Applying Taint Analysis and Theorem Proving to Exploit Development

Sean Heelan, Immunity Inc.

RECON 2010

Me

- Security Researcher with Immunity Inc
- Background in verification/program analysis
- Hobbies include watching the sec industry reinvent 30 year old academic research… badly :P

sean@immunityinc.com http://twitter.com/seanhn

Topics to be Covered

- Static and dynamic analysis tradeoffs
- Dataflow and taint analysis
- Intermediate Representations of ASM
- Building logical formulae from execution traces
- Solving the above formulae for useful results
- Applying all of the above to RE and Exploit development

Introduction & Motivation

Exploit development

- Exploit dev seems to involves two primary talents (+practice/knowledge)
  - Creativity/Being a devious bastard
  - Tenacity/Painstaking reverse engineering and debugging
- Success at the former?
  - Innate ability?
- Success at the latter?
  - Motivation? Tool support?

Vulnerability -> Exploit

- Our workflow primarily depends on how we have found the bug
- Fuzzing
- Source code/Binary auditing
- Reversing a patch
- ‘Reversing’ a public bug announcement

Where is Your Time Actually Spent?

Fuzzing – The Rollercoaster of Fail

Yay, I found a bug!

Fuzzing – The Rollercoaster of Fail

Um, hang on… wtf just happened?

Fuzzing – The Rollercoaster of Fail

- Why did the crash occur?
- Where did the data involved come from?
- Is the data attacker influencable?
- What conditions are imposed on it?
- Exactly what computations have been performed on the data?
- Where is the rest of the attacker controllable data?
- Rinse/Repeat for all interesting data

Are other bug finding methods any better?

- How do I reach the vulnerable function/path?
- What conditions does input have to meet?
- What the hell does ObfuscatedFunctionXYZ even do to my data?
  - Unintentional and intentional arithmetic obfuscation is common and oftentimes automatically reversible
  - Even basic data copying can make your day miserable if done frequently

A General RE Problem

- Can variable X have value Y after a given instruction sequence?
  - What input value(s) cause this to occur

