Angr

A powerful and user-friendly binary analysis platform!

简介

angr 是一个多架构的二进制分析平台，具备对二进制文件的动态符号执行能力和多种静态分析能力。在近几年的 CTF 中也大有用途。

安装

在 Ubuntu 上，首先我们应该安装所有的编译所需要的依赖环境：
`$ sudo apt install python-dev libffi-dev build-essential virtualenvwrapper`

强烈建议在虚拟环境中安装 angr，因为有几个 angr 的依赖（比如z3）是从他们的原始库中 fork 而来，如果你已经安装了 z3,那么肯定不希望 angr 的依赖覆盖掉官方的共享库，开一个隔离的环境就好了： `$ mkvirtualenv angr $ sudo pip install angr`

安装过程中可能会有一些奇怪的错误，可以到官方文档中查看。

另外 angr 还有一个 GUI 可以用，查看 angr Management。

使用方法

快速入门

- 使用 angr 的第一步是新建一个工程，几乎所有的操作都是围绕这个工程展开的：

```
>>> import angr
>>> proj = angr.Project('/bin/true')
WARNING | 2017-12-08 10:46:58,836 | cle.loader | The main binary is a position-independent executable. It is being load
ed with a base address of 0x400000.
```

- 这样就得到了二进制文件的各种信息，如：

```
>>> proj.filename     # 文件名
'/bin/true'
>>> proj.arch         # 一个 archinfo.Arch 对象
<Arch AMD64 (LE)>
>>> hex(proj.entry)   # 入口点
'0x401370'
```

- 程序加载时会将二进制文件和共享库映射到虚拟地址中，CLE 模块就是用来处理这些东西的。

```
>>> proj.loader
<Loaded true, maps [0x400000:0x5008000]>
```

- 所有对象文件如下，其中二进制文件本身是 main_object，然后还可以查看对象文件的相关信息：

```
>>> for obj in proj.loader.all_objects:
...     print obj
...
<ELF Object true, maps [0x400000:0x60721f]>
<ELF Object libc-2.27.so, maps [0x1000000:0x13bb98f]>
<ELF Object ld-2.27.so, maps [0x2000000:0x22260f7]>
<ELFTLSObject Object cle##tls, maps [0x3000000:0x300d010]>
<ExternObject Object cle##externs, maps [0x4000000:0x4008000]>
<KernelObject Object cle##kernel, maps [0x5000000:0x5008000]>
>>> proj.loader.main_object
<ELF Object true, maps [0x400000:0x60721f]>
>>> hex(proj.loader.main_object.min_addr)
'0x400000'
>>> hex(proj.loader.main_object.max_addr)
'0x60721f'
>>> proj.loader.main_object.execstack
False
```

- 通常我们在创建工程时选择关闭 `auto_load_libs` 以避免 angr 加载共享库：

```
>>> p = angr.Project('/bin/true', auto_load_libs=False)
WARNING | 2017-12-08 11:09:28,629 | cle.loader | The main binary is a position-independent executable. It is being load
ed with a base address of 0x400000.
>>> p.loader.all_objects
[<ELF Object true, maps [0x400000:0x60721f]>, <ExternObject Object cle##externs, maps [0x1000000:0x1008000]>, <KernelOb
ject Object cle##kernel, maps [0x2000000:0x2008000]>, <ELFTLSObject Object cle##tls, maps [0x3000000:0x300d010]>]
```

