Vulnerability Modeling with Binary Ninja

Pacific Hackers 2018

Who I am

Josh Watson (@josh_watson)

- Senior Security Engineer for Trail of Bits
- Previously a cog in the Military Industrial Complex
- Before that, other stuff
- Prominent member of the Binary Ninja community

Agenda

1. Introduction
2. Case study: Heartbleed
3. Automated bug hunting without source
4. Results
5. Conclusion
6. Q&A

Introduction

Static analysis is hard.

Introduction

Static analysis is harder when you don't have source.

Hacking like it's 2014:
let's find Heartbleed!

TRAIL OF BITS

Heartbleed

tls1_process_heartbeat

```
hbtype = *p++;
n2s(p, payload);
pl = p;
/* Skip some stuff... */
if (hbtype == TLS1_HB_REQUEST)
    {
    unsigned char *buffer, *bp;
    int r;

    /* Allocate memory for the response, size is 1 bytes
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
```

Heartbleed

tls1_process_heartbeat

```
hbtype = *p++;
n2s(p, payload);
pl = p;
/* Skip some stuff... */
if (hbtype == TLS1_HB_REQUEST)
    {
    unsigned char *buffer, *bp;
    int r;

    /* Allocate memory for the response, size is 1 bytes
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
```

Heartbleed

tls1_process_heartbeat

```
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

Heartbleed

tls1_process_heartbeat

```
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

Heartbleed

tls1_process_heartbeat

```
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

How do we automate analysis of this?

"Let's be clear: it is trivial to create a static analyzer that runs fast and flags heartbleed. I can accomplish this, for example, by flagging a taint error in every line of code that is analyzed. The task that is truly difficult is to create a static analysis tool that is performant and that has a high signal to noise ratio for a broad range of analyzed programs."

—John Regehr

A New Development for Coverity and Heartbleed

Previous work modeling Heartbleed

On detecting Heartbleed with static analysis

Byte-swapping is probably untrusted data that should be tainted.

Look for:

1. byte-swapping
2. combining smaller integers into larger ones
3. byte-swapped values being used as array indices or size parameters for `memcpy`

```
    1. byte_swapping: Performing a byte swapping operation on p implies that it came from an external source, and is therefore tainted.
    2. var_assign_var: Assigning: payload = ((unsigned int)p[0] << 8) | (unsigned int)p[1] . Both are now tainted.
        n2s(p, payload);
        pl = p;

    3. Condition s->msg_callback , taking true branch
        if (s->msg_callback)
                s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                        &s->s3->rrec.data[0], s->s3->rrec.length,
                        s, s->msg_callback_arg);

    4. Condition hbtype == 1 , taking true branch
        if (hbtype == TLS1_HB_REQUEST)
                {
                unsigned char *buffer, *bp;
                int r;

                /* Allocate memory for the response, size is 1 bytes
                 * message type, plus 2 bytes payload length, plus
                 * payload, plus padding
                 */
                buffer = OPENSSL_malloc(1 + 2 + payload + padding);
                bp = buffer;

                /* Enter response type, length and copy payload */
                *bp++ = TLS1_HB_RESPONSE;
                s2n(payload, bp);
    CID 1201699 (#1 of 1): Untrusted value as argument (TAINTED_SCALAR)
    5. tainted_data: Passing tainted variable payload to a tainted sink.
                memcpy(bp, pl, payload);
```

Previous work modeling Heartbleed

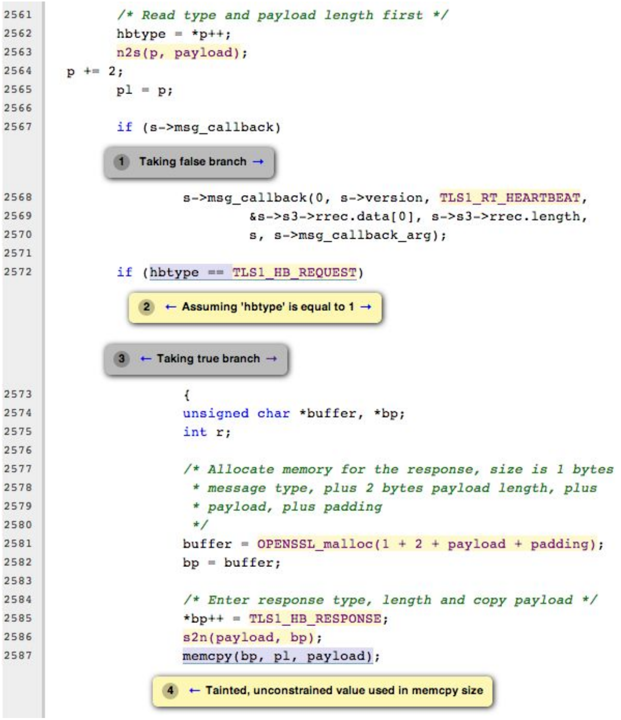
Using Static Analysis and Clang To Find Heartbleed

n2s calls are probably untrusted data that should be tainted.

Look for:

1. Results of n2s calls
2. No constraints on the results
3. The results being used as a size parameter in `memcpy` calls

Modifications to source are needed to facilitate this.



What do those tools have in common?

Automated bug hunting without source

Building our toolkit

Constraint Solving

$$x + y = 8$$
$$2x + 3 = 7$$

Constraint Solving

$$x + y = 8$$

$$2x + 3 = 7$$

$$x = ?$$

$$y = ?$$

Constraint Solving

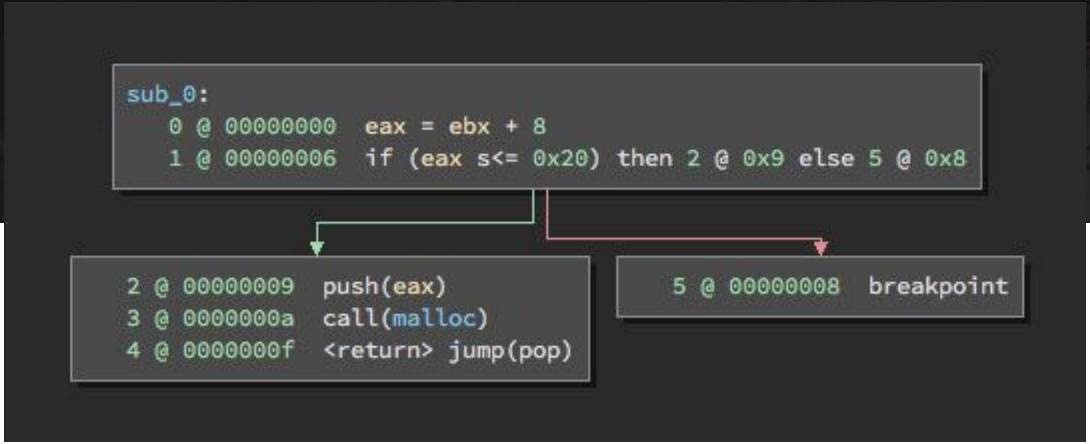
$$x + y = 8$$
$$2x + 3 = 7$$
$$x = 2$$
$$y = 6$$

Constraint Solving with Z3

