介绍

Triton 是一款动态二进制分析框架，它支持符号执行和污点分析，同时提供了 `pintools` 的 `python` 接口，我们可以使用 `python` 来使用 `pintools` 的功能。 `Triton` 支持的架构有 `x86`, `x64`, `AArch64`.

所有相关文件位于

```
https://gitee.com/hac425/data/tree/master/triton
```

安装

首先需要安装依赖

```
sudo apt-get install libz3-dev  libcapstone-dev libboost-dev  libopenmpi-dev
```

然后根据官网教程进行安装

```
$ git clone https://github.com/JonathanSalwan/Triton.git
$ cd Triton
$ mkdir build
$ cd build
$ cmake ..
$ sudo make -j install
```

报错的解决方案

缺少 openmp 库

```
[ 86%] Built target python-triton
[ 87%] Linking CXX executable simplification
../../libtriton/libtriton.so: undefined reference to `omp_get_thread_num'
../../libtriton/libtriton.so: undefined reference to `omp_get_num_threads'
../../libtriton/libtriton.so: undefined reference to `omp_destroy_nest_lock'
../../libtriton/libtriton.so: undefined reference to `omp_set_nest_lock'
../../libtriton/libtriton.so: undefined reference to `omp_get_num_procs'
../../libtriton/libtriton.so: undefined reference to `omp_unset_nest_lock'
../../libtriton/libtriton.so: undefined reference to `GOMP_critical_name_end'
../../libtriton/libtriton.so: undefined reference to `omp_in_parallel'
../../libtriton/libtriton.so: undefined reference to `omp_init_nest_lock'
../../libtriton/libtriton.so: undefined reference to `GOMP_parallel'
../../libtriton/libtriton.so: undefined reference to `omp_set_nested'
../../libtriton/libtriton.so: undefined reference to `GOMP_critical_name_start'
collect2: error: ld returned 1 exit status
```

在 `CMakeLists.txt` 增加编译参数

在 `CMakeLists.txt` 增加编译参数

```
set(CMAKE_C_FLAGS "-fopenmp")
set(CMAKE_CXX_FLAGS "-fopenmp")
```

z3版本太老

如果使用 `ubuntu 16.04` 由于 `apt` 的 `z3` 版本太老，需要下载最新版的 `z3` 进行编译， 然后使用新版的 `z3` 来编译.

```
cmake .. -DZ3_INCLUDE_DIRS="/home/hac425/z3-4.8.4.d6df51951f4c-x64-ubuntu-16.04/include"  -DZ3_LIBRARIES="/home/hac425/z3-4.8.4.d6df51951f4c-x64-ubuntu-16.04/bin/libz3.a"
```

使用介绍

下面以一些使用示例来介绍 `Triton` 的使用， `Triton` 的基本使用流程是提取出指令的字节码和指令的地址，然后传递给 `Triton` 去执行指令，在指令的执行过程中会维持符号量和污点值的传播。

模拟执行

Triton 首先的一个应用场景就是模拟执行，在 Triton 中执行的执行是由我们控制的，污点分析和符号执行都是基于模拟执行实现的。

下面是一个模拟执行的示例

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from __future__ import print_function
from triton import TritonContext, ARCH, Instruction, OPERAND
import sys

# 每一项的结构是 (指令的地址， 指令的字节码)
code = [
    (0x40000, b"\x40\xf6\xee"),      # imul   sil
    (0x40003, b"\x66\xf7\xe9"),      # imul   cx
    (0x40006, b"\x48\xf7\xe9"),      # imul   rcx
    (0x40009, b"\x6b\xc9\x01"),      # imul   ecx,ecx,0x1
    (0x4000c, b"\x0f\xaf\xca"),      # imul   ecx,edx
    (0x4000f, b"\x48\x6b\xd1\x04"),  # imul   rdx,rcx,0x4
    (0x40013, b"\xC6\x00\x01"),      # mov    BYTE PTR [rax],0x1
    (0x40016, b"\x48\x8B\x10"),      # mov    rdx,QWORD PTR [rax]
    (0x40019, b"\xFF\xD0"),          # call   rax
    (0x4001b, b"\xc3"),              # ret
    (0x4001c, b"\x80\x00\x01"),      # add    BYTE PTR [rax],0x1
    (0x4001f, b"\x64\x48\x8B\x03"),  # mov    rax,QWORD PTR fs:[rbx]
]
if __name__ == '__main__':
```

```
    Triton = TritonContext()
    # 首先设置后面需要模拟执行的代码的架构， 这里是 x64 架构
    Triton.setArchitecture(ARCH.X86_64)
    for (addr, opcode) in code:

        # 新建一个指令对象
        inst = Instruction()
        inst.setOpcode(opcode)  # 传递字节码
        inst.setAddress(addr)   # 传递指令的地址