- `project.factory` 提供了很多类对二进制文件进行分析，它提供了几个方便的构造函数。
- `project.factory.block()` 用于从给定地址解析一个 basic block，对象类型为 Block：

```
>>> block = proj.factory.block(proj.entry)    # 从程序头开始解析一个 basic block
>>> block
<Block for 0x401370, 42 bytes>
>>> block.pp()                  # 打印
0x401370:       xor     ebp, ebp
0x401372:       mov     r9, rdx
0x401375:       pop     rsi
0x401376:       mov     rdx, rsp
0x401379:       and     rsp, 0xfffffffffffffff0
0x40137d:       push    rax
0x40137e:       push    rsp
0x40137f:       lea     r8, qword ptr [rip + 0x32da]
0x401386:       lea     rcx, qword ptr [rip + 0x3263]
0x40138d:       lea     rdi, qword ptr [rip - 0xe4]
0x401394:       call    qword ptr [rip + 0x205b76]
>>> block.instructions          # 指令数量
11
>>> block.instruction_addrs     # 指令地址
[4199280L, 4199282L, 4199285L, 4199286L, 4199289L, 4199293L, 4199294L, 4199295L, 4199302L, 4199309L, 4199316L]
```

- 另外，还可以将 Block 对象转换成其他形式：

```
>>> block.capstone
<CapstoneBlock for 0x401370>
>>> block.capstone.pp()

>>> block.vex
IRSB <0x2a bytes, 11 ins., <Arch AMD64 (LE)>> at 0x401370
>>> block.vex.pp()
```

- 程序的执行需要初始化一个模拟程序状态的 `SimState` 对象：

```
>>> state = proj.factory.entry_state()
>>> state
<SimState @ 0x401370>
```

- 该对象包含了程序的内存、寄存器、文件系统数据等等模拟运行时动态变化的数据，例如：

```
>>> state.regs                          # 寄存器名对象
<angr.state_plugins.view.SimRegNameView object at 0x7f126fdfe810>
>>> state.regs.rip                      # BV64 对象
<BV64 0x401370>
>>> state.regs.rsp
<BV64 0x7fffffffffeff98>
>>> state.regs.rsp.length               # BV 对象都有 .length 属性
64
>>> state.regs.rdi
<BV64 reg_48_0_64{UNINITIALIZED}>       # BV64 对象，符号变量
>>> state.mem[proj.entry].int.resolved  # 将入口点的内存解释为 C 语言的 int 类型
<BV32 0x8949ed31>
```

- 这里的 BV，即 bitvectors，可以理解为一个比特串，用于在 angr 里表示 CPU 数据。看到在这里 rdi 有点特殊，它没有具体的数值，而是在符号执行中所使用的符号变量。

- 初始化的 state 可以经过模拟执行得到一系列的 states，模拟管理器（Simulation Managers）的作用就是对这些 states 进行管理：

```
>>> simgr = proj.factory.simulation_manager(state)
>>> simgr
<SimulationManager with 1 active>
>>> simgr.active                        # 当前 state
[<SimState @ 0x401370>]
>>> simgr.step()                        # 模拟执行一个 basic block
<SimulationManager with 1 active>
>>> simgr.active                        # 当前 state 被更新
[<SimState @ 0x1022f80>]
>>> simgr.active[0].regs.rip            # active[0] 是当前 state
<BV64 0x1022f80>
>>> state.regs.rip                      # 但原始的 state 并没有改变
<BV64 0x401370>
```

- angr 提供了大量函数用于程序分析，在这些函数在 `Project.analyses.`，例如：

```
>>> cfg = p.analyses.CFGFast()          # 得到 control-flow graph
>>> cfg
<CFGFast Analysis Result at 0x7f1265b62650>
>>> cfg.graph
<networkx.classes.digraph.DiGraph object at 0x7f1265e77310> # 详情请查看 networkx
>>> len(cfg.graph.nodes())
934
>>> entry_node = cfg.get_any_node(proj.entry)   # 得到给定地址的 CFGNode
>>> entry_node
<CFGNode 0x401370[42]>
>>> len(list(cfg.graph.successors(entry_node)))
2
```

如果要想画出图来，还需要安装 matplotlib。 ```python

> import networkx as nx import matplotlib matplotlib.use('Agg') import matplotlib.pyplot as plt nx.draw(cfg.graph) # 画图
> plt.savefig('temp.png') # 保存 ```