```
>>> from z3 import *
>>> x = Int('x')
>>> y = Int('y')
>>> s = Solver()
>>> s.add(x + y == 8)
>>> s.add(2*x + 3 == 7)
>>> s.check()
sat
>>> s.model()
[x = 2, y = 6]
```

Constraint Solving with Z3

```
  lea eax, [ebx+8]
  cmp eax, 0x20
  jle allocate
  int3
allocate:
  push eax
  call malloc
  ret
```


sub_0:
0 @ 00000000 eax = ebx + 8
1 @ 00000006 if (eax s<= 0x20) then 2 @ 0x9 else 5 @ 0x8
2 @ 00000009 push(eax)
3 @ 0000000a call(malloc)
4 @ 0000000f <return> jump(pop)
5 @ 00000008 breakpoint

Constraint Solving with Z3

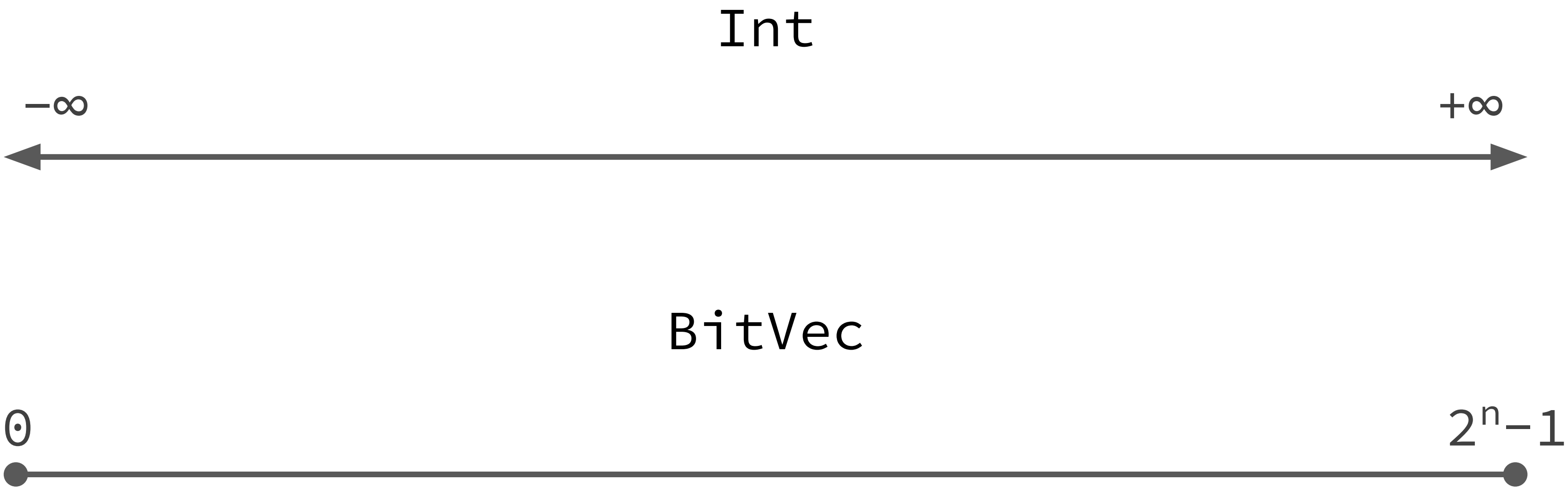
```
eax = ebx + 8
ebx > 0x20
eax <= 0x20
```

Constraint Solving with Z3

```
>>> eax = Int('eax')
>>> ebx = Int('ebx')
>>> s = Solver()
>>> s.add(eax == ebx + 8)
>>> s.add(ebx > 0x20)
>>> s.add(eax <= 0x20)
>>> s.check()
unsat
```

Constraint Solving with Z3

What went wrong?



Constraint Solving with Z3

```
>>> eax = Int('eax')
>>> ebx = Int('ebx')
>>> s = Solver()
>>> s.add(eax == ebx + 8)
>>> s.add(ebx > 0x20)
>>> s.add(eax <= 0x20)
>>> s.check()
unsat
```

Constraint Solving with Z3

```
>>> eax = BitVec('eax', 32)
>>> ebx = BitVec('ebx', 32)
>>> s = Solver()
>>> s.add(eax == ebx + 8)
>>> s.add(ebx > 0x20)
>>> s.add(eax <= 0x20)
>>> s.check()
sat
```

Constraint Solving with Z3