        # 执行指令
        Triton.processing(inst)

        # 打印指令的信息
        print(inst)
        print('    ---------------')
        print('    Is memory read :', inst.isMemoryRead())
        print('    Is memory write:', inst.isMemoryWrite())
        print('    ---------------')
        for op in inst.getOperands():
            print('    Operand:', op)
            if op.getType() == OPERAND.MEM:
                print('    - segment :', op.getSegmentRegister())
                print('    - base    :', op.getBaseRegister())
                print('    - index   :', op.getIndexRegister())
                print('    - scale   :', op.getScale())
                print('    - disp    :', op.getDisplacement())
            print('    ---------------')
        print()
    sys.exit(0)
```

这个脚本的功能是 `code` 列表中的指令，并打印指令的信息。

- 首先需要新建一个 `TritonContext` ， `TritonContext` 用于维护指令执行过程的状态信息，比如寄存器的值，符号量的传播等，后面指令的执行过程中会修改 `TritonContext` 里面的一些状态。
- 然后调用 `setArchitecture` 设置后面处理指令集的架构类型，在这里是 `ARCH.X86_64` 表示的是 `x64` 架构，其他两个可选项分别为: `ARCH.AARCH64` 和 `ARCH.X86` .
- 之后就可以去执行指令了，首先需要用 `Instruction` 类封装每条指令，设置指令的地址和字节码。
- 然后通过 `Triton.processing(inst)` 就可以执行一条指令。
- 同时 `Instruction` 对象里面还有一些与指令相关的信息可以使用，比如是否会读写内存，操作数的类型等，在这个示例中就是简单的打印这些信息。

下面再以 `cmu` 的 `bomb` 题目中 `phase_4` 为实例，加深 `Triton` 执行指令的流程。

首先看看 `phase_4` 的代码逻辑

```
unsigned int __cdecl phase_4(int a1)
{
  unsigned int v2; // [esp+4h] [ebp-14h]
  int v3; // [esp+8h] [ebp-10h]
  unsigned int v4; // [esp+Ch] [ebp-Ch]

  v4 = __readgsdword(0x14u);
  if ( __isoc99_sscanf(a1, "%d %d", &v2, &v3) != 2 || v2 > 0xE )
    explode_bomb();
  if ( func4(v2, 0, 14) != 5 || v3 != 5 )
    explode_bomb();
  return __readgsdword(0x14u) ^ v4;
}
```

要求输入两个数字存放到 v2, v3 , 其中 v3 为 5， v2不能大于 0xe, 之后 v2 会传入 func4， 并且要求 func4 的返回值为 5。这里 v2 的可能取值只有 0xe 次，这里使用 `Triton` 来模拟执行这段代码，然后爆破 `v2` 的解。我们的目标是让 func4 的返回值为 5， 所以只需要在调用 func4 函数前开始模拟执行即可。

调用 func4 的汇编代码如下

```
.text:08048CED                push    0Eh
.text:08048CEF                push    0
.text:08048CF1                push    [ebp+var_14]  # var_14 --> -14
.text:08048CF4                call    func4
.text:08048CF9                add     esp, 10h
.text:08048CFC                cmp     eax, 5
```

`v2` 保存在 `ebp-14` 的位置，在爆破的过程中不断的重新设置 `v2` (`ebp-14`) 即可。

具体代码如下

```
# -*- coding: utf-8 -*-
from __future__ import print_function
from triton import ARCH, TritonContext, Instruction, MODE, MemoryAccess, CPUSIZE
from triton import *
import os
import sys


EBP_ADDR = 0x100000
# 存放参数的地址
ARG_ADDR = 0x200000

Triton = TritonContext()
Triton.setArchitecture(ARCH.X86)

def init_machine():
    Triton.concretizeAllMemory()
    Triton.concretizeAllRegister()
    Triton.clearPathConstraints()
    Triton.setConcreteRegisterValue(Triton.registers.ebp, EBP_ADDR)
```

```
    # 设置栈
    Triton.setConcreteRegisterValue(Triton.registers.ebp, EBP_ADDR)
    Triton.setConcreteRegisterValue(Triton.registers.esp, EBP_ADDR -  0x2000)

    for i in range(2):
        Triton.setConcreteMemoryValue(MemoryAccess(EBP_ADDR - 0x14 + i * 4, CPUSIZE.DWORD),
5)

# 加载 elf 文件到内存
def loadBinary(path):
    import lief
    binary = lief.parse(path)
    phdrs = binary.segments
    for phdr in phdrs:
        size = phdr.physical_size
        vaddr = phdr.virtual_address
        print('[+] Loading 0x%06x - 0x%06x' % (vaddr, vaddr+size))
        Triton.setConcreteMemoryAreaValue(vaddr, phdr.content)
    return

def crack():
    i = 1
    Triton.setConcreteMemoryValue(MemoryAccess(EBP_ADDR - 0x14, CPUSIZE.DWORD), i)
    pc = 0x8048CED
    while pc:

        # x86 指令集的字节码的最大长度为 15
        opcode = Triton.getConcreteMemoryAreaValue(pc, 16)
        instruction = Instruction()
        instruction.setOpcode(opcode)
        instruction.setAddress(pc)
        Triton.processing(instruction)

        if instruction.getAddress() == 0x08048D01:
            print("solve!  answer: %d" %(i))
            break


        if instruction.getAddress() == 0x8048D07:
            pc = 0x8048CED
            i += 1
            # 重置运行时
            init_machine()
            # 再次设置参数
            Triton.setConcreteMemoryValue(MemoryAccess(EBP_ADDR - 0x14, CPUSIZE.DWORD), i)
            continue
        pc = Triton.getConcreteRegisterValue(Triton.registers.eip)
    print('[+] Emulation done.')


if __name__ == '__main__':
    init_machine()
    loadBinary(os.path.join(os.path.dirname(__file__), 'bomb'))
```

```
    crack()
    sys.exit(0)
```

一些 `api` 的解释

```
Triton.setConcreteRegisterValue(Triton.registers.ebp, EBP_ADDR)
设置具体的寄存器值，设置 ebp 为 EBP_ADDR