二进制文件加载器

我们知道 angr 是高度模块化的，接下来我们就分别来看看这些组成模块，其中用于二进制加载模块称为 CLE。主类为 `cle.loader.Loader`，它导入所有的对象文件并导出一个进程内存的抽象。类 `cle.backends` 是加载器的后端，根据二进制文件类型区分为 `cle.backends.elf`、`cle.backends.pe`、`cle.backends.macho` 等。

首先我们来看加载器的一些常用参

- `auto_load_libs`：是否自动加载主对象文件所依赖的共享库
- `except_missing_libs`：当有共享库没有找到时抛出异常
- `force_load_libs`：强制加载列表指定的共享库，不论其是否被依赖
- `skip_libs`：不加载列表指定的共享库，即使其被依赖
- `custom_ld_path`：可以到列表指定的路径查找共享库

如果希望对某个对象文件单独指定加载参数，可以使用 `main_ops` 和 `lib_opts` 以字典的形式指定参数。一些通用的参数如下：

- `backend`：使用的加载器后端，如："elf", "pe", "mach-o", "ida", "blob" 等
- `custom_arch`：使用的 archinfo.Arch 对象
- `custom_base_addr`：指定对象文件的基址
- `custom_entry_point`：指定对象文件的入口点

举个例子：

```
python angr.Project(main_opts={'backend': 'ida', 'custom_arch': 'i386'}, lib_opts={'libc.so.6': {'backend': 'elf'}})
```

加载对象文件和细分类型如下：

```
>>> for obj in proj.loader.all_objects:
...     print obj
...
<ELF Object true, maps [0x400000:0x60721f]>
<ELF Object libc-2.27.so, maps [0x1000000:0x13bb98f]>
<ELF Object ld-2.27.so, maps [0x2000000:0x22260f7]>
<ELFTLSObject Object cle##tls, maps [0x3000000:0x300d010]>
<ExternObject Object cle##externs, maps [0x4000000:0x4008000]>
<KernelObject Object cle##kernel, maps [0x5000000:0x5008000]>
```

- `proj.loader.main_object`：主对象文件
- `proj.loader.shared_objects`：共享对象文件
- `proj.loader.extern_object`：外部对象文件
- `proj.loader.all_elf_object`：所有 elf 对象文件
- `proj.loader.kernel_object`：内核对象文件

通过对这些对象文件进行操作，可以解析出相关信息：

```
>>> obj = proj.loader.main_object
>>> obj
<ELF Object true, maps [0x400000:0x60721f]>
>>> hex(obj.entry)                          # 入口地址
'0x401370'
>>> hex(obj.min_addr), hex(obj.max_addr)    # 起始地址和结束地址
('0x400000', '0x60721f')
>>> for seg in obj.segments:                # segments
...     print seg
...
<ELFSegment offset=0x0, flags=0x5, filesize=0x5f48, vaddr=0x400000, memsize=0x5f48>
<ELFSegment offset=0x6c30, flags=0x6, filesize=0x450, vaddr=0x606c30, memsize=0x5f0>
>>> for sec in obj.sections:                # sections
...     print sec
...
<Unnamed | offset 0x0, vaddr 0x400000, size 0x0>
<.interp | offset 0x238, vaddr 0x400238, size 0x1c>
<.note.ABI-tag | offset 0x254, vaddr 0x400254, size 0x20>
<.note.gnu.build-id | offset 0x274, vaddr 0x400274, size 0x24>
...etc
```

