前言

在基本了解了模糊测试以后，接下来就开始看看一直心心念念的符号执行吧。听群友说这个东西的概念在九几年就有了，算是个老东西，不过 Angr 本身倒是挺新的，看看这个工具能不能有什么收获吧。

按照计划，一方面是 Angr 的使用技巧，另一方面是 Angr 的源代码阅读。不过因为两者的内容都挺多的，所以本篇只写使用技巧部分，如果未来有这样的预订，或许还会有另外一篇。希望以我这种菜鸡水平也能看得懂吧。

Angr 的基本描述

首先在开始解释 `Angr` 的各个模块和使用之前，我们需要先对**它是如何工作的**有一个大概的认识。

我们一般用 `Angr` 的目的其实就是为了自动化的求解输入，比如说逆向或是 PWN。而它的原理被称之为“符号执行”。

`Angr` 其实并不是真正被运行起来的，它就向一个虚拟机，会读取每一条命令并在虚拟机中模拟该命令的行为。我们类比到更加常用的 `z3` 库中，每个寄存器都可以相当与 `z3` 中的一个变量，在模拟执行的过程中，这个变量会被延伸为一个表达式，而当我们成功找到了目标地址之后，通过表达式就可以求解对应的初值应该是什么了。

看着简单，但是您或许听说过，这类符号执行有一个现今仍为解决的麻烦问题：**路径爆炸**。

`Angr` 被称之为 `IR-Based` 类的符号执行引擎，他会对输入的二进制重建对应的 CFG ，在完成重建后开始模拟执行。而对于分支语句，就需要分支出两个不同的情况：`跳转` 和 `不跳转` 。在一般情况下，这不会引发问题，但是我们可以考虑如下的代码：

```
num=xxx
for(int i=0;i<1000;i++)//<---- judge 1
{
        if(num==0x6666){//<----- judge 2
                break;
        }
        else{
        num+=1;
        }
}
```

当符号执行引起遇到循环语句，由于循环语句本身就需要判断是否应该跳出循环，因此引擎会在这里开始分叉为两个情况。

而如果这个循环里又嵌套了判断条件，那么就需要再次分叉为两条路径。

也就是说，对于一个人为理解起来相当易懂的循环判断，符号执行引擎却会因此分叉出指数级别增长的分支数量。

但这还不是最简单的情况，我们可以更极端一点考虑这么一个情况：

```
while(1)
{
        if(condition)
        {
                break;
        }
}
```

循环本身是一个死循环，尽管我们靠自己的思维能够理解，它会在未来的某一个跳出循环，但符号执行引擎却不知道这件事，因此每一次遇到判断跳转都需要进行分叉，最后这个路径就会无限增长，最后把内存挤爆，然后程序崩溃。

说了这么多，其实是为了将清楚一件事，“**符号执行引擎是通过按行读取的方式模拟执行每条机器码，并更新对应变量，最后在通过约束求解的方式去逆推输入初值的**”。

Angr 基本模块

一般来说，使用 Angr 的基本流程如下：

```
import angr
project = angr.Project(path_to_binary, auto_load_libs=False)
state = project.factory.entry_state()
sim = project.factory.simgr(state)
sim.explore(find=target)
if simulation.found:
        res = simulation.found[0]
        res = res.posix.dumps(0)
        print("[+] Success! Solution is: {}".format(res.decode("utf-8")))
```

笔者一直以来都是套这个模板对二进制程序一把梭，但既然现在要开始正经思考一下怎么办，总要对里面的各种模块有所了解了。

Project 模块

```
project = angr.Project(path_to_binary, auto_load_libs=False)
```

对于一个使用 `angr.Project` 加载的二进制程序，`angr` 会读取它的一些基本属性：

```
>>> project=angr.Project("02_angr_find_condition",auto_load_libs=False)
>>> project.filename
'02_angr_find_condition'
>>> project.arch
<Arch X86 (LE)>
>>> hex(project.entry)
'0x8048450'
```

这些信息会由 `angr` 自动分析，但是如果你有需要，可以通过 `angr.Project` 中的其他参数手动进行设定。

Loader 模块

而对于一个 `Project` 对象，它拥有一个自己的 `Loader` ，提供如下信息：

```
>>> project.loader
<Loaded 02_angr_find_condition, maps [0x8048000:0x8407fff]>
>>> project.loader.main_object
<ELF Object 02_angr_find_condition, maps [0x8048000:0x804f03f]>
>>> project.loader.all_objects
[<ELF Object 02_angr_find_condition, maps [0x8048000:0x804f03f]>,
<ExternObject Object cle##externs, maps [0x8100000:0x8100018]>,
<ExternObject Object cle##externs, maps [0x8200000:0x8207fff]>,
<ELFTLSObjectV2 Object cle##tls, maps [0x8300000:0x8314807]>, <KernelObject
Object cle##kernel, maps [0x8400000:0x8407fff]>]
```