Triton.setConcreteMemoryValue(MemoryAccess(EBP_ADDR - 0x14, CPUSIZE.DWORD), i)
设置具体的内存值，第一个参数是一个 MemoryAccess 对象，表示一个内存范围，实例化的时候会给出内存的地址和内存的长度， 第二个参数是需要设置的值，设置值的时候会根据架构的情况按大小端设置，比如 x86 就会以小端的方式设置内存值。 这里就是往 EBP_ADDR - 0x14 的位置写入 DWORD（4 字节）的数据，数据的内容为 i ，按照小端的方式存放、

Triton.getConcreteMemoryAreaValue(pc, 16)
获取内存数据，第一个参数是内存的地址，第二个是需要获取的内存数据的长度。这里表示从 pc 出，取出 16 字节的数据。

instruction.getAddress()
获取指令执行的地址


Triton.getConcreteRegisterValue(Triton.registers.eip)
这里可以获取下一条指令的地址，在 Triton 处理完一条指令后会更新 eip 的值为下一条指令的起始地址
```

程序的流程如下：

- 首先 `init_machine` 的作用就是初始化 `TritonContext` ，同时设置`ebp` 和 `esp` 的值，伪造一个栈。因为程序一开始和每次爆破都要保证 `TritonContext` 的一致性。
- 然后使用 `loadBinary` 函数把 `bomb` 二进制文件加载进内存，加载使用了 `lief` 模块。
- 之后调用 `crack` 函数开始暴力破解的过程。`crack`函数的主要流程是在 栈上设置 `v2` 的值 ，然后从 `0x8048CED` 开始执行，当返回值不是 `5` 时（此时会执行到 `0x8048D07`）初始化 `TritonContext` 同时设置栈里面的参数，修改 `pc` 回到 `0x8048CED` 继续爆破，直到求出结果（此时会执行到 `0x08048D01`）为止。

运行输出如下

```
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ /usr/bin/python
/home/hac425/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/src/examples/python/ctf-
writeups/bomb/p4.py
[+] Loading 0x8048034 - 0x8048154
[+] Loading 0x8048154 - 0x8048167
[+] Loading 0x8048000 - 0x804a998
[+] Loading 0x804bf08 - 0x804c3a0
[+] Loading 0x804bf14 - 0x804bffc
[+] Loading 0x8048168 - 0x80481ac
[+] Loading 0x804a3f4 - 0x804a4f8
[+] Loading 0x000000 - 0x000000
[+] Loading 0x804bf08 - 0x804c000
solve!  answer: 10
[+] Emulation done.
```

求出解是 `10` .

污点分析

污点分析通过标记污点源，然后通过在执行指令时进行污点传播，来最终数据的走向。本节以 `crackme_xor` 二进制程序为例来介绍污点分析的使用。

程序的主要功能是把命令行参数传给 `check` 函数去校验， 函数的代码如下：

```
signed __int64 __fastcall check(__int64 a1)
{
  signed int i; // [rsp+14h] [rbp-4h]

  for ( i = 0; i <= 4; ++i )
  {
    if ( ((*(i + a1) - 1) ^ 0x55) != serial[i] )
      return 1LL;
  }
  return 0LL;
}
```

通过分析代码，输入的字符串的长度为 `5` 个字节，然后会对输入进行一些简单的变化然后和 `serial` 数组进行比较。下面我们使用 `Triton` 的污点分析来看看追踪程序对输入内存的访问情况。

脚本如下：

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
from __future__ import print_function
from triton import TritonContext, ARCH, MODE, AST_REPRESENTATION, Instruction, OPERAND
from triton import *
import sys
import os
import lief

# 加载 elf 文件到内存
INPUT_ADDR = 0x100000

RBP_ADDR = 0x600000
RSP_ADDR = RBP_ADDR - 0x200000
def loadBinary(ctx, path):
    binary = lief.parse(path)
    phdrs = binary.segments
    for phdr in phdrs:
        size = phdr.physical_size
        vaddr = phdr.virtual_address
        print('[+] Loading 0x%06x - 0x%06x' % (vaddr, vaddr+size))
        ctx.setConcreteMemoryAreaValue(vaddr, phdr.content)
    return

if __name__ == '__main__':
    ctx = TritonContext()
```

```
    ctx.setArchitecture(ARCH.X86_64)
    ctx.enableMode(MODE.ALIGNED_MEMORY, True)

    loadBinary(ctx, os.path.join(os.path.dirname(__file__), 'crackme_xor'))
    ctx.setAstRepresentationMode(AST_REPRESENTATION.PYTHON)

    pc = 0x0400556
    # 参数是输入字符串的指针
    ctx.setConcreteRegisterValue(ctx.registers.rdi, INPUT_ADDR)

    # 设置栈的值
    ctx.setConcreteRegisterValue(ctx.registers.rsp, RSP_ADDR)
    ctx.setConcreteRegisterValue(ctx.registers.rbp, RBP_ADDR)

    # ctx.taintRegister(ctx.registers.rdi)

    input = "elite\x00"
    ctx.setConcreteMemoryAreaValue(INPUT_ADDR, input)
    ctx.taintMemory(MemoryAccess(INPUT_ADDR, 8))

    while pc != 0x4005B1:
        # Build an instruction
        inst = Instruction()
        opcode = ctx.getConcreteMemoryAreaValue(pc, 16)
        inst.setOpcode(opcode)
        inst.setAddress(pc)

        # 执行指令
        ctx.processing(inst)

        if inst.isTainted():
            # print('[tainted] %s' % (str(inst)))

            if inst.isMemoryRead():
                for op in inst.getOperands():
                    if op.getType() == OPERAND.MEM:
                        print("read:0x{:08x}, size:{}".format(
                            op.getAddress(), op.getSize()))

            if inst.isMemoryWrite():
                for op in inst.getOperands():
                    if op.getType() == OPERAND.MEM:
                        print("write:0x{:08x}, size:{}".format(
                            op.getAddress(), op.getSize()))

        # 取出下一条指令的地址
        pc = ctx.getConcreteRegisterValue(ctx.registers.rip)
    sys.exit(0)
```