```
>>> s.model()
[ebx = 2147483640, eax = 2147483648]
>>> hex(2147483640)
'0x7ffffff8'
>>> hex(2147483648)
'0x80000000'
```

Constraint Solving with Z3

How do we collect semantics and translate those into a set of constraints?

- `BitVec` trivially works for registers
- Memory accesses need Z3's `Array` sort
- Stack variables reside in memory, tracking pushes and pops is complicated
- Can we treat stack variables the same as registers?

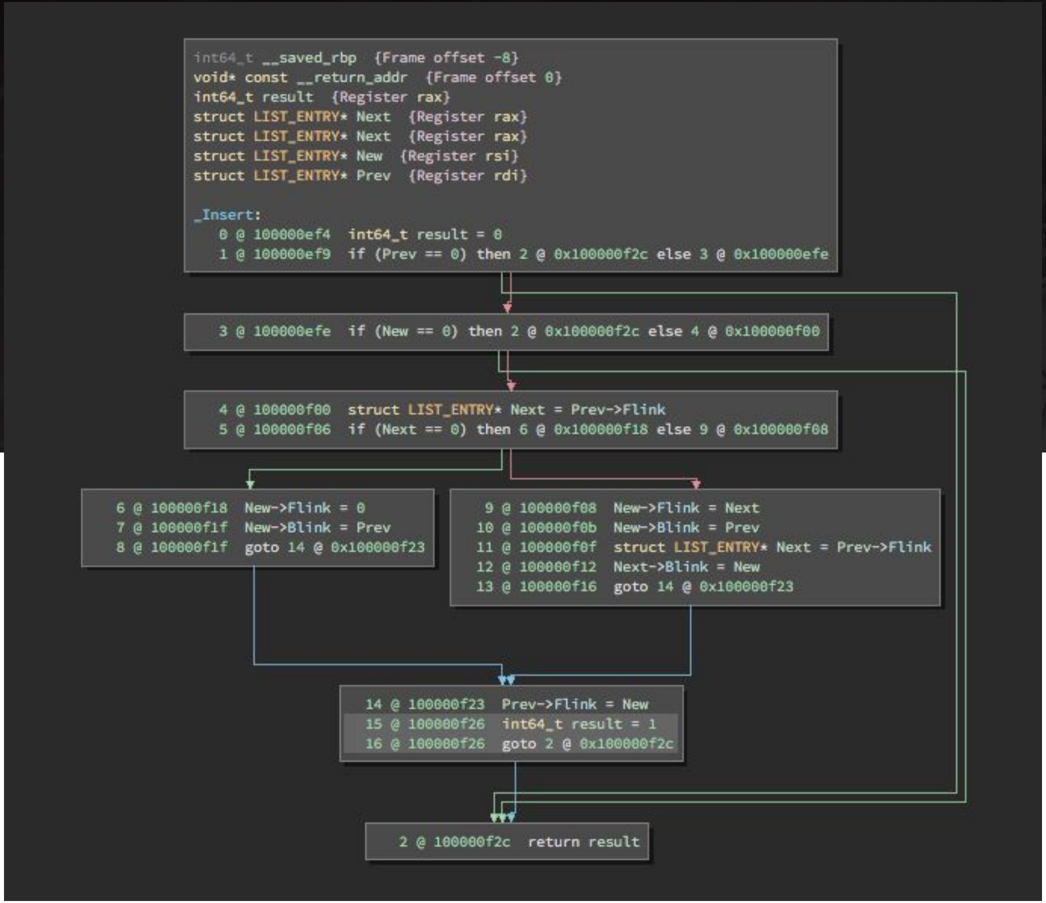
```
sub_0:
00000000  push    ebp
00000001  mov     ebp, esp
00000003  sub     esp, 0x8 {var_c}
00000006  mov     dword [ebp-0x4 {var_8}], 0x0
0000000d  mov     dword [ebp-0x8 {var_c}], 0x10
00000014  mov     eax, dword [ebp+0x8 {arg1}]
00000017  add     eax, dword [ebp-0x8 {var_c}]
0000001a  mov     esp, ebp
0000001c  pop     ebp
0000001d  retn
```

LLIL

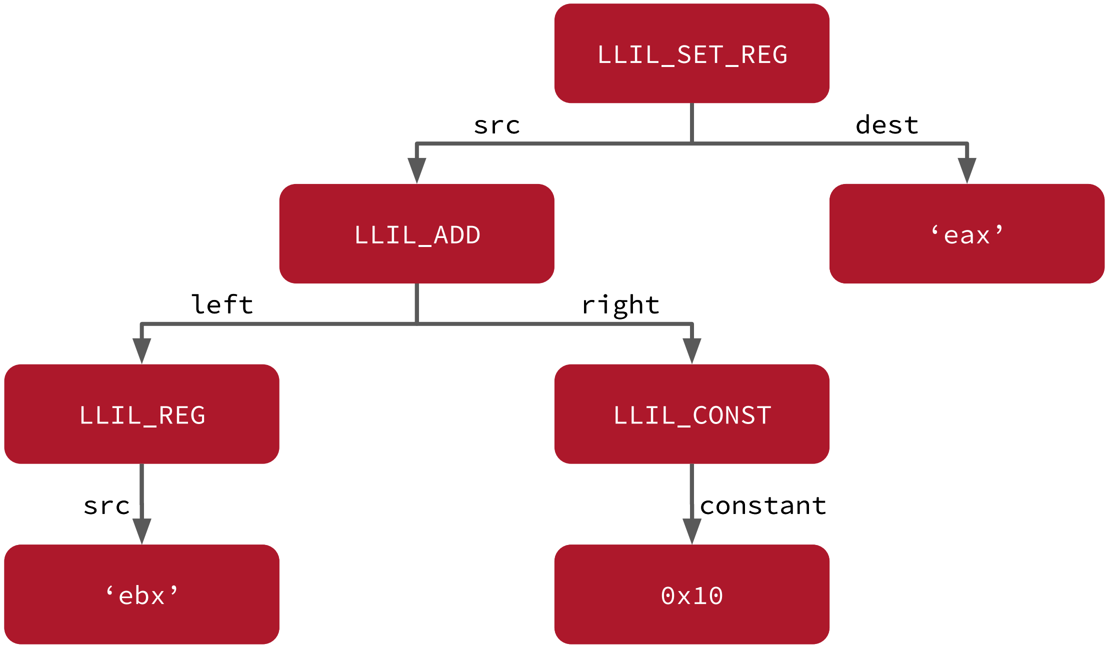
```
sub_0:
   0 @ 00000000  push(ebp)
   1 @ 00000001  ebp = esp {__saved_ebp}
   2 @ 00000003  esp = esp - 8
   3 @ 00000006  [ebp - 4 {var_8}].d = 0
   4 @ 0000000d  [ebp - 8 {var_c}].d = 0x10
   5 @ 00000014  eax = [ebp + 8 {arg1}].d
   6 @ 00000017  eax = eax + [ebp - 8 {var_c}].d
   7 @ 0000001a  esp = ebp
   8 @ 0000001c  ebp = pop
