“Solving for Inputs in a homebrew IL against a homebrew architecture with a homebrew framework to solve a challenge you wrote yourself and already have the solution to.”

Other Working Title: “The Nacho Framework”

By endeavor/rednovae

Setting the Stage :: HSVM

16-bit, big-endian, RISC architecture developed for Haxathon Supremacy, a 2 ½ month-long binary/reversing/exploitation CTF hosted in late 2012.

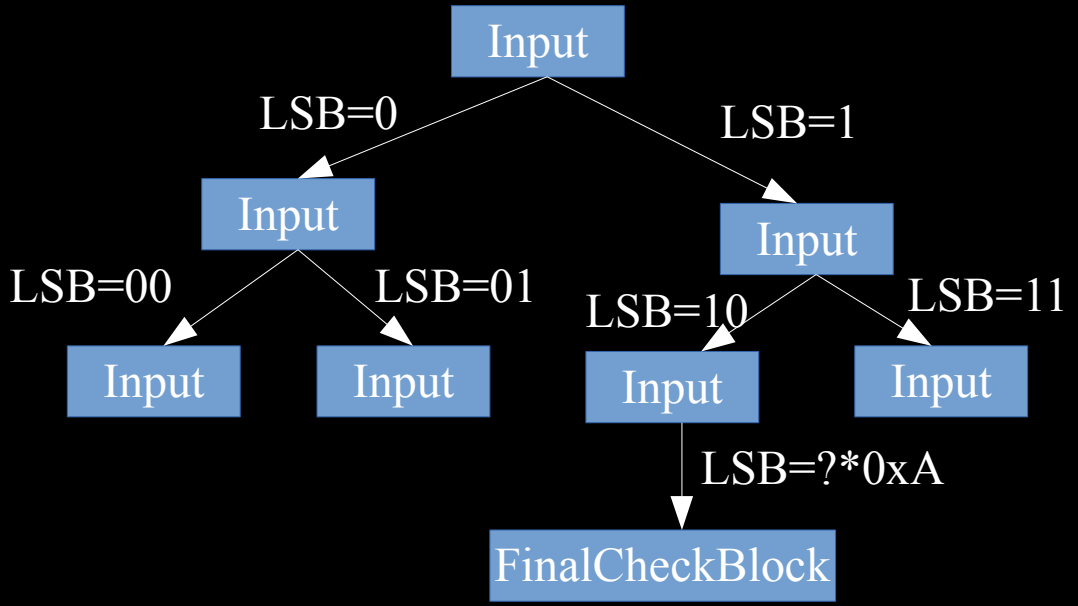
http://github.com/endeav0r/hsvm (the VM, sources to all challenges, solutions, etc.)

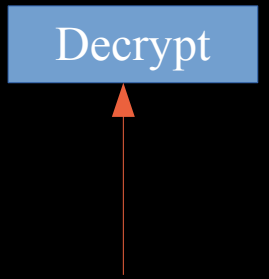
The Instruction Set: add, sub, mul, div, mod, and, or, xor, cmp, je, jne, jl, jle, jg, jge, call, jmp, load, loadb, stor, storb, push, pop, in, out, mov, hlt, nop, syscall

Setting the Stage :: RE101

Crackme written in HSVM for USMA cadet CTF-Team Tryouts.

http://tfpwn.com/re101.html





Everything is xor encrypted, where the key is stored in the parent/preceding block. Confused? See the blog post. This complicates naïve decompilation/static analysis.

Step One :: Queso

Queso is the name of the IL. It operates on, and only on, variables. It is very similar to RREIL

Variables → {variable, constant, array}

Instructions → {comment, assign, store, load, signextend, ite, arithmetic, cmp}

Arithmetic → {add, sub, mul, udiv, umod, and, or, xor, shl, shr}

Cmp → {cmpeq, cmplts, cmples, cmpltu, cmpleu}

Step One :: Queso

Comment(string JustAboutAnything) // ok, not a Variable

Store(DstMem, SrcMem, Address, Value)

Load(SrcMem, Address, Dst)

Assign(Dst, Src)

SignExtend(Dst, Src)

Ite(Dst, Condition, ThanValue, ElseValue)

Arithmetic(Dst, Lhs, Rhs)

Cmp(Dst, Lhs, Rhs)

Step One :: Queso

Bonus!