这个脚本的作用是打印对参数字符串所在内存的访问情况， 脚本流程如下：

- 程序首先构造好栈帧， 然后把输入字符串存放到 `INPUT_ADDR` 内存处, 同时设置`RDI` 为 `INPUT_ADDR` 因为在 `x64` 下第一个参数通过 `RDI` 寄存器设置。

- 之后把输入字符串所在的内存区域转换为污点源，之后随着指令的执行会执行污点传播过程。
- 通过 `inst.isTainted()` 可以判断该指令的操作数中是否包含污点值，如果指令包含污点值，就把对污点内存的访问情况给打印出来。

脚本的输出如下：

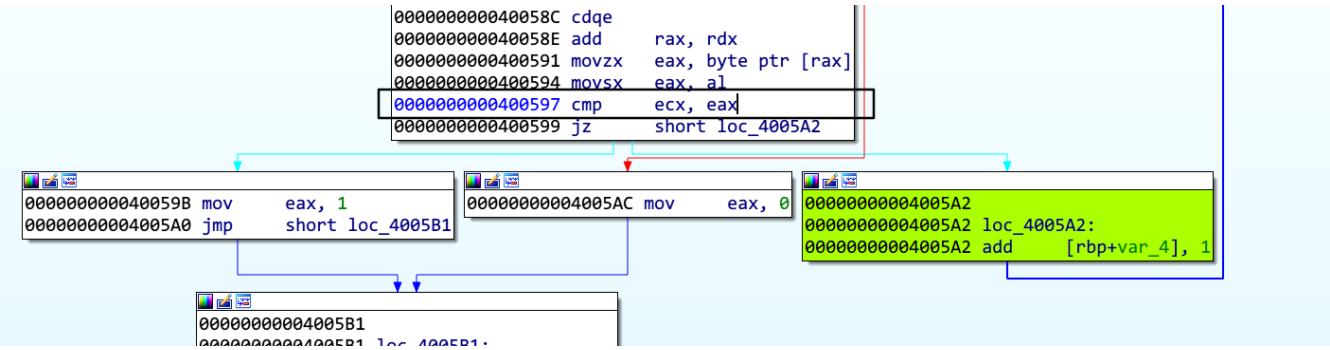
```
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ /usr/bin/python
/home/hac425/pin-2.14-71313-gcc.4.4.7-
linux/source/tools/Triton/src/examples/python/taint/taint.py
[+] Loading 0x400040 - 0x400270
[+] Loading 0x400270 - 0x40028c
[+] Loading 0x400000 - 0x4007f4
[+] Loading 0x600e10 - 0x601048
[+] Loading 0x600e28 - 0x600ff8
[+] Loading 0x40028c - 0x4002ac
[+] Loading 0x4006a4 - 0x4006e0
[+] Loading 0x000000 - 0x000000
[+] Loading 0x600e10 - 0x601000
[+] Loading 0x000000 - 0x000000
read:0x00100000, size:1
read:0x00100001, size:1
read:0x00100002, size:1
read:0x00100003, size:1
read:0x00100004, size:1
```

可以看到成功监控了对输入字符串(`0x00100000` 开始的 5 个字节)的访问。

符号执行

符号执行首先要设置符号量，然后随着指令的执行在 `Triton` 可以维持符号量的传播，然后我们在一些特点的分支出设置约束条件，进而通过符号执行来求出程序的解。

下面还是以 `crackme_xor` 为例介绍一下符号执行的使用。



通过分析可知，在对输入字符串的每个字符进行简单变化后，会把变化后的字符与 `serial` 里面的相应字符进行比较，然后在 `0x400599` 会根据比较的结果决定是否需要跳转。

如果输入的字符串正确的话，程序会走图中染色的分支，所以我们需要在执行完 `0x400597` 指令设置约束条件为 **ZF 寄存器为 1** ，这样就可以跳转到染色的分支进而可以求出程序的解。最终的脚本如下：

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
```

```
from __future__ import print_function
from triton import TritonContext, ARCH, MODE, AST_REPRESENTATION, Instruction, OPERAND
from triton import MemoryAccess,CPUSIZE
import sys
import os
import lief

# 加载 elf 文件到内存


INPUT_ADDR = 0x100000

RBP_ADDR = 0x600000
RSP_ADDR = RBP_ADDR - 0x200000

def loadBinary(ctx, path):
    binary = lief.parse(path)
    phdrs = binary.segments
    for phdr in phdrs:
        size = phdr.physical_size
        vaddr = phdr.virtual_address
        print('[+] Loading 0x%06x - 0x%06x' % (vaddr, vaddr+size))
        ctx.setConcreteMemoryAreaValue(vaddr, phdr.content)
    return

if __name__ == '__main__':
    ctx = TritonContext()
    ctx.setArchitecture(ARCH.X86_64)
    ctx.enableMode(MODE.ALIGNED_MEMORY, True)

    loadBinary(ctx, os.path.join(os.path.dirname(__file__), 'crackme_xor'))
    ctx.setAstRepresentationMode(AST_REPRESENTATION.PYTHON)

