

介绍

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。

相关资源位于

<https://gitee.com/hac425/data/tree/master/triton/learn>

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)

    # ■■■■■■■■■■
    startAnalysisFromEntry()

    # ■■■■■■■■■■ mycb
    insertCall(mycb, INSERT_POINT.BEFORE)

    # ■■■■■■■■■■
    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 ■■■■■ ■■■■■■■■■■ ■■■■■■■■■■
```

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 s
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)
# ■ addr ■■■■■■■■

startAnalysisFromEntry()
# ■■■■■■, ■ start ■■■■■■■■

startAnalysisFromOffset(integer offset)
# ■■■■■■■■■■■■
```

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, ■■■■■■■■■■■■■■■■■■■■
BEFORE
Pin ctx update■ ■■■■■■ ■■ TritonContext ■■■■■■■■■■
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)
        # ■■■■■■
        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()))
```

- 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。

```

hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ sudo ./build/triton ./src/examples/pin/learn/taint.py ./s
[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

```

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：



```
# -*- 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)

    # ■■■■■■■■■■ ■■■■■■■■■■

    if instruction.getAddress() == 0x400556 and isSnapshotEnabled() == False:
        takeSnapshot()
        return

    if instruction.getAddress() == 0x400574:
        rax = getCurrentRegisterValue(Triton.registers.rax)
        cur_char_ptr = rax # ■■■■■■■■■■

    # ■■■■■■■■■■■■■■■■■■■■

    if rax in password:
        setCurrentMemoryValue(rax, password[rax])

    return
```

```

# check [REDACTED], [REDACTED]
if instruction.getAddress() == 0x4005b2:
    rax = getCurrentRegisterValue(Triton.registers.rax)
    # [REDACTED] rax [REDACTED] 0 [REDACTED]
    if rax != 0:
        # [REDACTED]
        restoreSnapshot()
    else:
        disableSnapshot()
        # [REDACTED]
        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]
    # [REDACTED] 0x400574 [REDACTED] rax [REDACTED]
    # [REDACTED]
    if instruction.getAddress() == 0x400574:
        var = Triton.convertRegisterToSymbolicVariable(Triton.registers.rax)
        return

    # 400597 cmp        ecx, eax
    # [REDACTED]
    if instruction.getAddress() == 0x400597:
        astCtxt = Triton.getAstContext()
        zf = Triton.getRegisterAst(Triton.registers.zf)
        # [REDACTED] zf == 1 [REDACTED]
        cstr = astCtxt.land([
            Triton.getPathConstraintsAst(),
            zf == 1
        ])
        models = Triton.getModel(cstr)
        for k, v in list(models.items()):
            # [REDACTED]
            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_s
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  
  
    # ■ argv ■■■■■■■■■■■■■■  
    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('■■■■B1■■■: %s (%c) | ■■■■B2■■■: %s (%c)' %(seed[0], chr(seed[0].getValue()), seed[1], chr(seed[1]).
    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 (1) | 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 (1) | 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 (1) | 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 (1) | 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 (1) | 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/>

triton.rar (0.09 MB) [下载附件](#)

点击收藏 | 0 关注 | 1

[上一篇：Uber Bug Bounty：将...](#) [下一篇：高级ROP ret2dl runt...](#)

1. 0 条回复

- 动手手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)