当然实际的属性不止这些，而且在常规的使用中似乎也用不到这些信息，不过这里为了完整性就一起记录一下吧。

Loader 模块主要是负责记录二进制程序的一些基本信息，包括段、符号、链接等。

```
>>> obj=project.loader.main_object
>>> obj.plt
{'strcmp': 134513616, 'printf': 134513632, '__stack_chk_fail': 134513648,
'puts': 134513664, 'exit': 134513680, '__libc_start_main': 134513696,
'__isoc99_scanf': 134513
712, '__gmon_start__': 134513728}
>>> obj.sections
<Regions: [<Unnamed | offset 0x0, vaddr 0x0, size 0x0>, <.interp | offset
0x154, vaddr 0x8048154, size 0x13>, <.note.ABI-tag | offset 0x168, vaddr
```

```
0x8048168, size 0x20>
, <.note.gnu.build-id | offset 0x188, vaddr 0x8048188, size 0x24>,
<.gnu.hash | offset 0x1ac, vaddr 0x80481ac, size 0x20>, <.dynsym | offset
0x1cc, vaddr 0x80481cc, siz
e 0xa0>, <.dynstr | offset 0x26c, vaddr 0x804826c, size 0x91>, <.gnu.version
| offset 0x2fe, vaddr 0x80482fe, size 0x14>, <.gnu.version_r | offset 0x314,
vaddr 0x804831
4, size 0x40>, <.rel.dyn | offset 0x354, vaddr 0x8048354, size 0x8>,
<.rel.plt | offset 0x35c, vaddr 0x804835c, size 0x38>, <.init | offset
0x394, vaddr 0x8048394, size
0x23>, <.plt | offset 0x3c0, vaddr 0x80483c0, size 0x80>, <.plt.got | offset
0x440, vaddr 0x8048440, size 0x8>, <.text | offset 0x450, vaddr 0x8048450,
size 0x4ea2>, <
.fini | offset 0x52f4, vaddr 0x804d2f4, size 0x14>, <.rodata | offset
0x5308, vaddr 0x804d308, size 0x39>, <.eh_frame_hdr | offset 0x5344, vaddr
0x804d344, size 0x3c>,
<.eh_frame | offset 0x5380, vaddr 0x804d380, size 0x110>, <.init_array |
offset 0x5f08, vaddr 0x804ef08, size 0x4>, <.fini_array | offset 0x5f0c,
vaddr 0x804ef0c, size
0x4>, <.jcr | offset 0x5f10, vaddr 0x804ef10, size 0x4>, <.dynamic | offset
0x5f14, vaddr 0x804ef14, size 0xe8>, <.got | offset 0x5ffc, vaddr 0x804effc,
size 0x4>, <.go
t.plt | offset 0x6000, vaddr 0x804f000, size 0x28>, <.data | offset 0x6028,
vaddr 0x804f028, size 0x15>, <.bss | offset 0x603d, vaddr 0x804f03d, size
0x3>, <.comment |
offset 0x603d, vaddr 0x0, size 0x34>, <.shstrtab | offset 0x67fa, vaddr 0x0,
size 0x10a>, <.symtab | offset 0x6074, vaddr 0x0, size 0x4d0>, <.strtab |
offset 0x6544, va
ddr 0x0, size 0x2b6>]>
```

对外部库的链接也同样支持查找：

```
>>> project.loader.find_symbol('strcmp')
<Symbol "strcmp" in cle##externs at 0x8100000>
>>> project.loader.find_symbol('strcmp').rebased_addr
135266304
>>> project.loader.find_symbol('strcmp').linked_addr
0
>>> project.loader.find_symbol('strcmp').relative_addr
0
```

同时也支持一些加载选项：

- auto_load_libs：是否自动加载程序的依赖
- skip_libs：避免加载的库

- except_missing_libs：无法解析共享库时是否抛出异常
- force_load_libs：强制加载的库
- ld_path：共享库的优先搜索搜寻路径

我们知道，在一般情况下，加载程序都会将 `auto_load_libs` 置为 `False` ，这是因为如果将外部库一并加载，那么 `Angr` 就也会跟着一起去分析那些库了，这对性能的消耗是比较大的。