根据地址查找我们需要的东西：

```
>>> proj.loader.find_object_containing(0x400000)  # 包含指定地址的 object
<ELF Object true, maps [0x400000:0x60721f]>
>>> free = proj.loader.find_symbol('free')        # 根据名字或地址在 project 中查找 symbol
>>> free
<Symbol "free" in libc.so.6 at 0x1083ab0>
>>> free.name                                     # 符号名
u'free'
>>> free.owner_obj                                # 所属 object
<ELF Object libc-2.27.so, maps [0x1000000:0x13bb98f]>
>>> hex(free.rebased_addr)                        # 全局地址空间中的地址
'0x1083ab0'
>>> hex(free.linked_addr)                         # 相对于预链接基址的地址
'0x83ab0'
>>> hex(free.relative_addr)                       # 相对于对象基址的地址
'0x83ab0'
>>> free.is_export                                # 是否为导出符号
True
>>> free.is_import                                # 是否为导入符号
False

>>> obj.find_segment_containing(obj.entry)        # 包含指定地址的 segment
<ELFSegment offset=0x0, flags=0x5, filesize=0x5f48, vaddr=0x400000, memsize=0x5f48>
>>> obj.find_section_containing(obj.entry)        # 包含指定地址的 section
<.text | offset 0x12b0, vaddr 0x4012b0, size 0x33d9>
>>> main_free = obj.get_symbol('free')            # 根据名字在当前 object 中查找 symbol
>>> main_free
<Symbol "free" in true (import)>
>>> main_free.is_export
False
>>> main_free.is_import
True
>>> main_free.resolvedby                          # 从哪个 object 获得解析
<Symbol "free" in libc.so.6 at 0x1083ab0>

>>> hex(obj.linked_base)                          # 预链接的基址
'0x0'
>>> hex(obj.mapped_base)                          # 实际映射的基址
'0x400000'
```

通过 `obj.relocs` 可以查看所有的重定位符号信息，或者通过 `obj.imports` 可以得到一个符号信息的字典：

```
>>> for imp in obj.imports:
...     print imp, obj.imports[imp]
...
strncmp <cle.backends.elf.relocation.amd64.R_X86_64_GLOB_DAT object at 0x7faf8301b110>
lseek <cle.backends.elf.relocation.amd64.R_X86_64_GLOB_DAT object at 0x7faf8301b7d0>
malloc <cle.backends.elf.relocation.amd64.R_X86_64_GLOB_DAT object at 0x7faf8301be10>

>>> obj.imports['free'].symbol                    # 从重定向信息得到导入符号
<Symbol "free" in true (import)>
>>> obj.imports['free'].owner_obj                 # 从重定向信息得到所属的 object
<ELF Object true, maps [0x400000:0x60721f]>
```

这一部分还有个 hooking 机制，用于将共享库中的代码替换为其他的操作。使用函数 `proj.hook(addr, hook)` 和 `proj.hook_symbol(name, hook)` 来做到这一点，其中 `hook` 是一个 SimProcedure 的实例。通过 `.is_hooked` 、 `.unhook` 和 `.hooked_by` 来进行管理： ```python

> stub*func = angr.SIM*PROCEDURES['stubs']['ReturnUnconstrained'] # 获得一个类 stub_func
>
> proj.hook(0x10000, stub*func()) # 使用类的一个实例来 hook proj.is*hooked(0x10000) True proj.hooked_by(0x10000)
>
> proj.hook*symbol('free', stub*func()) 17316528 proj.is*symbol*hooked('free') True proj.is*hooked(17316528) True
> *当然也可以利用装饰器编写自己的 `hook` 函数：* *python @proj.hook(0x20000, length=5) # length 参数可选，表示程序执行完 hook 后跳过几个字节 ... def my*hook(state): ... state.regs.rax = 1 ... proj.is_hooked(0x20000) True ```

求解器引擎

angr 是一个符号执行工具，它通过符号表达式来模拟程序的执行，将程序的输出表示成包含这些符号的逻辑或数学表达式，然后利用约束求解器进行求解。

Z3

angr 使用 z3 作为约束求解器 https://zhuanlan.zhihu.com/p/30548907

https://ericpony.github.io/z3py-tutorial/guide-examples.htm

程序状态

`state.step()` 用于模拟执行的一个 basic block 并返回一个 SimSuccessors 类型的对象，由于符号执行可能产生多个 state，所以该对象的 `.successors` 属性是一个列表，包含了所有可能的 state。

