MAKE TRITON FIND BUG

OUTLINE

- Triton Introduction
- Triton Basic Usage
- Code Coverage Practice
- Security Model
- Find Bug

TRITON

- Triton is a concolic execution framework
  - Pintool
  - Stand alone
- It provides advanced classes to improve dynamic binary analysis (DBA)
  - Symbolic execution engine
  - SMT semantics representation
  - Interface with SMT Solver
  - Taint analysis engine
  - Snapshot engine (only for pin)
  - API and Python bindings

VM ENVIRONMENT

- Environment
  - Ubuntu 14.04
    - Docker + Triton
    - Vxcon/vxcon

- sudo docker exec –it vxcon /bin/bash
- Inside the docker
  - Pin
  - Triton
    - /home/vxcon/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton

TRITON PROJECT: CODE TRAVERSER

- To demonstrate the capability, we will build a small project - Code Traverser
  - Symbolize the general input functions and argument
  - Automatic identify every(most) branches in the program
  - Calculate the input that can reaches to the other side of branch
  - Check Security Property

TRITON IR

- As a dynamic analysis system, Triton receive the runtime trace
  - Online: If the runtime analysis backend is utilized, e.g. PIN
  - Offline: Only the runtime trace is available, e.g. GDB dump

- Triton translates every instructions into IRs, which are the symbolic expression

TRITON: INSTRUMENT AND IRS

- Implement our first Triton tool
  - <source code>
  - Triton/src/examples/pin/ir.py

```
if __name__ == '__main__':
    # Set arch
    setArchitecture(ARCH.X86_64)

    # Start JIT at the entry point
    startAnalysisFromEntry()

    # Add callback
    insertCall(mycb, INSERT_POINT.BEFORE)

    # Run Program
    runProgram()
```

```
#!/usr/bin/env python2
## -*- coding: utf-8 -*-
import sys
from pintool import *
from triton import *

def mycb(inst):
    print inst
    for expr in  inst.getSymbolicExpressions():
        print expr
    print
    return
```

PRACTICE: TRITON SYMBOLIC EXPRESSION

Tracking Triton's Symbolic Expression

- Write a simple program and trace expression

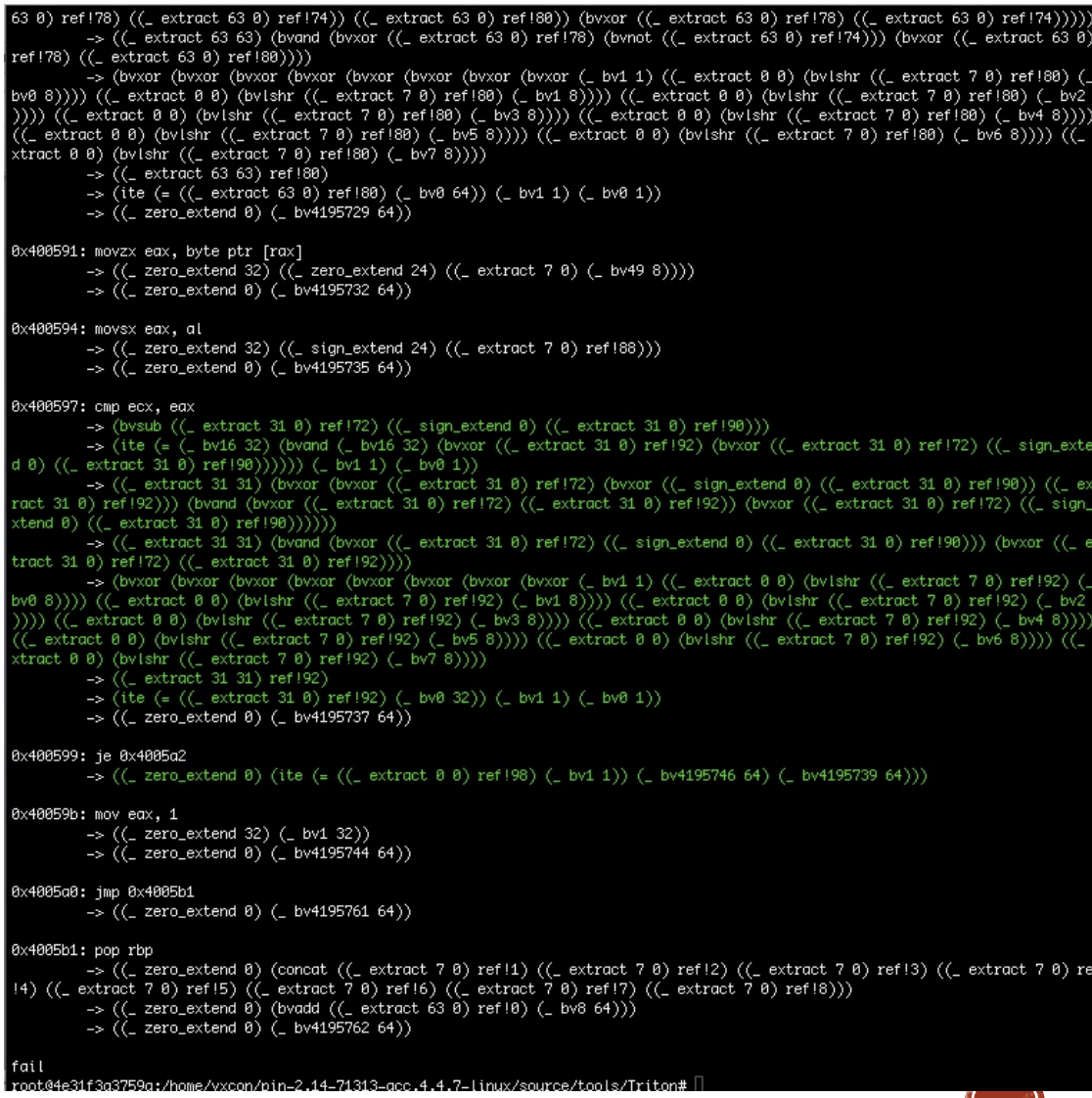
- Compare between startAnalysisFromEntry and startAnalysisFromSymbol("main")