Queso optionally **embeds extra information from traces**, such as pc or the runtime address used for loads/stores.

Bonus!

Queso serializes to **json and protobuf**. Can your favorite language easily parse json!? Yay!

Step Two :: Tracer

HSVM runs in a small, emulated environment. Really easy, just wrap the execution loop and start logging information.

Because our executions are not in the hundreds of millions or billions of instructions, we can **log everything**. **No need for taint tracking** here to reduce scope.

Really lazy. Logs everything to **json**.

Step Three :: Transform Trace to Queso

This is the tedious boring step.

```
Variable Hsvmt :: load16 (const Variable & mem, const Variable & address, uint64_t trace_address)
{
    Variable loadl8 = Variable("loadl8", 8);
    Variable loadh8 = Variable("loadh8", 8);
    Variable loadl16 = Variable("loadh16", 16);
    Variable loadh16 = Variable("loadh16", 16);
    Variable loadh162 = Variable("loadh162", 16);
    Variable result = Variable("load16", 16);
    Variable address2 = Variable("address2", 16);

    append(new InstructionLoad(mem, address, loadh8, trace_address));
    append(new InstructionAdd(address2, address, Variable(16, 1)));
    append(new InstructionLoad(mem, address2, loadl8, trace_address));

    append(new InstructionAssign(loadl16, loadl8));
    append(new InstructionAssign(loadh16, loadh8));
    append(new InstructionShl(loadh162, loadh16, Variable(16, 8)));
    append(new InstructionOr(result, loadh162, loadl16));

    return result;
}
```

Step Four :: Convert Queso to SMTLIB2

```
endeavor@debian:~$ code/rnpnacho/src/transform -n code/rnpnacho/il.queso -s /tmp/smt2
endeavor@debian:~$ tail /tmp/smt2 -n 32
(assert (= tmp_1990 (ite (= flags_502 #x0000) #b1 #b0)))
(assert (= tmp_1991 (bvxor tmp_1990 #b1)))
(assert (= tmp2_502 (bvadd rip_524 #xffe8)))
(assert (= rip_525 (ite (= tmp_1991 #b1) tmp2_502 rip_524)))
; 0048 loadb r1, r0
(assert (= tmp_1992 (select mem_66040 r0_505)))
(assert (= r1_505 (concat (_ bv0 8) tmp_1992)))
; 004c out r1
(assert (= out_1 ((_ extract 7 0) r1_505)))
; 0050 add r0, 0001
(assert (= r0_506 (bvadd r0_505 #x0001)))
; 0054 loadb r1, r0
(assert (= tmp_1993 (select mem_66040 r0_506)))
(assert (= r1_506 (concat (_ bv0 8) tmp_1993)))
; 0058 cmp r1, 0000
(assert (= flags_503 (bvsub r1_506 #x0000)))
; 005c jne ffe8
(assert (= tmp_1994 (ite (= flags_503 #x0000) #b1 #b0)))
(assert (= tmp_1995 (bvxor tmp_1994 #b1)))
(assert (= tmp2_503 (bvadd rip_525 #xffe8)))
(assert (= rip_526 (ite (= tmp_1995 #b1) tmp2_503 rip_525)))
; 0060 ret
(assert (= loadh8_0 (select mem_66040 rsp_2)))
(assert (= address2_1 (bvadd rsp_2 #x0001)))
(assert (= loadl8_0 (select mem_66040 address2_1)))
(assert (= loadh16_0 (concat (_ bv0 8) loadl8_0)))
(assert (= loadh16_1 (concat (_ bv0 8) loadh8_0)))
(assert (= loadh162_0 (bvshl loadh16_1 #x0008)))
(assert (= load16_0 (bvor loadh162_0 loadh16_1)))
(assert (= rsp_3 load16_0))
(assert (= rsp_4 (bvadd rsp_3 #x0002)))
; 201c hlt
endeavor@debian:~$
```