   9 @ 0000001d  <return> jump(pop)
```

MLIL

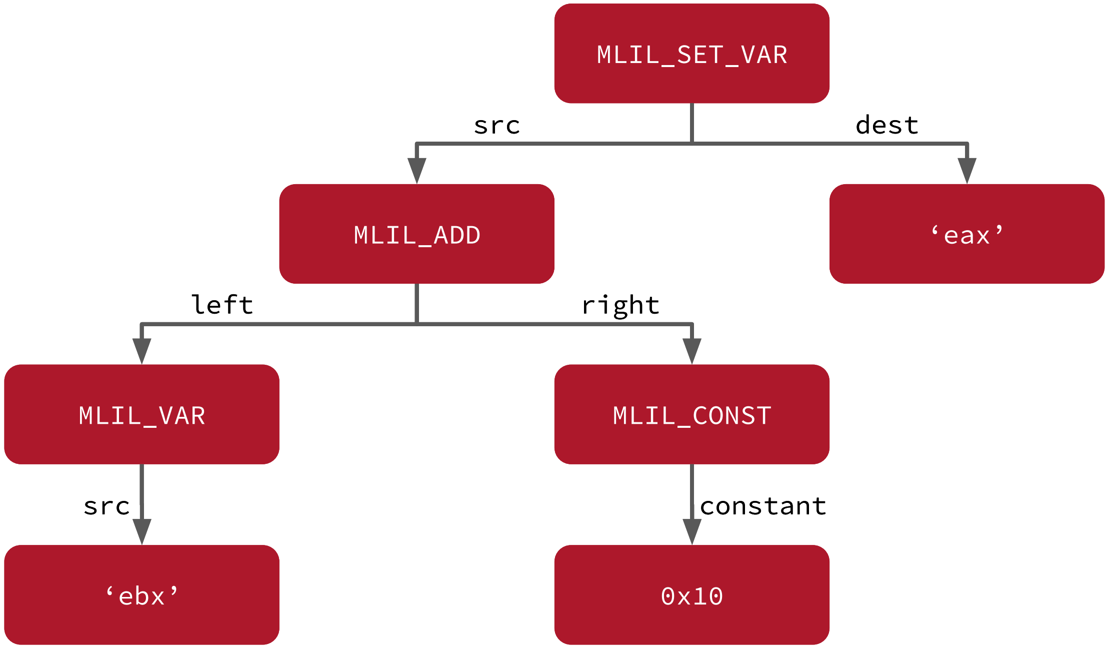
```
sub_0:
   0 @ 00000014  int32_t eax = arg1
   1 @ 00000017  int32_t eax_1 = eax + 0x10
   2 @ 0000001d  return eax_1
```


int64_t __saved_rbp {Frame offset -8}
void* const __return_addr {Frame offset 0}
int64_t result {Register rax}
struct LIST_ENTRY* Next {Register rax}
struct LIST_ENTRY* Next {Register rax}
struct LIST_ENTRY* New {Register rsi}
struct LIST_ENTRY* Prev {Register rdi}
_Insert:
0 @ 100000ef4 int64_t result = 0
1 @ 100000ef9 if (Prev == 0) then 2 @ 0x100000f2c else 3 @ 0x100000efe
3 @ 100000efe if (New == 0) then 2 @ 0x100000f2c else 4 @ 0x100000f00
4 @ 100000f00 struct LIST_ENTRY* Next = Prev->Flink
5 @ 100000f06 if (Next == 0) then 6 @ 0x100000f18 else 9 @ 0x100000f08
6 @ 100000f18 New->Flink = 0
7 @ 100000f1f New->Blink = Prev
8 @ 100000f1f goto 14 @ 0x100000f23
9 @ 100000f08 New->Flink = Next
10 @ 100000f0b New->Blink = Prev
11 @ 100000f0f struct LIST_ENTRY* Next = Prev->Flink
12 @ 100000f12 Next->Blink = New
13 @ 100000f16 goto 14 @ 0x100000f23
14 @ 100000f23 Prev->Flink = New
15 @ 100000f26 int64_t result = 1
16 @ 100000f26 goto 2 @ 0x100000f2c
2 @ 100000f2c return result