TRITON TAINT

- Triton is the concolic execution system
  - Concolic – Unrelated information is represent as concrete value
    - Unrelated information – un-taint register/memory
    - Concrete value – the real value in the runtime without symbolic expression

TAINT API

- taintMemory
- isMemoryTainted
- untaintMemory
- getTaintedMemory
- taintRegister
- isRegisterTainted
- untaintRegister
- getTaintedRegisters



PRACTICE: TRITON TAINT

Triton Taint

- Sample Code
  - <code path>
  - src/examples/pin/runtime_memory_tainting.py

$ ./triton ./src/examples/pin/runtime_memory_tainting.py ./src/samples/crackmes/crackme_xor a

```
#!/usr/bin/env python2
## -*- coding: utf-8 -*-
import sys
from pintool import *
from triton import *

if __name__ == '__main__':
    setArchitecture(ARCH.X86_64)
    startAnalysisFromSymbol('check')
    insertCall(cbeforeSymProc,  INSERT_POINT.BEFORE_SYMPROC)
    insertCall(cafter,          INSERT_POINT.AFTER)
    runProgram()
```

TRITON TAINT

```
def cbeforeSymProc(instruction):
    if instruction.getAddress() == 0x400574:
        rax = getCurrentRegisterValue(REG.RAX)
        taintMemory(rax)
```

```
def cafter(instruction):
    print '%#x: %s' %(instruction.getAddress(),
    instruction.getDisassembly())
    for se in instruction.getSymbolicExpressions():
        if se.isTainted() == True:
            print '\t -> %s%s%s' %(GREEN, se.getAst(), ENDC)
        else:
            print '\t -> %s' %(se.getAst())
    print
```

CONTROL FLOW AND DATA FLOW

Break the taint

- An example about control flow break taint
  - /home/vxcon/vxcon/cfg

```
char transfer(char in)
{
    char ret;
    if(in=='a')
        ret = 'a';
    if(in=='b')
        ret = 'b';
    ...
    return ret
}
```

TRITON: SYMBOLIZE AND MODEL

- The fundamental function of symbolic/concolic framework
  - Symbolize register and memory
  - Get the model with the given constraints

- Code
  - compare.c
  - compare

```
int main(int argc, char* argv[])
{
    if( *((int*)argv[1]) == 0x61616161 )
    {
        printf("pass\n");
    }else
    {
        printf("fails\n");
    }
}
```