而对于一些比较常规的函数，比如说 `malloc` 、`printf`、`strcpy` 等，`Angr` 内置了一些替代函数去 hook 这些系统库函数，因此即便不去加载 `libc.so.6` ，也能保证分析的正确性。这部分内容接下来会另说。

factory 模块

该模块主要负责将 `Project` 实例化。

我们知道，加载一个二进制程序只是符号执行能够开始的第一步，为了实现符号执行，我们还需要为这个二进制程序去构建符号、执行流等操作。这些操作会由 `Angr` 帮我们完成，而它也提供一些方法能够让我们获取到它构造的一些细节。

Block 模块

`Angr` 对程序进行抽象的一个关键步骤就是从二进制机器码去重构 CFG ，而 `Block` 模块提供了和它抽象出的基本块间的交互接口：

```
>>> project.factory.block(project.entry)
<Block for 0x8048450, 33 bytes>
>>> project.factory.block(project.entry).pp()
        _start:
8048450  xor     ebp, ebp
8048452  pop     esi
8048453  mov     ecx, esp
8048455  and     esp, 0xfffffff0
8048458  push    eax
8048459  push    esp
804845a  push    edx
804845b  push    __libc_csu_fini
8048460  push    __libc_csu_init
8048465  push    ecx
8048466  push    esi
8048467  push    main
804846c  call    __libc_start_main
>>> project.factory.block(project.entry).instruction_addrs
(134513744, 134513746, 134513747, 134513749, 134513752, 134513753,
134513754, 134513755, 134513760, 134513765, 134513766, 134513767, 134513772)
```

可以看出 `Angr` 用 `call` 指令作为一个基本块的结尾。在 `Angr` 中，它所识别的基本块和 IDA 里看见的 CFG 有些许不同，它会把所有的跳转都尽可能的当作一个基本块的结尾。

> 当然也有无法识别的情况，比如说使用寄存器进行跳转，而寄存器的值是上下文有关的，它有可能是函数开始时传入的一个回调函数，而参数有可能有很多种，因此并不是总能够识别出结果的。

```
>>> block.
block.BLOCK_MAX_SIZE           block.capstone
block.instructions             block.reset_initial_regs()     block.size
block.addr                     block.codenode                 block.parse(
block.serialize()              block.thumb
block.arch                     block.disassembly
block.parse_from_cmessage(     block.serialize_to_cmessage()  block.vex
block.bytes                    block.instruction_addrs        block.pp(
block.set_initial_regs()       block.vex_nostmt
```

State 模块

```
>>> state=project.factory.entry_state()
<SimState @ 0x8048450>
>>> state.regs.eip
<BV32 0x8048450>
>>> state.mem[project.entry].int.resolved
<BV32 0x895eed31>
>>> state.mem[0x1000].long = 4
>>> state.mem[0x1000].long.resolved
<BV32 0x4>
```

这个 `state` 包括了符号实行中所需要的所有符号。

通过 `state.regs.eip` 可以看出，所有的寄存器都会替换为一个符号。该符号可以由模块自行推算，也可以人为的进行更改。也正因如此，`Angr` 能够通过条件约束对符号的值进行解方程，从而去计算输入，比如说：

```
>>> bv = state.solver.BVV(0x2333, 32)
<BV32 0x2333>
>>> state.solver.eval(bv)
9011
```

另外还存在一些值，它只有在运行时才能够得知，对于这些值，`Angr` 会将它标记为 `UNINITIALIZED`：

```
>>> state.regs.edi
WARNING  | 2023-04-12 17:28:41,490 |
```

```
angr.storage.memory_mixins.default_filler_mixin | The program is accessing
register with an unspecified value. This could indicate
unwanted behavior.
WARNING  | 2023-04-12 17:28:41,491 |
angr.storage.memory_mixins.default_filler_mixin | angr will cope with this
by generating an unconstrained symbolic variable and con
tinuing. You can resolve this by:
WARNING  | 2023-04-12 17:28:41,491 |
angr.storage.memory_mixins.default_filler_mixin | 1) setting a value to the
initial state
WARNING  | 2023-04-12 17:28:41,492 |
angr.storage.memory_mixins.default_filler_mixin | 2) adding the state option
ZERO_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to make un
known regions hold null
WARNING  | 2023-04-12 17:28:41,492 |
angr.storage.memory_mixins.default_filler_mixin | 3) adding the state option
SYMBOL_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to suppr
ess these messages.
WARNING  | 2023-04-12 17:28:41,492 |
angr.storage.memory_mixins.default_filler_mixin | Filling register edi with
4 unconstrained bytes referenced from 0x8048450 (_start
+0x0 in 02_angr_find_condition (0x8048450))
<BV32 reg_edi_1_32{UNINITIALIZED}>
```