```
lea eax, [ebx + 0x10]
```

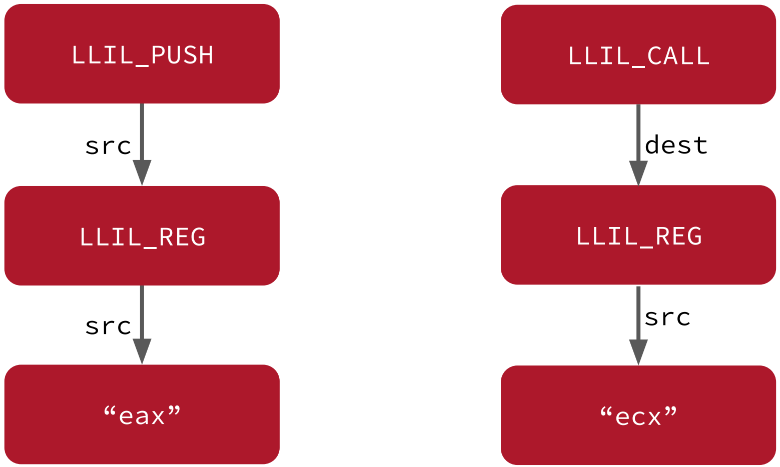



```
lea eax, [ebx + 0x10]
```

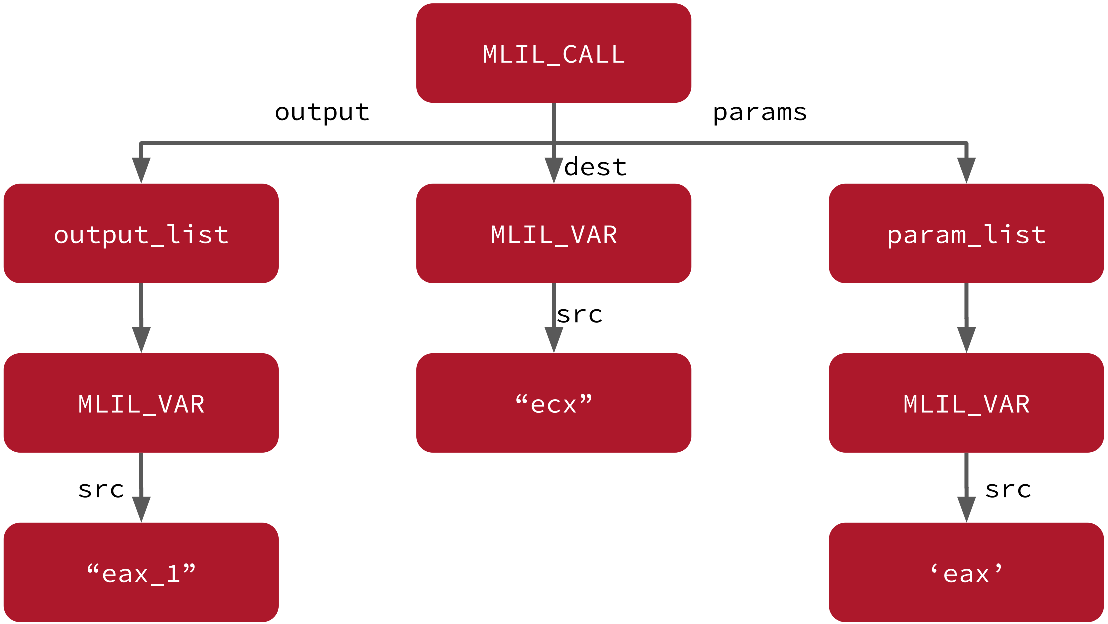



```
lea eax, [ebx + 0x10]
```

```
push eax; call ecx
```



push eax; call ecx



push eax; call ecx

But…

What's wrong with modeling this?

```
mov eax, ebx
lea eax, [ecx+eax*4]
```

What's wrong with modeling this?

```
eax = ebx
eax = ecx + eax << 2
```

What's wrong with modeling this?

Constraints are purely expressing mathematical truths about variables in a system of equations and have **no** temporal element at all.

Single Static Assignment

Single Static Assignment (SSA)

SSA form is a representation of a program in which every variable is defined once and only once.
If the variable is assigned a new value, a new "version" of that variable is defined instead.

Single Static Assignment

Original form