FIND COMPARISON INSTRUCTION

objdump -M intel-mnemonic -d /compare_char

```
000000000040052d <main>:
 40052d:    55              push   rbp
 40052e:    48 89 e5          mov    rbp,rsp
 400531:    48 83 ec 10         sub    rsp,0x10
 400535:    89 7d fc          mov    DWORD PTR [rbp-0x4],edi
 400538:    48 89 75 f0         mov    QWORD PTR [rbp-0x10],rsi
 40053c:    48 8b 45 f0         mov    rax,QWORD PTR [rbp-0x10]
 400540:    48 83 c0 08         add    rax,0x8
 400544:    48 8b 00          mov    rax,QWORD PTR [rax]
 400547:    8b 00           mov    eax,DWORD PTR [rax]
 400549:    3d 61 61 61 61        cmp    eax,0x61616161
 40054e:    75 0c           jne    40055c <main+0x2f>
 400550:    bf f4 05 40 00        mov    edi,0x4005f4
 400555:    e8 b6 fe ff ff        call   400410 <puts@plt>
 40055a:    eb 0a           jmp    400566 <main+0x39>
 40055c:    bf f9 05 40 00        mov    edi,0x4005f9
 400561:    e8 aa fe ff ff        call   400410 <puts@plt>
 400566:    c9              leave
 400567:    c3              ret
```

SIMPLE SYMBOLIC SOLVER

Solve Constraint

- Compare.py
  - Instrument on 0x400547
  - Change register to Symbolic Variable
  - Get the symbolic AST of register
  - Craft the AST
  - Get a model

```
def inst_cb(inst):
    if inst.getAddress() == 0x400547:
        convertRegisterToSymbolicVariable(REG.EAX)
        rax    = getSymbolicRegisterId(REG.EAX)
        raxExpr =  getAstFromId(rax)
        expr    = ast.assert_( raxExpr == 0x61616161 )
        models  = getModel(expr)
        print models
        for k, v in models.items():
            print k, v
```

TRITON: SYMBOLIZE ARGUMENT

- In previous, the simplest solution is implemented
- But
  - Only focus on the last comparison inst
  - The data may be copied to other place
  - The calculation may be more complex
- Guideline
  - Symbolize variables as early as possible
    - Argument
    - Function for reading input
- Where is the program's argument?

WHERE IS YOUR ARGUMENTS?

```
Int main(int argc, char** argv)
{
   …
}
```

reg	value
RDI	2
RSI	0x7ffc80cd3228 L

addr	value
0x7ffc80cd3228	0x7ffc80cd36e4
0x7ffc80cd3230	0x7ffc80cd36f2

addr						
0x7ffc80cd36e4	a	.	o	u	T	
0x7ffc80cd36f2	a	r	g	l		

SYMBOLIZE ARGUMENT

Symbolize Argument

Address of Main, can be found by objdump

```
if inst.getAddress() == 0x40052d:

        rdi = getCurrentRegisterValue(REG.RDI) # argc
        rsi = getCurrentRegisterValue(REG.RSI) # argv

        for i in xrange(0, rdi ):
            tmp = getConcreteRegisterValue(REG.RSI)+i*CPUSIZE.QWORD
            argv_addr= getCurrentMemoryValue(MemoryAccess(tmp , CPUSIZE.QWORD))
            index = 0
            while True:
                memv = getCurrentMemoryValue(val+index)
                if memv == 0: break
                convertMemoryToSymbolicVariable(MemoryAccess(val+index, CPUSIZE.BYTE))
                y += 1
```

TRITON: HOOK FUNCTIONS

- Hook function is a handy function
  - Intercept the control flow when the function be triggered
- Use Scenario
  - Hook functions to symbolize arguments or return value
  - Customize symbolic rule for certain functions
    - Avoid tracking too many instructions
    - Avoid unsupported instructions

TRITON: SYMBOLIZE FUNCTION'S ARGV OR RETURN VALUE

- Triton supports function hook
  - Callbacks before and after function called

```
insertCall(hook_scanf, INSERT_POINT.ROUTINE_ENTRY, "scanf")
insertCall(hook_scanf_after, INSERT_POINT.ROUTINE_EXIT, "scanf")
insertCall(hook_get_char, INSERT_POINT.ROUTINE_EXIT, "getchar")
```

- Calling Convention
  - Where is the argument stored ?
  - Where is the returned value ?

