6 分钟就跑完了。

使用 unicorn

还可以设置 `angr` 的选项，使用 `unicorn` 引擎来做模拟执行

```
#!/usr/bin/env python

import angr
p = angr.Project('wyvern')
st = p.factory.full_init_state(add_options=angr.options.unicorn)

for _ in xrange(28):
    k = st.posix.files[0].read_from(1)
    st.se.add(k != 0)
    st.se.add(k != 10)

k = st.posix.files[0].read_from(1)
st.se.add(k == 10)

st.posix.files[0].seek(0)
st.posix.files[0].length = 29

sm = p.factory.simgr(st)
sm.run()
for st in sm.deadended:
    out = st.posix.dumps(1)
    if "flag" in out:
        print out
```

然后再跑一次

```
23:58 haclh@ubuntu:csaw_wyvern $ pypy my.py
WARNING | 2018-05-22 23:59:26,853 | angr.analyses.disassembly_utils | Your verison of capstone does not support MIPS instruction groups.
WARNING | 2018-05-22 23:59:35,539 | angr.manager | No completion state defined for SimulationManager; stepping until all states deadend
WARNING | 2018-05-23 00:03:58,458 | angr.state_plugins.symbolic_memory | Concretizing symbolic length. Much sad; think about implementing.
+------------------------+
|    Welcome Hero        |
+------------------------+

[!] Quest: there is a dragon prowling the domain.
    brute strength and magic is our only hope. Test your skill.

Enter the dragon's secret: success

[+] A great success! Here is a flag{dr4gOn_or_p4triclan_it5_LLVM}
```

只用了 3 分钟就跑完了。

总结

- 对于 `c++` 的程序，如果调用了 `c++` 的函数，使用 `full_init_state`
- 如果通过 `sm.run()` 来探索路径，最后遍历 `sm.deadended` 查看结果
- 可以通过 `st.posix.files[0]` 对 `stdin` 做约束
- 可以使用 `pypy` 和 `unicorn` 来加速脚本的执行

cmu_binary_bomb

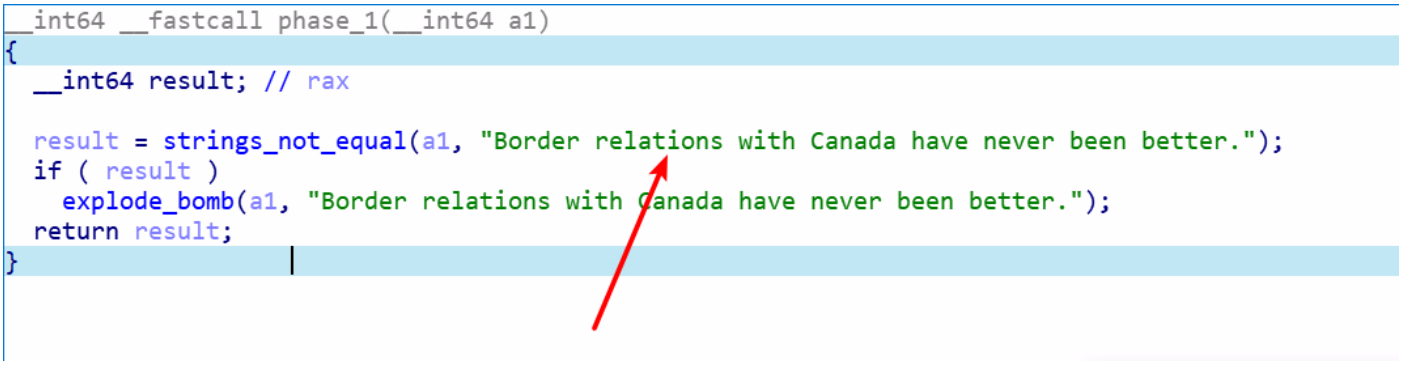
这个是 `cmu` 给学生练习逆向分析能力的一个题，相信大多数计算机专业的都做过这东西。今天看看怎么用 `angr` 来解决它。

程序文件位于

```
https://github.com/angr/angr-doc/tree/master/examples/cmu_binary_bomb
```

这个例子主要用于介绍 使用 `angr` 设置 内存符号量。

以第一个关卡为例



这个就是判断一个字符串。

于是我们把 一块内存设置为符号量， 然后把第一个参数设置为符号量的地址即可

```
import angr
import claripy

proj = angr.Project('bomb', load_options={'auto_load_libs': False})

start = 0x400ee0
bomb_explode = 0x40143a
end = 0x400ef7

# initial state is at the beginning of phase_one()
# 设置选项让 angr 支持 内存符号量
state = proj.factory.blank_state(addr=start)
state.options |= angr.options.unicorn
state.options |= {"SYMBOLIC_WRITE_ADDRESSES"}

# 初始化内存符号量， 128 字节
arg = state.se.BVS("input_string", 8 * 128)
bind_addr = 0x603780

# 符号量存在 内存中， 这块内存就变成了符号量
state.memory.store(bind_addr, arg)
# 设置 rdi (第一个参数为符号量的地址)
state.regs.rdi = bind_addr

sm = proj.factory.simgr(state)
sm.explore(find=end, avoid=bomb_explode)
st = sm.found[0]
# 求解符号量
print st.se.eval(arg, cast_to=str)
```

流程就是

- 首先设置选项，让 `angr` 支持 内存符号量
- 然后初始化一块 `128` 字节的符号量内存，并把符号量存到 `0x603780`， 此时 `0x603780` 处是一块符号量内存
- 然后把内存符号量的地址设置为参数传个第一关函数 `0x400ee0`

参考

https://github.com/axt/angr-utils
http://ysc21.github.io/blog/2016-01-27-angr-script.html
http://docs.angr.io/