```
a = 1
b = 2
a = a + b
```

SSA form

$$a_1 = 1$$
$$b_1 = 2$$
$$a_2 = a_1 + b_1$$

Single Static Assignment

Original form

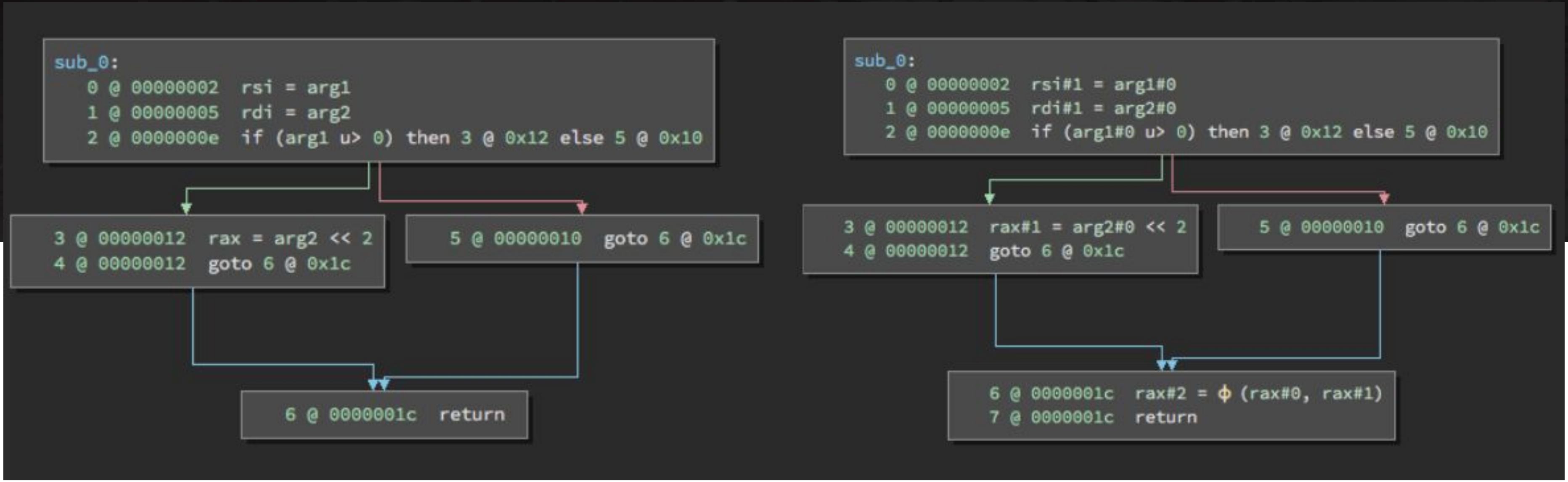
```
def f(a):
  if a > 20:
    a = a * 2
  else:
    a = a + 5
  return a
```

SSA form

$$
\begin{array}{l}
\texttt{def f}(a_0)\texttt{:} \\
\quad \texttt{if } a_0 > 20\texttt{:} \\
\quad\quad a_1 = a_0 * 2 \\
\quad \texttt{else:} \\
\quad\quad a_2 = a_0 + 5 \\
\quad \boxed{a_3 = \Phi(a_1, a_2)} \\
\quad \texttt{return } a_3
\end{array}
$$

Single Static Assignment

SSA makes it easy to explicitly track all definitions and uses of a variable throughout the lifetime of the program.


sub_0:
0 @ 00000002 rsi = arg1
1 @ 00000005 rdi = arg2
2 @ 0000000e if (arg1 u> 0) then 3 @ 0x12 else 5 @ 0x10
3 @ 00000012 rax = arg2 << 2
4 @ 00000012 goto 6 @ 0x1c
5 @ 00000010 goto 6 @ 0x1c
6 @ 0000001c return
sub_0:
0 @ 00000002 rsi#1 = arg1#0
1 @ 00000005 rdi#1 = arg2#0
2 @ 0000000e if (arg1#0 u> 0) then 3 @ 0x12 else 5 @ 0x10
3 @ 00000012 rax#1 = arg2#0 << 2
4 @ 00000012 goto 6 @ 0x1c
5 @ 00000010 goto 6 @ 0x1c
6 @ 0000001c rax#2 = ϕ (rax#0, rax#1)
7 @ 0000001c return

MLIL SSA form

Medium Level IL	Medium Level IL SSA
`MLIL_SET_VAR`	`MLIL_SET_VAR_SSA`
`MLIL_VAR`	`MLIL_VAR_SSA`
`MLIL_CALL`	`MLIL_CALL_SSA`
`MLIL_LOAD`	`MLIL_LOAD_SSA`
`Variable`	`SSAVariable`

Putting it all together

Combining MLIL SSA with Z3

```
eax = ebx
eax = ecx + (eax << 2)
```

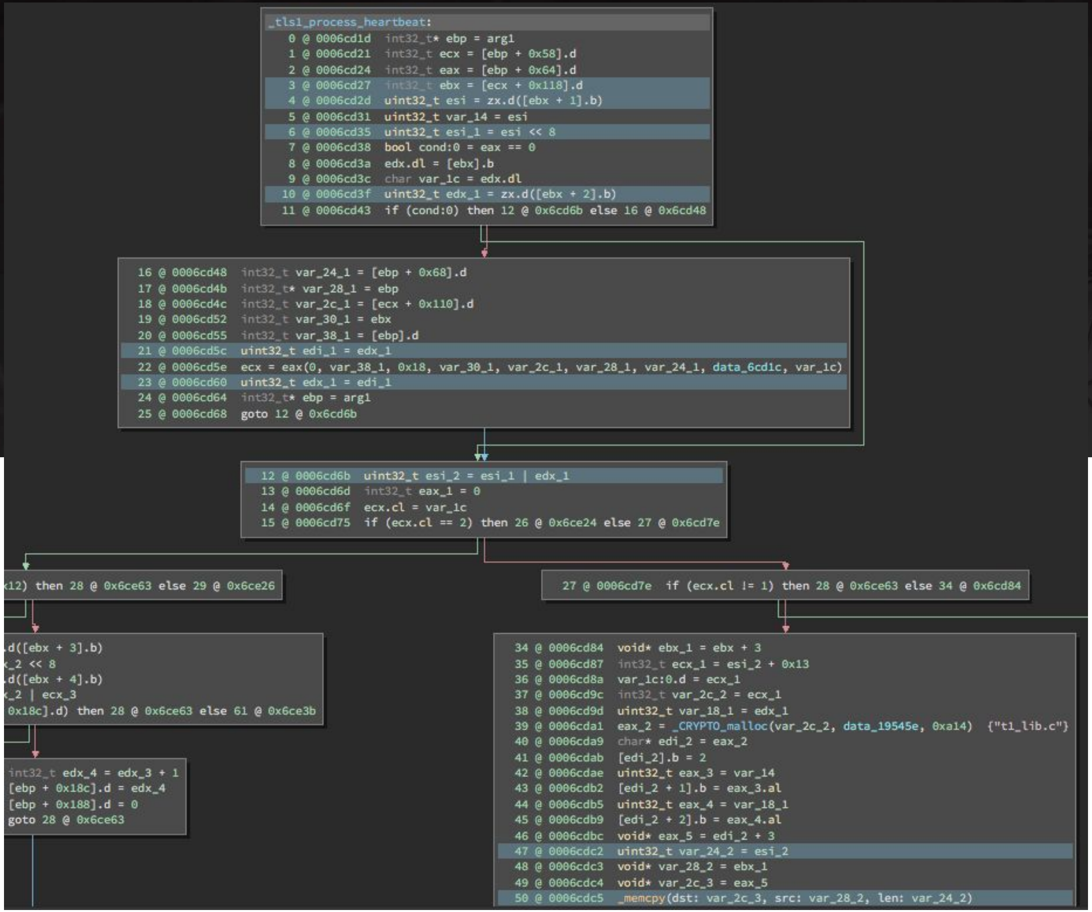
Combining MLIL SSA with Z3