TRITON: SYMBOLIZE FUNCTION'S ARGV OR RETURN VALUE

Hook & Symbolize

- Practice
  - Symbolize char returned by getc
  - Symbolize string returned by scanf
  - compare_scanf

SYMBOLIZE FOR SCANF()(1)

```
def hook_scanf(threadId):
    rdi = getCurrentRegisterValue(REG.RDI)
    rsi = getCurrentRegisterValue(REG.RSI)
    print "HOOK SCANF"

    ctx = {}
    ctx['context'] = getCurrentMemoryValue(getCurrentRegisterValue(REG.RBP), 16)
    ctx['format_string'] = getString(rdi)
    ctx['buffer'] = rsi

    scanf_context[getCurrentMemoryValue(getCurrentRegisterValue(REG.RBP), 16)] = ctx
```

```
def hook_scanf_after(threadId):
    print "HOOK SCANF AFTER"
    ctx = scanf_context[getCurrentMemoryValue(getCurrentRegisterValue(REG.RBP), 16)]

    if ctx['format_string'] == "%s":
        print "buffer", getString(ctx['buffer'])
        cur_in = getString(ctx['buffer'])
        symbolize_string(ctx['buffer'])
```

SYMBOLIZE FOR SCANF()(2)

```
def symbolize_string(mem_adr):
    cur = mem_adr
    string = ""
    while True:
        print hex(cur)
        c = getCurrentMemoryValue(cur)
        if c == 0: break
        string += chr(c)
        convertMemoryToSymbolicVariable(MemoryAccess(cur, CPUSIZE.BYTE))
        cur += 1
    return string
```

```
def getString(mem_adr):
    cur = mem_adr
    string = ""
    while True:
        c = getCurrentMemoryValue(cur)
        if c == 0: break
        string += chr(c)
        cur += 1
    return string
```

SYMBOLIZE FOR GETCHAR()

```
def hook_get_char(threadId):
    print hex(getCurrentRegisterValue(REG.RAX))
    convertRegisterToSymbolicVariable(REG.RAX)
```

CODE COVERAGE

- As perspective of bug finding, code coverage is the main measurement factor

TRITON: IDENTIFY BRANCHES

- Branch is the critical factor to the code coverage

- Triton provides APIs to find the branches

```
def mycb(inst):

    if inst.isBranch():
        print "isBranch"

        if inst.isConditionTaken():
            print "isConditionTaken"
```

TRITON: SNAPSHOT

- Snapshot can be used to preserve current program state
- The saved state can be restore afterward

CRACK_ME XOR

- Triton/src/examples/pin/inject_model_with_snapshot.py

```
def csym(instruction):
    # Prologue of the function
    if instruction.getAddress() == 0x400556
    and isSnapshotEnabled() == False:
        takeSnapshot()
        print '[+] Take a snapshot at the'
              'prologue of the function'
        return
```

```
# Epilogue of the function
    if instruction.getAddress() == 0x4005b1:
        rax = getCurrentRegisterValue(REG.RAX)
        # The function returns 0 if the password is valid
        # So, we restore the snapshot until this function
        # returns something else than 0.
        if rax != 0:
            print '[+] Still not the good password. ⇗
            restoreSnapshot()
        else:
            print '[+] Good password found!'
            disableSnapshot()
        return
    return
```

UPDATE SYMBOLIC EXPRESSION

- Expression associates to register and memory can be modify via assignSymbolicExpressionToRegister() and assignSymbolicExpressionToMemory()
- This may break Triton's internal status, avoid this usage.

```
n = newSymbolicExpression( ite( alast==dlva, bv(1, 1), bv(0, 1) ), "test")
assignSymbolicExpressionToRegister( n, REG.ZF)
```

PRACTICE:STRCMP

MY STRCMP

code_coverage_strcmp.c

```
int my_strcmp(char* str1, char* str2)
{
    int i = 0;
    for(i=0;  str1[i]==str2[i] && str1[i]!=0 && str2[i]!=0; i++);
    return str1[i] - str2[i];
}

int main(int argc, char** argv)
{
    if(my_strcmp(“canyoupass", argv[1])==0)
        printf("pass!!\n");
}
```

TRITON WITHOUT PIN

- In some cases, Triton may not correctly sync with pin
- Use triton without pin is more robust, but more complex