Oh yeah, there was some Single Static Assignment that took place in there as well

Step Five :: Slicing Variables

We want to solve for single variables. Only a few instructions affect that single variable. We, “Slice,” those instructions out of the larger trace, greatly reducing in scope the problem we off-load to the SMT solver.

For example, maybe we want to see all conditional jumps that are affected by a single byte of input. *(Actually yes, yes this is exactly what we want to do).*

Step Five :: Slicing Forward (in0_0)

```
endeavor@debian:~/code/rnpnacho$ src/transform -n il.queso -l in0_0 -q in0_0.queso
endeavor@debian:~/code/rnpnacho$ luajit lua/printqueso.lua in0_0.queso
0x64 assign 16:r4_1 8:in0_0
0x68 assign 8:tmp_192 16:r4_1
0x68 store 8:mem_65585[16:r7_0] 8:tmp_192
0x7c and 16:r4_2 16:r4_1 16:0000000000000001
0x80 sub 16:flags_48 16:r4_2 16:0000000000000000
0x84 cmpeq 1:tmp_193 16:flags_48 16:0000000000000000
endeavor@debian:~/code/rnpnacho$
```

This is a flag which is influenced by in0_0.

Step Five :: Slicing Backward (tmp_193)

```
endeavor@debian:~/code/rnpnacho$ src/transform -n il.queso -b tmp_193 -q in0_0.queso
endeavor@debian:~/code/rnpnacho$ luajit lua/printqueso.lua in0_0.queso
0x64 assign 16:r4_1 8:in0_0
0x7c and 16:r4_2 16:r4_1 16:0000000000000001
0x80 sub 16:flags_48 16:r4_2 16:0000000000000000
0x84 cmpeq 1:tmp_193 16:flags_48 16:0000000000000000
endeavor@debian:~/code/rnpnacho$
```

We slice backward to make sure we have all the information we need to solve for the flag bit.

Oh look, there's our target input byte

Oh look, there's our target flag bit

Step Six :: Wrap with lua

Using command-line tools is tedious/slow. We'll write a lua library that will wrap our framework. We will **avoid overhead of serializing/deserializing**.

This is how we write the logic for our analysis from here on out.

```lua
require('lnacho')
require('nacho')

instructions = instructions_t.new()
instructions:from_queso_file(arg[1])

for k,v in pairs(instructions:instructions()) do print(nacho.fmtins(v)) end
```

Step Six :: Wrap with Lua

```
require(“lnacho”)
instructions = instructions_t.new()
instructions:from_queso_file(string filename)
instructions:slice_forward(string var_name, int var_count)
instructions:slice_backward(string var_name, int var_count)
instructions:smtlib2()
instructions:instructions()
instructions:concretize()
instructions:ssa_variable(string var_name)
```

Step Seven :: Logic

```
function traceSolveInput (input)
    executeTrace(input)
    if no-new-destinations then return false end
    newInputs = slice-and-solve-for-all-inputs()
    return new-inputs(newInputs)
end
```

Step Seven :: Logic

```
Boring
```

Step Eight :: Results

```
548   find new destinations
549   solving input 0,0,0,0,1,1,0,0,0,0
550   executing trace
551   loading queso
552   find new destinations
553   new destination address 0xd14
554   new destination address 0xce4
555   new destination address 0xcb4
556   new destination address 0xc84
557   slice and solve
558   gen new inputs
559   new input   0,0,0,0,1,1,1,0,0,0
560   new input   0,0,0,0,1,1,0,1,0,0
561   new input   0,0,0,0,1,1,0,0,1,0
562   new input   0,0,0,0,1,1,0,0,0,1
563   solving input 0,0,0,0,1,0,1,0,0,0
564   executing trace
565   loading queso
566   find new destinations
```

These are the bytes we feed as input

We trace execution, load the trace

We find new destinations, defined here as the address of the instruction after a conditional input-determined jump

Because we encountered new blocks, we will generate new inputs to further explore those blocks.

