Triton 学习

hackedbylh / 2019-05-13 08:20:00 / 浏览数 4300 安全技术 二进制安全 顶(1) 踩(1)

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.
```

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

# ■■■■■■■■ (■■■■■■■ ■■■■■■■)
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()
   # ■■■■■■■■■■■■■■■■■■■■ ■■■■ x64 ■■
   Triton.setArchitecture(ARCH.X86_64)
   for (addr, opcode) in code:

       # ■■■■■■■■■
       inst = Instruction()
       inst.setOpcode(opcode)  # ■■■■■■
       inst.setAddress(addr)   # ■■■■■■■■

       # ■■■■■
       Triton.processing(inst)

       # ■■■■■■■■
       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
# ■■■■■■■
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)

   # ■■■
   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)

# ■■ elf ■■■■■
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 ■■■■■■■■■■■■■■■ 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
            # ■■■■■■
            init_machine()
            # ■■■■■■■
            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)
■■■■■■■■■■■■■■ ebp ■ EBP_ADDR

Triton.setConcreteMemoryValue(MemoryAccess(EBP_ADDR - 0x14, CPUSIZE.DWORD), i)
■■■■■■■■■■■■■■■■■■■■ MemoryAccess ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ x

Triton.getConcreteMemoryAreaValue(pc, 16)
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ pc ■■■■ 16 ■■■■■■■

instruction.getAddress()
■■■■■■■■■■


Triton.getConcreteRegisterValue(Triton.registers.eip)
■■■■■■■■■■■■■■■■■■■ Triton ■■■■■■■■■■■■■ eip ■■■■■■■■■■■■■■■
```

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-linu
[+] 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

# ■■ elf ■■■■■
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
   # ■■■■■■■■■■■■
   ctx.setConcreteRegisterValue(ctx.registers.rdi, INPUT_ADDR)
```

```
    # ■■■■■
    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)

        # ■■■■
        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()))

        # ■■■■■■■■■■
        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：

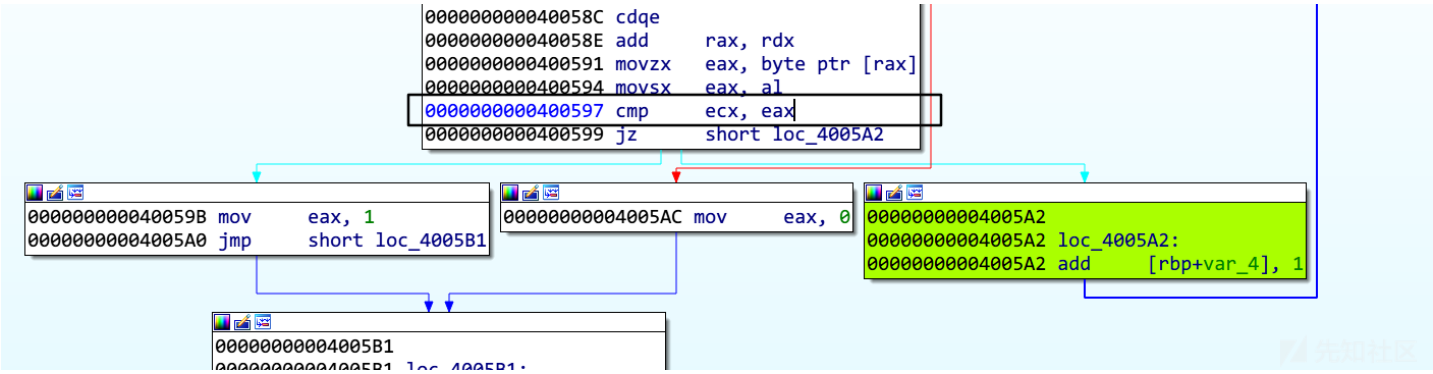
```
hac425@ubuntu:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton$ /usr/bin/python /home/hac425/pin-2.14-71313-gcc.4.4.7-linu
[+] Loading 0x400040 - 0x400270
[+] Loading 0x400270 - 0x40028c
[+] Loading 0x400000 - 0x4007f4
[+] Loading 0x600e10 - 0x601048
[+] Loading 0x600e28 - 0x600ff8
[+] Loading 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

# ■■ elf ■■■■■


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

   # ■■■■■■■■■■■■
   ctx.setConcreteRegisterValue(ctx.registers.rdi, INPUT_ADDR)

   # ■■■■■
   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)

        # ■■■■
        ctx.processing(inst)

        if inst.getAddress() == 0x400597:
            zf   = ctx.getRegisterAst(ctx.registers.zf)
            cstr  = ast.land([
                        ctx.getPathConstraintsAst(),
                        zf == 1
                    ])
            # ■■■■■■■■■■■
            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

        # ■■■■■■■■■■■
        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-linu
[+] 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
```

参考

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

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

triton.rar (0.086 MB) 下载附件
点击收藏 | 0 关注 | 1

上一篇：Intigriti XSS Cha... 下一篇：PDF双重释放漏洞CVE-2018...

1. 0 条回复
   - 动动手指，沙发就是你的了！