    pc = 0x0400556

    # 参数是输入字符串的指针
    ctx.setConcreteRegisterValue(ctx.registers.rdi, INPUT_ADDR)

    # 设置栈的值
    ctx.setConcreteRegisterValue(ctx.registers.rsp, RSP_ADDR)
    ctx.setConcreteRegisterValue(ctx.registers.rbp, RBP_ADDR)


    for index in range(5):
            ctx.setConcreteMemoryValue(MemoryAccess(INPUT_ADDR + index, CPUSIZE.BYTE),
ord('b'))
            ctx.convertMemoryToSymbolicVariable(MemoryAccess(INPUT_ADDR + index,
CPUSIZE.BYTE))

    ast = ctx.getAstContext()
    while pc:
        # Build an instruction
```

```
        inst = Instruction()
        opcode = ctx.getConcreteMemoryAreaValue(pc, 16)
        inst.setOpcode(opcode)
        inst.setAddress(pc)

        # 执行指令
        ctx.processing(inst)

        if inst.getAddress() == 0x400597:
            zf   = ctx.getRegisterAst(ctx.registers.zf)
            cstr  = ast.land([
                        ctx.getPathConstraintsAst(),
                        zf == 1
                    ])
            # 为暂时求出的解具体化
            model = ctx.getModel(cstr)
            for k, v in list(model.items()):
                value = v.getValue()
                ctx.setConcreteVariableValue(ctx.getSymbolicVariableFromId(k), value)

        if inst.getAddress() == 0x4005B1:
            model = ctx.getModel(ctx.getPathConstraintsAst())
            answer = ""
            for k, v in list(model.items()):
                value = v.getValue()
                answer += chr(value)
            print("answer: {}".format(answer))
            break

        # 取出下一条指令的地址
        pc = ctx.getConcreteRegisterValue(ctx.registers.rip)

    sys.exit(0)
```

- 首先使用 `convertMemoryToSymbolicVariable` 将字符串所在的内存转换为符号量
- 然后在运行到 `0x400599` 后， 使用 `ast.land` 把之前搜集到的约束和走染色分支需要的约束集合起来，然后求出每个字符对应的解，并设置符号量为具体的解。
- 然后在 `0x4005B1` 说明输入的所有字符都是正确的，此时打印所有的解即可。

运行结果如下：

```
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ /usr/bin/python
/home/hac425/pin-2.14-71313-gcc.4.4.7-
linux/source/tools/Triton/src/examples/python/taint/sym.py
[+] Loading 0x400040 - 0x400270
[+] Loading 0x400270 - 0x40028c
[+] Loading 0x400000 - 0x4007f4
[+] Loading 0x600e10 - 0x601048
[+] Loading 0x600e28 - 0x600ff8
[+] Loading 0x40028c - 0x4002ac
[+] Loading 0x4006a4 - 0x4006e0
[+] Loading 0x000000 - 0x000000
[+] Loading 0x600e10 - 0x601000
[+] Loading 0x000000 - 0x000000
answer: elite
```

进阶

本节介绍 `triton` 中 `pin` 使用方式以及一些有意思的`demo`。`pin` 是一个二进制插桩工具，可以在程序运行时通过回调函数的机制监控程序的运行，可以用来做代码覆盖率，污点分析等。`triton` 为 `pin` 包装了一层 `python` 的接口，现在我们可以使用 `python` 来运行 `pin`, 非常的方便。

简单使用

下面以一个简单示例来看看如何在 `Triton` 里面使用 `pin`。

```
#!/usr/bin/env python2
## -*- coding: utf-8 -*-

from pintool import *
from triton  import ARCH

count = 0
def mycb(inst):
    global count
    count += 1

def fini():
    print("Instruction count : ", count)

if __name__ == '__main__':
    ctx = getTritonContext()
    ctx.enableSymbolicEngine(False)
    ctx.enableTaintEngine(False)

    # 从程序入口开始插桩
    startAnalysisFromEntry()

    # 在每条指令执行前调用 mycb
```

```
    insertCall(mycb, INSERT_POINT.BEFORE)

    # 程序运行完毕后的回调函数
    insertCall(fini, INSERT_POINT.FINI)
    runProgram()
```

这个脚本的作用是统计被测程序从 `Entry` 开始执行过的指令条数。

- 首先通过 `getTritonContext` 从 `pintools` 里面获取一个 `TritonContext` 实例用于维持程序执行过程中的状态信息，比如污点传播的信息，符号执行的信息等。
- 然后为了节省效率关闭了污点分析和符号执行引擎。
- 然后使用 `startAnalysisFromEntry` 设置 `pin` 在程序入口时进行插桩。
- 通过 `insertCall` 可以在程序执行的过程中设置回调函数，监控程序的执行。比如 `INSERT_POINT.BEFORE` 就是在每次指令执行的时候会调用回调函数，回调函数接收一个参数表示接下来要执行的指令。
  `INSERT_POINT.FINI` 表示在程序执行完毕后调用回调函数。为了统计运行的指令只需要在指令执行 `count += 1` 即可。
- 准备好 `pin` 的参数后，就可以 `runProgram` 运行程序了。

使用了 `pintool` 模块的脚本需要用编译目录下的 `triton` 来加载脚本执行。脚本执行的语法的是

```
sudo ./build/triton 脚本的路径 要运行的目标应用程序 目标应用程序的参数
```

下面对 `crackme_xor` 插桩得到的结果。