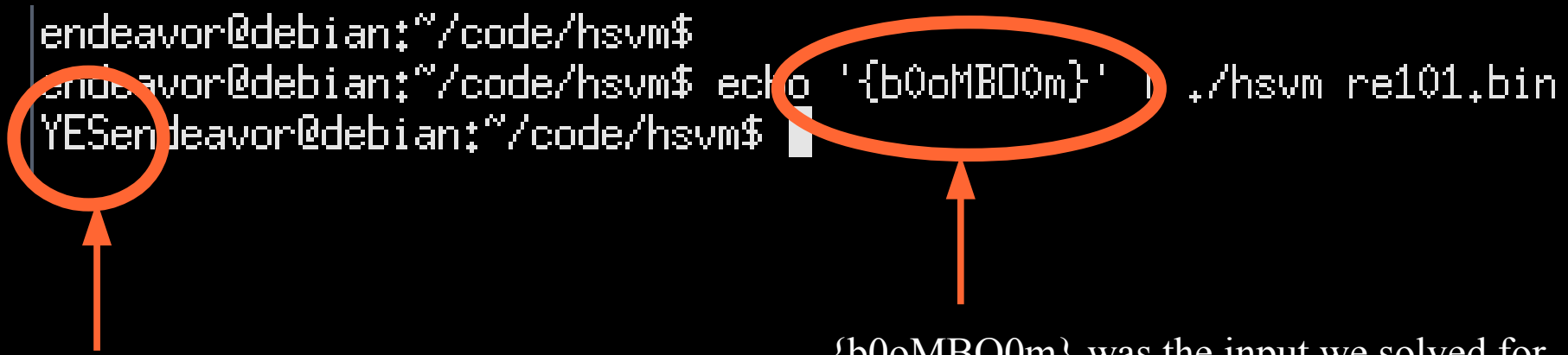
The process repeats, a lot

Step Eight :: Results

```
7810  solving input 123,98,48,111,77,66,1,48,109,1
7811  executing trace
7812  loading queso
7813  find new destinations
7814  solving input 123,98,48,111,77,66,0,48,109,1
7815  executing trace
7816  loading queso
7817  find new destinations
7818  solving input 123,98,48,111,77,66,79,1,109,1
7819  executing trace
7820  loading queso
7821  find new destinations
7822  solving input 123,98,48,111,77,66,79,0,109,1
7823  executing trace
7824  loading queso
7825  find new destinations
7826  solving input 123,98,48,111,77,66,79,48,109,0
7827  executing trace
7828  loading queso
7829  find new destinations
7830  solving input 123,98,48,111,77,66,79,48,109,125
7831  executing trace
7832  result found!
7833  123,98,48,111,77,66,79,48,109,125
```

```
123 = {
98  = b
48  = 0
111 = o
77  = M
66  = B
79  = O
48  = 0
109 = m
125 = }
```

Step Eight :: Results



{b0oMBO0m} was the input we solved for

The outputting of the string YES was the condition we were trying to achieve

Step Nine :: Drink

tfpwn.com ← A website. I have a blog there. Maybe I'll update it.

endeavor@rainbowsandpwnies.com ← If you send emails here I'll get them.

http://github.com/endeav0r/ ← I release most of my code here. There's a lot of it.

@rednovae ← Twitter. I have one of those.

http://github.com/rainbowsandpwnies/nacho

All of the code you saw here today