TRITON EMULATOR

- Direct use Triton without pin
  - Set instruction
  - processing

```
code = [
    (0x400000, "\x48\xB8\x48\x47\x46\x45\x44\x43\x42\x41"),
    # movabs rax, 0x4142434445464748
    (0x40000a, "\x48\xC7\xC6\x08\x00\x00\x00"),
    # mov    rsi, 0x8
    (0x400011, "\x48\xC7\xC7\x00\x00\x01\x00"),
    # mov    rdi, 0x10000
    (0x400018, "\x48\x89\x84\x77\x34\x12\x00\x00"),
    # mov    QWORD PTR [rdi+rsi*2+0x1234], rax
]
```

```
if __name__ == '__main__':

    # Set the architecture
    setArchitecture(ARCH.X86_64)

    for (addr, opcodes) in code:
        # Build an instruction
        inst = Instruction()

        " Setup opcodes
         nst.setOpcodes(opcodes)

          Setup Address
         nst.setAddress(addr)

          Process everything
         rocessing(inst)

          Display instruction
         rint inst

          Display symbolic expressions
        .or expr in inst.getSymbolicExpressions():
            print '\t', expr

        print
```

ENVIRONMENT MODEL

Environment Model

- Without the help of pin, environment model must be constructed by ourselves

- Emulate the function of loader
  - Parse Binary
  - Memory Layout
  - Shared Library
  - Initial Program State
    - register and memory

```
if __name__ == '__main__':
    # Set the architecture
    setArchitecture(ARCH.X86_64)

    # Parse the binary
    binary = Elf(os.path.join(os.path.dirname(__file__),
    'samples', 'sample_1'))

    # Load the binary
    loadBinary(binary)

    # Perform our own relocations
    makeRelocation(binary)

    # Define a fake stack
    setConcreteRegisterValue(Register(REG.RBP, 0x7fffffff))
    setConcreteRegisterValue(Register(REG.RSP, 0x6fffffff))

    # Let's emulate the binary from the entry point
    emulate(binary.getHeader().getEntry())
```

PARSE BINARY

- ELF Class

```
binary = Elf(os.path.join(os.path.dirname(__file__), 'samples', 'sample_1'))
```

MEMORY LAYOUT

- Executable header
  - The memory address to load the binary

```
def loadBinary(binary):
    # Map the binary into the memory
    raw = binary.getRaw()
    phdrs = binary.getProgramHeaders()
    for phdr in phdrs:
        offset = phdr.getOffset()
        size   = phdr.getFilesz()
        vaddr  = phdr.getVaddr()
        print '[+] Loading 0x%06x - 0x%06x' %(vaddr, vaddr+size)
        setConcreteMemoryAreaValue(vaddr, raw[offset:offset+size])
    return
```

SHARED LIBRARY

- The import function is stored in relocation table
- The shared library is not loaded, therefore imported function should be emulated

```
customRelocation = [
    ('strlen',            strlenHandler,   0x10000000),
    ('printf',            printfHandler,   0x10000001),
    ('__libc_start_main', libcMainHandler, 0x10000002),
]
```

```
def makeRelocation(binary):
    # Perform our own relocations
    symbols = binary.getSymbolsTable()
    relocations = binary.getRelocationTable()
    for rel in relocations:
        symbolName = symbols[rel.getSymidx()].getName()
        symbolRelo = rel.getOffset()
        for crel in customRelocation:
            if symbolName == crel[0]:
                print '[+] Hooking %s' %(symbolName)
                setConcreteMemoryValue(MemoryAccess(symbolRelo,
                CPUSIZE.QWORD, crel[2]))
    return
```

SHARED LIBRARY

- During the execution, redirect the call to the corresponding callback if the imported function called.

```
def hookingHandler():
    pc = getConcreteRegisterValue(REG.RIP)
    for rel in customRelocation:
        if rel[2] == pc:
            # Emulate the routine and the return value
            ret_value = rel[1]()
            concretizeRegister(REG.RAX)
            setConcreteRegisterValue(Register(REG.RAX,
            ret_value))

            # Get the return address
            ret_addr = getConcreteMemoryValue(
                MemoryAccess(
                 getConcreteRegisterValue(REG.RSP),
                 CPUSIZE.QWORD
                 )
            )
```

```
# Hijack RIP to skip the call
            concretizeRegister(REG.RIP)
             setConcreteRegisterValue(
             Register(REG.RIP, ret_addr))

            # Restore RSP (simulate the ret)
            concretizeRegister(REG.RSP)
            setConcreteRegisterValue(
               Register(REG.RSP,
          getConcreteRegisterValue(REG.RSP)+
       CPUSIZE.QWORD))
    return
```