```
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ sudo ./build/triton
src/examples/pin/learn/count_inst.py src/samples/crackmes/crackme_xor aaaaa
fail
('Instruction count : ', 59417)
```

在 `Triton` 中我们可以指定 `pin` 插桩的位置，可以用的 `api` 如下

```
startAnalysisFromAddress(addr)
# 从 addr 开始插桩程序

startAnalysisFromEntry()
# 从程序入口，即 start 函数开始分析

startAnalysisFromOffset(integer offset)
# 从程序的偏移处开始分析
```

同时 triton 支持以下几种指令执行过程中的回调

INSERT_POINT.AFTER	**每条指令执行之后，执行回调函数**
INSERT_POINT.BEFORE	每条指令执行之前，执行回调函数
INSERT_POINT.BEFORE_SYMPROC	每条指令符号化处理之前，执行回调函数
INSERT_POINT.FINI	程序运行结束后
INSERT_POINT.ROUTINE_ENTRY	进入函数时
INSERT_POINT.ROUTINE_EXIT	退出函数时
INSERT_POINT.IMAGE_LOAD	镜像加载到内存时
INSERT_POINT.SIGNALS	出现一个信号时
INSERT_POINT.SYSCALL_ENTRY	系统调用执行前
INSERT_POINT.SYSCALL_EXIT	系统调用执行后

详细的信息可以看官方文档

```
https://triton.quarkslab.com/documentation/doxygen/py_INSERT_POINT_page.html
```

特别的，对于指令执行相关的回调，它们的执行顺序是

```
BEFORE_SYMPROC
ir processing，做污点分析与符号执行相关的操作
BEFORE
Pin ctx update， 执行指令， 修改 TritonContext 里面的运行时信息
AFTER
```

污点分析

之前我们通过模拟执行的方式使用了污点分析的功能，这节介绍使用 `pin` 来在程序运行过程中实现污点分析，还是以 `crachme_xor` 为例

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
from triton import ARCH, MemoryAccess, OPERAND
from pintool import *

Triton = getTritonContext()
def cbeforeSymProc(instruction):
    if instruction.getAddress() == 0x400556:
        rdi = getCurrentRegisterValue(Triton.registers.rdi)
        # 内存要对齐
        Triton.taintMemory(MemoryAccess(rdi, 8))


def cafter(inst):
    if inst.isTainted():
```

```
        # print('[tainted] %s' % (str(inst)))

        if inst.isMemoryRead():
            for op in inst.getOperands():
                if op.getType() == OPERAND.MEM:
                    print("read:0x{:08x}, size:{}".format(
                        op.getAddress(), op.getSize()))

        if inst.isMemoryWrite():
            for op in inst.getOperands():
                if op.getType() == OPERAND.MEM:
                    print("write:0x{:08x}, size:{}".format(
                        op.getAddress(), op.getSize()))


if __name__ == '__main__':
    startAnalysisFromSymbol('check')
    insertCall(cbeforeSymProc,  INSERT_POINT.BEFORE_SYMPROC)
    insertCall(cafter,          INSERT_POINT.AFTER)
    runProgram()
```

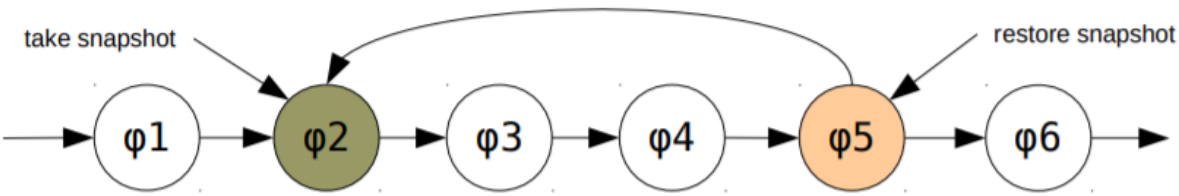
- 首先设置 `pin` 从 `check` 函数开始插桩
- 然后为 `BEFORE_SYMPROC` 和 `AFTER` 设置回调函数。
- `cbeforeSymProc` 函数的作用就是在进入 `check` 函数的时候，设置参数对应的内存区域为污点源，之后程序的执行就可以实现污点传播了。
- `cafter` 的作用是打印对污点内存的访问情况。

执行结果如下：

```
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ sudo ./build/triton
./src/examples/pin/learn/taint.py  ./src/samples/crackmes/crackme_xor elite
[sudo] password for hac425:
read:0x7ffcd5b8d626, size:1
read:0x7ffcd5b8d627, size:1
read:0x7ffcd5b8d628, size:1
read:0x7ffcd5b8d629, size:1
read:0x7ffcd5b8d62a, size:1
Win
```

符号执行

本节还是以 `crackme_xor` 为例介绍基于 `pin` 的符号执行的使用。triton 支持快照功能，我们可以在执行待分析函数之前拍个快照，然后在后面某个时间恢复快照就可以继续从快照点开始执行了，如图所示：