```
eax#1 = ebx#0
eax#2 = ecx#0 + (eax#1 << 2)
```

Combining MLIL SSA with Z3

```
eax_1 = BitVec(‘eax#1’, 32)
ebx_0 = BitVec(‘ebx#0’, 32)
ecx_0 = BitVec(‘ecx#0’, 32)
eax_2 = BitVec(‘eax#2’, 32)
s = Solver()
s.add(
  eax_1 == ebx_0,
  eax_2 == ecx_0 + (eax_1 << 2)
)
```

Writing the Script


_tls1_process_heartbeat:
0 @ 0006cd1d int32_t* ebp = arg1
1 @ 0006cd21 int32_t ecx = [ebp + 0x58].d
2 @ 0006cd24 int32_t eax = [ebp + 0x64].d
3 @ 0006cd27 int32_t ebx = [ecx + 0x118].d
4 @ 0006cd2d uint32_t esi = zx.d([ebx + 1].b)
5 @ 0006cd31 uint32_t var_14 = esi
6 @ 0006cd35 uint32_t esi_1 = esi << 8
7 @ 0006cd38 bool cond:0 = eax == 0
8 @ 0006cd3a edx.dl = [ebx].b
9 @ 0006cd3c char var_1c = edx.dl
10 @ 0006cd3f uint32_t edx_1 = zx.d([ebx + 2].b)
11 @ 0006cd43 if (cond:0) then 12 @ 0x6cd6b else 16 @ 0x6cd48
16 @ 0006cd48 int32_t var_24_1 = [ebp + 0x68].d
17 @ 0006cd4b int32_t* var_28_1 = ebp
18 @ 0006cd4c int32_t var_2c_1 = [ecx + 0x110].d
19 @ 0006cd52 int32_t var_30_1 = ebx
20 @ 0006cd55 int32_t var_38_1 = [ebp].d
21 @ 0006cd5c uint32_t edi_1 = edx_1
22 @ 0006cd5e ecx = eax(0, var_38_1, 0x18, var_30_1, var_2c_1, var_28_1, var_24_1, data_6cd1c, var_1c)
23 @ 0006cd60 uint32_t edx_1 = edi_1
24 @ 0006cd64 int32_t* ebp = arg1
25 @ 0006cd68 goto 12 @ 0x6cd6b
12 @ 0006cd6b uint32_t esi_2 = esi_1 | edx_1
13 @ 0006cd6d int32_t eax_1 = 0
14 @ 0006cd6f ecx.cl = var_1c
15 @ 0006cd75 if (ecx.cl == 2) then 26 @ 0x6ce24 else 27 @ 0x6cd7e
x12) then 28 @ 0x6ce63 else 29 @ 0x6ce26
27 @ 0006cd7e if (ecx.cl != 1) then 28 @ 0x6ce63 else 34 @ 0x6cd84
.d([ebx + 3].b)
_2 << 8
.d([ebx + 4].b)
_2 | ecx_3
0x18c].d) then 28 @ 0x6ce63 else 61 @ 0x6ce3b
int32_t edx_4 = edx_3 + 1
[ebp + 0x18c].d = edx_4
[ebp + 0x188].d = 0
goto 28 @ 0x6ce63
34 @ 0006cd84 void* ebx_1 = ebx + 3
35 @ 0006cd87 int32_t ecx_1 = esi_2 + 0x13
36 @ 0006cd8a var_1c:0.d = ecx_1
37 @ 0006cd9c int32_t var_2c_2 = ecx_1
38 @ 0006cd9d uint32_t var_18_1 = edx_1
39 @ 0006cda1 eax_2 = _CRYPTO_malloc(var_2c_2, data_19545e, 0xa14) {"t1_lib.c"}
40 @ 0006cda9 char* edi_2 = eax_2
41 @ 0006cdab [edi_2].b = 2
42 @ 0006cdae uint32_t eax_3 = var_14
43 @ 0006cdb2 [edi_2 + 1].b = eax_3.al
44 @ 0006cdb5 uint32_t eax_4 = var_18_1
45 @ 0006cdb9 [edi_2 + 2].b = eax_4.al
46 @ 0006cdbc void* eax_5 = edi_2 + 3
47 @ 0006cdc2 uint32_t var_24_2 = esi_2
48 @ 0006cdc3 void* var_28_2 = ebx_1
49 @ 0006cdc4 void* var_2c_3 = eax_5
50 @ 0006cdc5 _memcpy(dst: var_2c_3, src: var_28_2, len: var_24_2)

Writing the script

1. Find our “sinks”
2. Eliminate sinks that are obviously not vulnerable
3. Trace the variables the size depends on (backwards slice)
4. Identify variables that might be part of a byte swap
5. Identify additional constraints on the size parameter
6. Solve the model
7. Find bugs

Writing the script

Step 1: Finding our "sinks"

```
memcpy_refs = [
    (ref.function, ref.address)
    for ref in bv.get_code_refs(bv.symbols['_memcpy'].address)
]
dangerous_calls = []
for function, addr in memcpy_refs:
    call_instr = function.get_low_level_il_at(addr).medium_level_il
    if check_memcpy(call_instr.ssa_form):
        dangerous_calls.append((addr, call_instr.address))
```

Writing the script

Step 2: Eliminate sinks that we know aren't vulnerable