SHARED LIBRARY

- Strlen Handler

```
def strlenHandler():
    print '[+] Strlen hooked'

    # Get arguments
    arg1 = getMemoryString(getConcreteRegisterValue(REG.RDI))

    # Return value
    return len(arg1)
```

- Check src/examples/python/hooking_libc.py

PATH CONSTRAINTS

- To ease the effort to maintain constraint, Triton provides the path constraint API
- Path constraints
  - The branch condition that related to symbolize variable

```
mov ax, 5
mov dx, 5
cmp ax, dx
jne 0x11223344
```

```
mov ax, <symbolize var>
mov dx, 5
cmp ax, dx
jne 0x11223344
```

PATH CONSTRAINTS

- Path constraints can be found as follow

```
if inst.getType() == OPCODE.JNE:
        print inst
        print getFullAstFromId(getSymbolicRegisterId(REG.ZF))
        expr = getAstFromId(getSymbolicRegisterId(REG.ZF))
        if expr.isSymbolized():
             # reverse branch
             c = ast.bvtrue()
             for pc in path_constraint:
                 c = ast.land(c, pc[1]==pc[0])
             c = ast.land(c, expr == (not tmp_zf))
             print c
             print getModels(c, 1)

             print "symbolize constraint"
             path_constraint.append( (tmp_branch_addr, expr, tmp_zf)
```

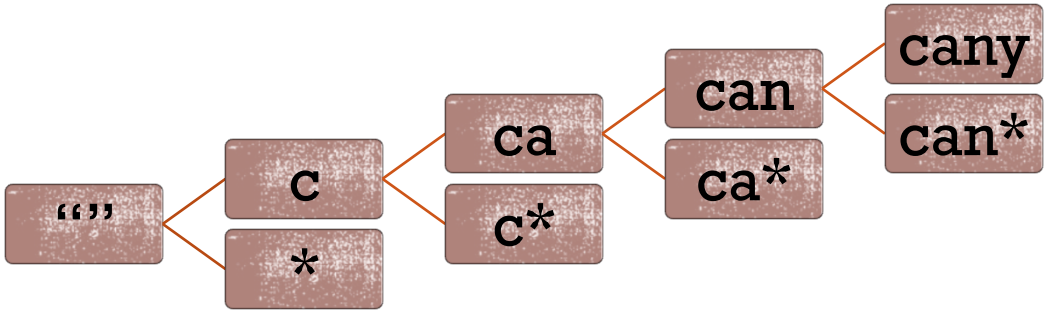
PATH CONSTRAINTS

- You should deal with every kind of branch instructions
- However, Triton provides API for it

```
# Go through the path constraints
    for pc in pco:
        if pc.isMultipleBranches():
            branches = pc.getBranchConstraints()
            for branch in branches:
                    if branch['isTaken'] == False:
                    # Ask for a model
                    models = getModel(assert_(land(previousConstraints,
                                branch['constraint'])))
                    for k, v in models.items():
                        # Get the symbolic variable assigned to the model
                        symVar = getSymbolicVariableFromId(k)
```

PRACTICE: TRAVERSE BRANCH PRACTICE

- Practice: Traverse branch practice



CHECK BUG

- After code coverage can be improved and more code can be reached
- Next security model should be defined and implement to check the existence of vulnerabilities

SECURITY MODEL

- How to define the security model
- Buffer Overflow

- Format String
  - "%n" should not be used

- Use After Free

PRINTF SECURITY PROPERTY

- First argument should never be control by input

```c
#include <stdio.h>

int main(int argc, char** argv)
{
    printf(argv[1]);
}
```

USE TRITON TO TRACK PRINTF VULNERABILITY

Printf Security Check

- Practice
  1. Use previous script to symbolize user's input
  2. Hook printf()
  3. Determine if first argument is symbolic
  4. (Option step) Use solver to check if the first argument is symbolize and can be "%s" or "%n"

TESTING YOUR SCRIPT

```
int main(int argc, char** argv)
{
    char str[100];
    int i;
    for(i=0; argv[1][i] != 0; i++)
    {
        *(char*)(str+i) = argv[1][i]-(unsigned char)3;
    }
    printf(str);
    printf("\nfin\n");
}
```

MEMCPY SECURITY PROPERTY

- Check memcpy's vulnerability

- Easiest approaches
  - Symbolize input
  - Check if the memcpy's size is controllable

- Difficult one
  - Estimate the dst buffer size
  - Check size
  - How to?

Q&A