脚本如下:

```
# -*- coding: utf-8 -*-
# sudo ./build/triton ./src/examples/pin/learn/crackme_xor_snapshot.py
./src/samples/crackmes/crackme_xor a
from triton import ARCH
from pintool import *
import sys

password = dict()
cur_char_ptr = None
Triton = getTritonContext()


def csym(instruction):
    global cur_char_ptr

    # print(instruction)

    # 目标函数的入口地址， 第一次执行就拍个快照
    if instruction.getAddress() == 0x400556 and isSnapshotEnabled() == False:
        takeSnapshot()
        return
    if instruction.getAddress() == 0x400574:
        rax = getCurrentRegisterValue(Triton.registers.rax)
        cur_char_ptr = rax  # 每次取字符的地址

        # 如果这个位置已经求出解了，就设置解
        if rax in password:
            setCurrentMemoryValue(rax, password[rax])
        return

    # check 函数的结尾，判断返回值是否满足要求
    if instruction.getAddress() == 0x4005b2:
        rax = getCurrentRegisterValue(Triton.registers.rax)
        # 如果 rax 不是 0 ， 说明还需要继续求解
        if rax != 0:
            # 恢复快照，继续运行
            restoreSnapshot()
        else:
            disableSnapshot()
            # 把解由地址从低往高开始打印
            addrs = password.keys()
            addrs.sort()
            answer = ""
            for addr in addrs:
                c = chr(password[addr])
                answer += c
                print("0x{:08x}: {}".format(addr, c))
            print("answer: {}".format(answer))
        return
    return
```

```
def cafter(instruction):
    global password
    # print(instruction)

    # 0000400574 movzx   eax, byte ptr [rax]
    # 执行完 0x400574 后， rax 里面存放的是刚刚从输入字符串中取出的字符
    # 将取出的字符设置为符号量， 后面用来求解
    if instruction.getAddress() == 0x400574:
        var = Triton.convertRegisterToSymbolicVariable(Triton.registers.rax)
        return

    # 400597 cmp     ecx, eax
    # 开始求解约束
    if instruction.getAddress() == 0x400597:
        astCtxt = Triton.getAstContext()
        zf = Triton.getRegisterAst(Triton.registers.zf)
        # 如果正确的话，需要 zf == 1， 增加约束
        cstr = astCtxt.land([
            Triton.getPathConstraintsAst(),
            zf == 1
        ])
        models = Triton.getModel(cstr)
        for k, v in list(models.items()):
            # 把计算的结果保存
            password.update({cur_char_ptr: v.getValue()})
        return
    return


def fini():
    print('[+] Analysis done!')
    return

if __name__ == '__main__':
    setupImageWhitelist(['crackme_xor'])
    startAnalysisFromAddress(0x0400556)
    insertCall(cafter,  INSERT_POINT.AFTER)
    insertCall(csym,    INSERT_POINT.BEFORE_SYMPROC)

    insertCall(fini,    INSERT_POINT.FINI)
    runProgram()
```

脚本的流程如下

- 通过 `setupImageWhitelist` 设置分析镜像的白名单，减少程序的运行时间。
- 通过 `startAnalysisFromAddress` 设置 `pin` 从 `0x0400556` （即 check函数的入口）分析。
- 然后设置了几个回调。

在程序第一次进入 `0x400556` 时使用 `takeSnapshot` 拍摄一个快照， 然后在 `0x0400597` 这个位置设置约束条件不断求出解，最后在函数执行完毕后检查返回值，如果返回值不为 `0` 说明解还没有完全求出，那么恢复快照继续去求解。

脚本运行如下：

```
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ sudo ./build/triton
./src/examples/pin/learn/crackme_xor_snapshot.py ./src/samples/crackmes/crackme_xor a
0x7ffc3471661f: e
0x7ffc34716620: l
0x7ffc34716621: i
0x7ffc34716622: t
0x7ffc34716623: e
answer: elite
Win
[+] Analysis done!
```

除了手动设置约束外，我们还可以基于分支指令的约束来不断的求出解，如下所示

```
#!/usr/bin/env python2
## -*- coding: utf-8 -*-
## sudo ./build/triton ./src/examples/pin/learn/path_constraints.py
./src/samples/crackmes/crackme_xor a
##  [+] 10 bytes tainted from the argv[1] (0x7ffd4a50c60e) pointer

from triton  import *
from pintool import *

TAINTING_SIZE = 10

Triton = getTritonContext()


def tainting(threadId):
    rdi = getCurrentRegisterValue(Triton.registers.rdi) # argc
    rsi = getCurrentRegisterValue(Triton.registers.rsi) # argv

    # 将 argv 的最后一个参数设置为符号量
    while rdi > 1:
        argv = getCurrentMemoryValue(rsi + ((rdi-1) * CPUSIZE.QWORD), CPUSIZE.QWORD)
        offset = 0
        while offset != TAINTING_SIZE:
            Triton.taintMemory(argv + offset)
            concreteValue = getCurrentMemoryValue(argv + offset)
            Triton.setConcreteMemoryValue(argv + offset, concreteValue)
            Triton.convertMemoryToSymbolicVariable(MemoryAccess(argv + offset,
CPUSIZE.BYTE))
            offset += 1
        print('[+] %02d bytes tainted from the argv[%d] (%#x) pointer' %(offset, rdi-1,
argv))
        rdi -= 1

    return


def fini():
```

```
    pco = Triton.getPathConstraints()
    astCtxt = Triton.getAstContext()
    for pc in pco:
        if pc.isMultipleBranches():
            b1   =  pc.getBranchConstraints()[0]['constraint']
            b2   =  pc.getBranchConstraints()[1]['constraint']
            seed = list()

            # Branch 1
            models  = Triton.getModel(b1)
            for k, v in list(models.items()):
                seed.append(v)

            # Branch 2
            models  = Triton.getModel(b2)
            for k, v in list(models.items()):
                seed.append(v)

            if seed:
                print('进入分支B1的要求: %s (%c)  |  进入分支B2的要求: %s (%c)' %(seed[0],
chr(seed[0].getValue()), seed[1], chr(seed[1].getValue())))
    return

if __name__ == '__main__':
    # Start the symbolic analysis from the 'main' function
    startAnalysisFromSymbol('main')

    # Align the memory
    Triton.enableMode(MODE.ALIGNED_MEMORY, True)

    # Only perform the symbolic execution on the target binary
    setupImageWhitelist(['crackme_xor'])

    # Add callbacks
    insertCall(tainting, INSERT_POINT.ROUTINE_ENTRY, 'main')
    insertCall(fini,     INSERT_POINT.FINI)

    # Run the instrumentation - Never returns
    runProgram()
```