```
def check_memcpy(memcpy_call):
    size_param = memcpy_call.params[2]
    if size_param.operation != MediumLevelILOperation.MLIL_VAR_SSA:
        return False
    possible_sizes = size_param.possible_values
    if possible_sizes.type != RegisterValueType.UndeterminedValue:
        return False
    model = ByteSwapModeler(size_param, bv.address_size)
    return model.is_byte_swap()
```

Writing the script

Step 3: Trace the variables the size depends on

```
var_def = self.function.get_ssa_var_definition(self.var.src)

# Visit statements that our variable directly depends on
self.to_visit.append(var_def)

while self.to_visit:
    idx = self.to_visit.pop()
    if idx is not None:
        self.visit(self.function[idx])
```

Writing the script

Step 3: Trace the variables the size depends on

```
def visit_MLIL_ADD(self, expr):
    left = self.visit(expr.left)
    right = self.visit(expr.right)

    if None not in (left, right):
        return left + right
```

Writing the script

Step 4: Identify variables that might be part of a byte swap

```
def visit_MLIL_VAR_SSA(self, expr):

    if expr.src not in self.visited:
        var_def = expr.function.get_ssa_var_definition(expr.src)

        if var_def is not None:
            self.to_visit.append(var_def)

    src = create_BitVec(expr.src, expr.size)

    value_range = identify_byte(expr, self.function)
    if value_range is not None:
        self.solver.add(Or(src == 0, And(src = value_range.step)))
        self.byte_vars.add(expr.src)

    return src
```

Writing the script

Step 4: Identify variables that might be part of a byte swap

```
phi_values = []

for var in expr.src:
    if var not in self.visited:
        var_def = self.function.get_ssa_var_definition(var)
        self.to_visit.append(var_def)

    src = create_BitVec(var, var.var.type.width)

    # ...

    phi_values.append(src)

if phi_values:
    phi_expr = reduce(
        lambda i, j: Or(i, j), [dest == s for s in phi_values]
    )

    self.solver.add(phi_expr)
```

Writing the script

Step 4: Identify variables that might be part of a byte swap

```
# If this value can never be larger than a byte,
# then it must be one of the bytes in our swap.
# Add it to a list to check later.
if src is not None and not isinstance(src, (int, long)):
    value_range = identify_byte(expr.src, self.function)
    if value_range is not None:
        self.solver.add(Or(src == 0, And(src <= value_range.end, src >= value_range.step)))

        self.byte_vars.add(*expr.src.vars_read)

        if self.byte_values.get((value_range.step, value_range.end)) is None:
            self.byte_values[
                (value_range.step, value_range.end)
            ] = simplify(Extract(
                        int(math.floor(math.log(value_range.end, 2))),
                        int(math.floor(math.log(value_range.step, 2))),
                        src
                    )
            )
```

Writing the script

Step 5: Identify constraints on the size parameter

```
for i, branch in self.var.branch_dependence.iteritems():
    for vr in self.function[i].vars_read:
        if vr in self.byte_vars:
            raise ModelIsConstrained()

        vr_def = self.function.get_ssa_var_definition(vr)
        if vr_def is None:
            continue

        for vr_vr in self.function[vr_def].vars_read:
            if vr_vr in self.byte_vars:
                raise ModelIsConstrained()
```

Writing the script

Step 6: Solve the model

```
self.solver.add(
    Not(
        And(
            var == ZeroExt(
                var.size() - len(ordering)*8,
                Concat(*ordering)
            ),
            reverse_var == ZeroExt(
                reverse_var.size() - reversed_ordering.size(),
                reversed_ordering
            )
        )
    )
)

if self.solver.check() == unsat:
    return True
```

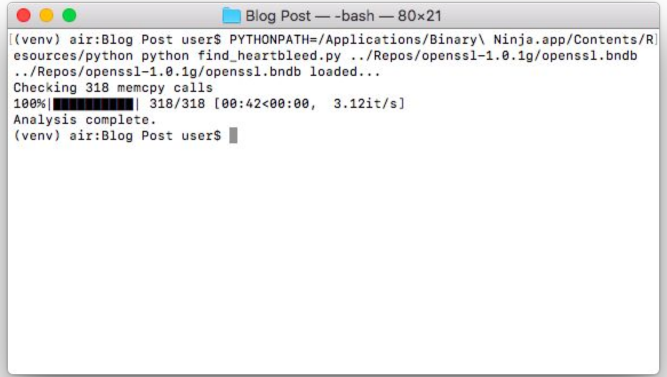
Results

Results

OpenSSL 1.0.1f

Compiled with `./Configure darwin-i386-cc`

Two functions are found: `tls1_process_heartbeat` and `dtls1_process_heartbeat`.



OpenSSL 1.0.1g

Compiled with `./Configure darwin-i386-cc`

The vulnerable functions are no longer present.

Conclusion

https://github.com/trailofbits/binjascripts/tree/master/find_heartbleed

Takeaway

Binary Ninja's MLIL and SSA form make advanced binary analysis easier to implement.

Questions?

Josh Watson

Senior Security Engineer

josh@trailofbits.com

@josh_watson

References/Links

On detecting Heartbleed with static analysis

https://www.synopsys.com/blogs/software-security/detecting-heartbleed-with-static-analysis/

Using Static Analysis and Clang To Find Heartbleed

https://blog.trailofbits.com/2014/04/27/using-static-analysis-and-clang-to-find-heartbleed

Heartbleed and Static Analysis

https://blog.regehr.org/archives/1125

A New Development for Coverity and Heartbleed

https://blog.regehr.org/archives/1128

Vulnerability Modeling with Binary Ninja

https://blog.trailofbits.com/2018/04/04/vulnerability-modeling-with-binary-ninja/

TRAIL OF BITS