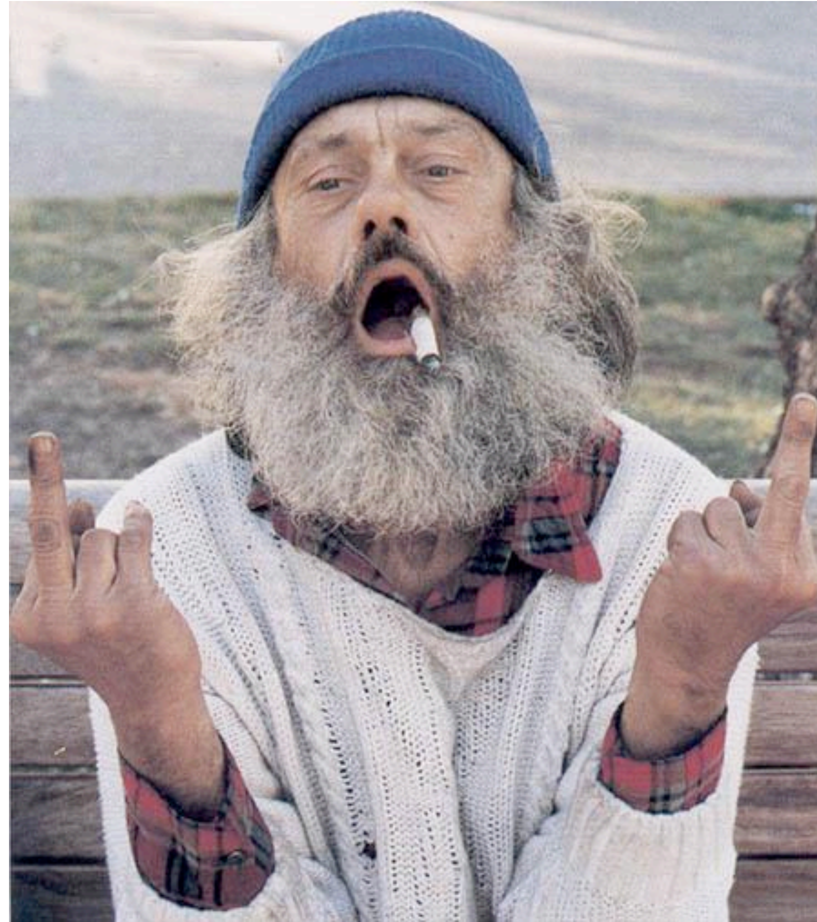
```
708FC8C8   xp_sp3_asycfilt.dll::sub_708FC8C8
708FC959   mov     eax, ebx
708FC95B   neg     eax
708FC95D   sbb     eax, eax
708FC95F   and     eax, ecx
708FC961   mov     ds:[edx], eax
708FC963   jmp     byte cs:loc_708FC967
```

```
708FC8C8   xp_sp3_asycfilt.dll::sub_708FC8C8
708FC967   cmp     ds:[edx], 0
708FC96A   jz      byte cs:loc_708FC976
```

```
708F92D7    asycfilt.dll::sub_708F92D7
708F95E1    mov      ebx, ds:[edi+0x18]
708F95E4    mov      edx, ds:[edi+8]
708F95E7    add      edx, ebx
708F95E9    movsx    edx, word dx
708F95EC    imul     edx, edx, 0x1151
708F95F2    mov      ss:[ebp+arg_0], edx
708F95F5    movsx    ebx, word bx
708F95F8    imul     ebx, ebx, 0x3B21
708F95FE    sub      edx, ebx
708F9600    mov      ebx, ds:[edi+0x10]
708F9603    mov      ss:[ebp+var_4], edx
708F9606    movsx    edx, word ds:[edi+8]
708F960A    mov      edi, ds:[edi]
708F960C    imul     edx, edx, 0x187E
708F9612    add      edx, ss:[ebp+arg_0]
708F9615    mov      ss:[ebp+arg_0], ebx
708F9618    add      ebx, edi
708F961A    sub      edi, ss:[ebp+arg_0]
708F961D    shl      ebx, byte 13
708F9620    shl      edi, byte 13
708F9623    mov      ss:[ebp+var_C], edi
708F9626    lea      edi, ds:[ebx+edx]
708F9629    mov      ss:[ebp+var_1C], edi
708F962C    mov      edi, ss:[ebp+var_4]
708F962F    mov      ss:[ebp+arg_4], ebx
708F9632    sub      ebx, edx
```

```
708F92D7 asycfilt.dll::sub_708F92D7
708F938E movsx esi, word ss:[ebp+arg_0]
708F9392 imul esi, ds:[ecx+0xC0]
708F9399 movsx edx, word ss:[ebp+var_20]
708F939D imul edx, ds:[ecx+0x40]
708F93A1 lea edi, ds:[edx+esi]
708F93A4 movsx edi, word di
708F93A7 imul edi, edi, 0x1151
708F93AD movsx esi, word si
708F93B0 imul esi, esi, 0x3B21
708F93B6 mov ebx, edi
708F93B8 sub ebx, esi
708F93BA movsx esi, word dx
708F93BD mov edx, ss:[ebp+var_8]
708F93C0 imul esi, esi, 0x187E
708F93C6 movsx edx, word ds:[edx]
708F93C9 imul edx, ds:[ecx]
708F93CC add esi, edi
708F93CE movsx edi, word ss:[ebp+arg_8]
708F93D2 imul edi, ds:[ecx+0x80]
708F93D9 mov ss:[ebp+var_4], edi
708F93DC add edi, edx
708F93DE sub edx, ss:[ebp+var_4]
708F93E1 shl edi, byte 13
708F93E4 shl edx, byte 13
708F93E7 mov ss:[ebp+var_C], edx
708F93EA lea edx, ds:[edi+esi]
708F93ED sub edi, esi
708F93EF mov ss:[ebp+var_1C], edx
708F93F2 mov edx, ss:[ebp+var_C]
708F93F5 lea esi, ds:[edx+ebx]
708F93F8 sub edx, ebx
708F93FA mov ss:[ebp+var_28], edx
708F93FD movsx edx, word ss:[ebp+var_14]
708F9401 imul edx, ds:[ecx+0xA0]
708F9408 mov ss:[ebp+var_C], edx
708F940B movsx edx, word ss:[ebp+var_2C]
708F940F imul edx, ds:[ecx+0x20]
708F9413 mov ss:[ebp+var_20], edi
708F9416 movsx edi, word ss:[ebp+var_10]
708F941A imul edi, ds:[ecx+0x60]
708F941E mov ss:[ebp+var_24], esi
708F9421 movsx esi, word ss:[ebp+arg_4]
708F9425 imul esi, ds:[ecx+0xE0]
708F942C lea ebx, ds:[esi+edx]
708F942F mov ss:[ebp+arg_0], ebx
708F9432 mov ebx, ss:[ebp+var_C]
708F9435 add ebx, edi
708F9437 mov ss:[ebp+var_4], edi
708F943A mov ss:[ebp+var_2C], edx
708F943D mov ss:[ebp+arg_8], ebx
708F9440 lea ebx, ds:[esi+edi]
708F9443 mov edi, ss:[ebp+var_C]
708F9446 add edx, edi
708F9448 movsx edi, word ss:[ebp+arg_0]
708F944C imul edi, edi, 0xFFFFE333
708F9452 mov ss:[ebp+arg_0], edi
708F9455 movsx edi, word ss:[ebp+arg_8]
708F9459 imul edi, edi, 0xFFFFADFD
708F945F mov ss:[ebp+arg_4], edx
708F9462 add edx, ebx
708F9464 movsx edx, word dx
708F9467 mov ss:[ebp+arg_8], edi
708F946A imul edx, edx, 0x25A1
```

Nuts to that!