这个脚本的作用是在进入 `main` 函数前将命令行参数设置为符号量， 然后在程序执行完毕后，对搜集到的约束条件进行遍历， 对其中的分支指令，对每种分支的约束进行求解，然后打印进入每条分支需要的值。

运行结果如下：

```
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ sudo ./build/triton ./src/examples/pin/learn/path_constraints.py ./src/samples/crackmes/crackme_xor a
[+] 10 bytes tainted from the argv[1] (0x7ffc18aac623) pointer
fail
进入分支B1的要求: SymVar_0:8 = 0x65 (e)  |  进入分支B2的要求: SymVar_0:8 = 0x0 ()
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ sudo ./build/triton ./src/examples/pin/learn/path_constraints.py ./src/samples/crackmes/crackme_xor e
[+] 10 bytes tainted from the argv[1] (0x7ffe42753623) pointer
fail
进入分支B1的要求: SymVar_0:8 = 0x65 (e)  |  进入分支B2的要求: SymVar_0:8 = 0x0 ()
进入分支B1的要求: SymVar_1:8 = 0x6C (l)  |  进入分支B2的要求: SymVar_1:8 = 0x0 ()
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ sudo ./build/triton ./src/examples/pin/learn/path_constraints.py ./src/samples/crackmes/crackme_xor el
[+] 10 bytes tainted from the argv[1] (0x7fffb4035621) pointer
fail
进入分支B1的要求: SymVar_0:8 = 0x65 (e)  |  进入分支B2的要求: SymVar_0:8 = 0x0 ()
进入分支B1的要求: SymVar_1:8 = 0x6C (l)  |  进入分支B2的要求: SymVar_1:8 = 0x0 ()
进入分支B1的要求: SymVar_2:8 = 0x69 (i)  |  进入分支B2的要求: SymVar_2:8 = 0x0 ()
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ sudo ./build/triton ./src/examples/pin/learn/path_constraints.py ./src/samples/crackmes/crackme_xor eli
[+] 10 bytes tainted from the argv[1] (0x7ffe8bebf61f) pointer
fail
进入分支B1的要求: SymVar_0:8 = 0x65 (e)  |  进入分支B2的要求: SymVar_0:8 = 0x0 ()
进入分支B1的要求: SymVar_1:8 = 0x6C (l)  |  进入分支B2的要求: SymVar_1:8 = 0x0 ()
进入分支B1的要求: SymVar_2:8 = 0x69 (i)  |  进入分支B2的要求: SymVar_2:8 = 0x0 ()
进入分支B1的要求: SymVar_3:8 = 0x74 (t)  |  进入分支B2的要求: SymVar_3:8 = 0x0 ()
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ sudo ./build/triton ./src/examples/pin/learn/path_constraints.py ./src/samples/crackmes/crackme_xor elit
[+] 10 bytes tainted from the argv[1] (0x7fff3998b61d) pointer
fail
进入分支B1的要求: SymVar_0:8 = 0x65 (e)  |  进入分支B2的要求: SymVar_0:8 = 0x0 ()
进入分支B1的要求: SymVar_1:8 = 0x6C (l)  |  进入分支B2的要求: SymVar_1:8 = 0x0 ()
进入分支B1的要求: SymVar_2:8 = 0x69 (i)  |  进入分支B2的要求: SymVar_2:8 = 0x0 ()
进入分支B1的要求: SymVar_3:8 = 0x74 (t)  |  进入分支B2的要求: SymVar_3:8 = 0x0 ()
进入分支B1的要求: SymVar_4:8 = 0x65 (e)  |  进入分支B2的要求: SymVar_4:8 = 0x0 ()
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ sudo ./build/triton ./src/examples/pin/learn/path_constraints.py ./src/samples/crackmes/crackme_xor elite
[+] 10 bytes tainted from the argv[1] (0x7ffcd78c361b) pointer
Win
进入分支B1的要求: SymVar_0:8 = 0x65 (e)  |  进入分支B2的要求: SymVar_0:8 = 0x0 ()
进入分支B1的要求: SymVar_1:8 = 0x6C (l)  |  进入分支B2的要求: SymVar_1:8 = 0x0 ()
进入分支B1的要求: SymVar_2:8 = 0x69 (i)  |  进入分支B2的要求: SymVar_2:8 = 0x0 ()
进入分支B1的要求: SymVar_3:8 = 0x74 (t)  |  进入分支B2的要求: SymVar_3:8 = 0x0 ()
进入分支B1的要求: SymVar_4:8 = 0x65 (e)  |  进入分支B2的要求: SymVar_4:8 = 0x0 ()
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$
```

参考

https://triton.quarkslab.com/documentation/doxygen/#install_sec

https://github.com/JonathanSalwan/Triton/tree/master/src/examples/python

https://github.com/JonathanSalwan/Triton/tree/master/src/examples/pin

https://0x48.pw/2017/04/02/0x30/