程序状态 state

- 程序状态 state 是一个 SimState 类型的对象， `angr.factory.AngrObjectFactory` 类提供了创建 state 对象的方法：
  - `.blank_state()`：返回一个几乎没有初始化的 state 对象，当访问未初始化的数据时，将返回一个没有约束条件的符号值。
  - `.entry_state()`：从主对象文件的入口点创建一个 state。
  - `.full_init_state()`：与 entry_state() 类似，但执行不是从入口点开始，而是从一个特殊的 SimProcedure 开始，在执行到入口点之前调用必要的初始化函数。
  - `.call_state()`：创建一个准备执行给定函数的 state。

模拟管理器

模拟管理器（Simulation Managers）是 angr 最重要的控制接口，它允许同时对各组状态的符号执行进行控制，同时应用搜索策略来探索程序的状态空间。states 会被整理到 stashes 里，从而进行各种操作。

我

VEX IR 翻译器

angr 使用了 VEX 作为二进制分析的中间表示。VEX IR 是由 Valgrind 项目开发和使用的中间表示，后来这一部分被分离出去作为 libVEX，libVEX 用于将机器码转换成 VEX IR（更多内容参考章节5.2.3）。在 angr 项目中，开发了模块 PyVEX 作为 libVEX 的 Python 包装。当然也对 libVEX 做了一些修改，使其更加适用于程序分析。

一些用法如下：```python

> > import pyvex, archinfo bb = pyvex.IRSB('\xc3', 0x400400, archinfo.ArchAMD64()) # 将一个位于 0x400400 的 AMD64 基本块（\xc3，即 ret）转成 VEX bb.pp() # 打印 IRSB（Intermediate Representation Super Block） IRSB { t0:Ity*I64 t1:Ity*I64 t2:Ity*I64 t3:Ity*I64

00 | ------ IMark(0x400400, 1, 0) ------ 01 | t0 = GET:I64(rsp) 02 | t1 = LDle:I64(t0) 03 | t2 = Add64(t0,0x0000000000000008) 04 | PUT(rsp) = t2 05 | t3 = Sub64(t2,0x0000000000000080) 06 | ====== AbiHint(0xt3, 128, t1) ====== NEXT: PUT(rip) = t1; Ijk_Ret }

> > bb.statements[3] # 表达式 bb.statements[3].pp() t2 = Add64(t0,0x0000000000000008)
>
> > bb.statements[3].data # 数据 bb.statements[3].data.pp() Add64(t0,0x0000000000000008)
>
> > bb.statements[3].data.op # 操作符 'Iop_Add64'
>
> > bb.statements[3].data.args # 参数 [,] bb.statements[3].data.args[0] bb.statements[3].data.args[0].pp() t0
>
> > bb.next # 基本块末尾无条件跳转的目标 bb.next.pp() t1
>
> > bb.jumpkind # 无条件跳转的类型 'Ijk_Ret' ```

扩展工具

由于 angr 强大的静态分析和符号执行能力，我们可以在 angr 之上开发其他的一些工： - angrop：rop 链自动化生成器 - Patcherex：二进制文件自动化 patch 引擎 - Driller：用符号执行增强 AFL 的下一代 fuzzer - Rex：自动化漏洞利用引擎

参考资料

- angr.io
- docs.angr.io
- angr API documentation
- The Art of War:Offensive Techniques in Binary Analysis

Angr在CTF中的应用

- 符号执行 + 约束求解
- 符号执行去扁平化

DEMO 0x1 符号执行去控制流扁平化

- https://paper.tuisec.win/detail/1c3738f895493a2

DEMO 0x2 :符号执行 + 约束求解

- 见例题