另外值得一提的是，除了 `entry_state` 外还有其他状态可用于初始化：

- blank_state：构造一个“空白板”空白状态，其中大部分数据未初始化。当访问未初始化的数据时，将返回一个不受约束的符号值。
- entry_state：造一个准备在主二进制文件的入口点执行的状态。
- full_init_state：构造一个准备好通过任何需要在主二进制文件入口点之前运行的初始化程序执行的状态，例如，共享库构造函数或预初始化程序。完成这些后，它将跳转到入口点。
- call_state：构造一个准备好执行给定函数的状态。

这些构造函数都能通过参数 `addr` 来指定初始时的 `rip/eip` 地址。而 `call_state` 可以用这种方式来构造传参：`call_state(addr, arg1, arg2, ...)`

Simulation Managers 模块

SM(Simulation Managers)是一个用来管理 `State` 的模块，它需要为符号指出如何运行。

```
>>> simgr = project.factory.simulation_manager(state)
<SimulationManager with 1 active>
>>> simgr.active
[<SimState @ 0x8048450>]
```

通过 `step` 可以让这组模拟执行一个基本块：

```
>>> simgr.step()
<SimulationManager with 1 active>
>>> simgr.active
[<SimState @ 0x8048420>]
>>> simgr.active[0].regs.eip
<BV32 0x8048420>
```

此时的 `eip` 对应了 `__libc_start_main` 的地址。

同样也可以查看此时的模拟内存状态，可以发现它储存了函数的返回地址：

```
>>> simgr.active[0].mem[simgr.active[0].regs.esp].int.resolved
<BV32 0x8048471>
```

而我们比较熟悉的 `simgr` 其实就是 `simulation_manager` 简写：

```
>>> project.factory.simgr()
<SimulationManager with 1 active>
>>> project.factory.simulation_manager()
<SimulationManager with 1 active>
```

SimProcedure

在前文中提到过 `Angr` 会 hook 一些常用的库函数来提高效率。它支持一下这些外部库：

```
>>> angr.procedures.
angr.procedures.SIM_LIBRARIES   angr.procedures.glibc
angr.procedures.java_util       angr.procedures.ntdll
angr.procedures.uclibc
angr.procedures.SIM_PROCEDURES  angr.procedures.gnulib
angr.procedures.libc            angr.procedures.posix
angr.procedures.win32
angr.procedures.SimProcedures   angr.procedures.java
angr.procedures.libstdcpp       angr.procedures.procedure_dict
angr.procedures.win_user32
angr.procedures.advapi32        angr.procedures.java_io
angr.procedures.linux_kernel    angr.procedures.stubs
angr.procedures.cgc             angr.procedures.java_jni
angr.procedures.linux_loader    angr.procedures.testing
angr.procedures.definitions     angr.procedures.java_lang
angr.procedures.msvcr           angr.procedures.tracer
```

以 libc 为例就可以看到，它支持了一部分 libc 中的函数：

```
>>> angr.procedures.libc.
angr.procedures.libc.abort      angr.procedures.libc.fprintf
angr.procedures.libc.getuid     angr.procedures.libc.setvbuf
angr.procedures.libc.strstr
angr.procedures.libc.access     angr.procedures.libc.fputc
angr.procedures.libc.malloc     angr.procedures.libc.snprintf
angr.procedures.libc.strtol
......
由于函数过多，这里就不展示了
```

因此如果程序中调用了这部分函数，默认情况下就会由 `angr.procedures.libc` 中实现的函数进行接管。但是请务必注意，官方文档中也有提及，一部分函数的实现并不完善，比如说对 `scanf` 的格式化字符串支持并不是很好，因此有的时候需要自己编写函数来 hook 它。

hook 模块

紧接着上文提到的问题，`Angr` 接受由用户自定义函数来进行 hook 的操作。

```
>>> func=angr.SIM_PROCEDURES['libc']['scanf']
>>> project.hook(0x10000, func())
>>> project.hooked_by(0x10000)
<SimProcedure scanf>
>>> project.unhook(0x10000)
>>> project.hooked_by(0x10000)
WARNING  | 2023-04-12 19:20:39,782 | angr.project   | Address 0x10000 is not
hooked
```

第一种方案是直接对地址进行 hook，通过直接使用 `project.hook(addr,function())` 的方法直接钩取。

同时，`Angr` 对于有符号的二进制程序也运行直接对符号本身进行钩取：`project.hook_symbol(name,function)` 。

参考阅读

angr 系列教程(一）核心概念及模块解读
https://xz.aliyun.com/t/7117

angr documentation
https://docs.angr.io/en/latest/quickstart.html