Current tool support

- Disassemblers
- Debuggers
- Manual static analysis platforms
- Scriptable debuggers and static analysis tools
- Instrumentation frameworks

Current tool support

- We have many tools that provide various levels of abstraction over a program
- Deriving meaning from these abstractions is still primarily up to the user
- More abstractions == Less pain
- More automation == Less pain
- Less pain == ???

Problem statement

- Given an arbitrary point in a program and a collection of memory locations/registers:
  - Are those locations *tainted* by user input?
  - What exact bytes of user input?
  - What computations were done on these bytes?
  - What conditions have been imposed on these bytes?
  - Bonus Round: Given memory location *m* with value *y* automatically generate an input that results in value *x* at location *m*

How does that help?

- What percentage of your exploit development involves figuring out what the relationship between input data and a given set of bytes is?
  - What byte values are forbidden in my shellcode?
  - What mangling is done on my input data?
  - What are the bounds on this write-4 address?
  - What are the bounds on X, where X is any numeric variable

A Collection of Problems

- Where is our data coming from and what conditions are on it?
  - Dataflow analysis, building path conditions
- What input do I need for variable X to equal value Y?
  - Theorem proving (Solving for satisfiability)
  - There are many similar problems we can solve by addressing this one

Agenda

- Static versus Dynamic dataflow analysis
- Taint Analysis
- Intermediate representations
  - ASM -> Intermediate Language
- Building logical formulae to represent program fragments
- Solving logical formulae
  - Solving for True/False
  - Solving for a satisfying input

Static vs. Dynamic Analysis

- For most program analysis problems this is our first question
  - Realistically many problems are best approached with a combination of both
- Tradeoffs to both
- Suitability depends on the problem at hand and the time one is willing to invest

Static Analysis

- Analysing code without running
- Imprecise by nature as many problems are undecidable in the general case
  - Loop/Program termination for example
- 'Solving' undecidable problems involves compromise
  - Conservative analysis -> False positives
  - Unsafe analysis -> False negatives
- Can give much more general (in a good way) answers than dynamic analysis

Dynamic Analysis

- Analysis of an executing program
- Restricted to the code that we can cause to be executed
- We can usually only ask questions regarding ‘this current path’ rather than ‘all possible paths’
- More precise by nature than static analysis but tradeoffs still exist
  - Program lag -> Is the problem you’re interested in time sensitive
  - Analysis storage -> Is the memory required by your analysis scaling linearly with the # instructions executed?
  - Generality of our results

Making a Choice

- What part of your workflow do you want to replace/assist/automate?
  - Will you settle for precise/instantly usable results at the cost of scope?
    - If you're replacing the human then probably no
    - If you're assisting the human then probably yes
  - Will you settle for answers only pertaining to this exact run or do you want generality over many/all paths
- Frameworks required versus frameworks available
- Time allocated

Dynamic Dataflow & Taint Analysis

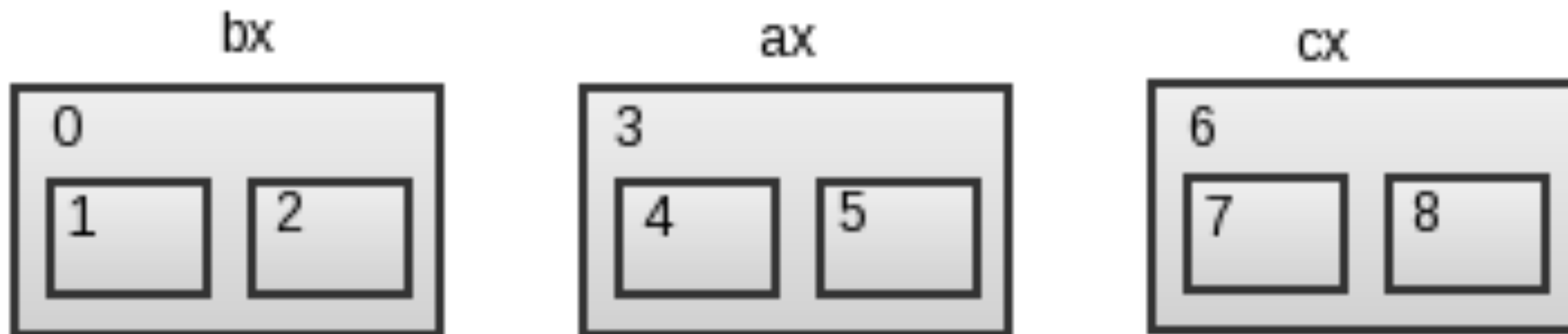
Tracing data and operations

- Instrumentation
  - Inserting analysis code into a running program
  - Won't be covered because it's really an entire other talk. See http://www.pintool.org to get started.
- Dataflow + Taint analysis
  - What information do we track/store and how do we do it
- Instruction semantics
  - How do we express instructions in terms of their dataflow semantics

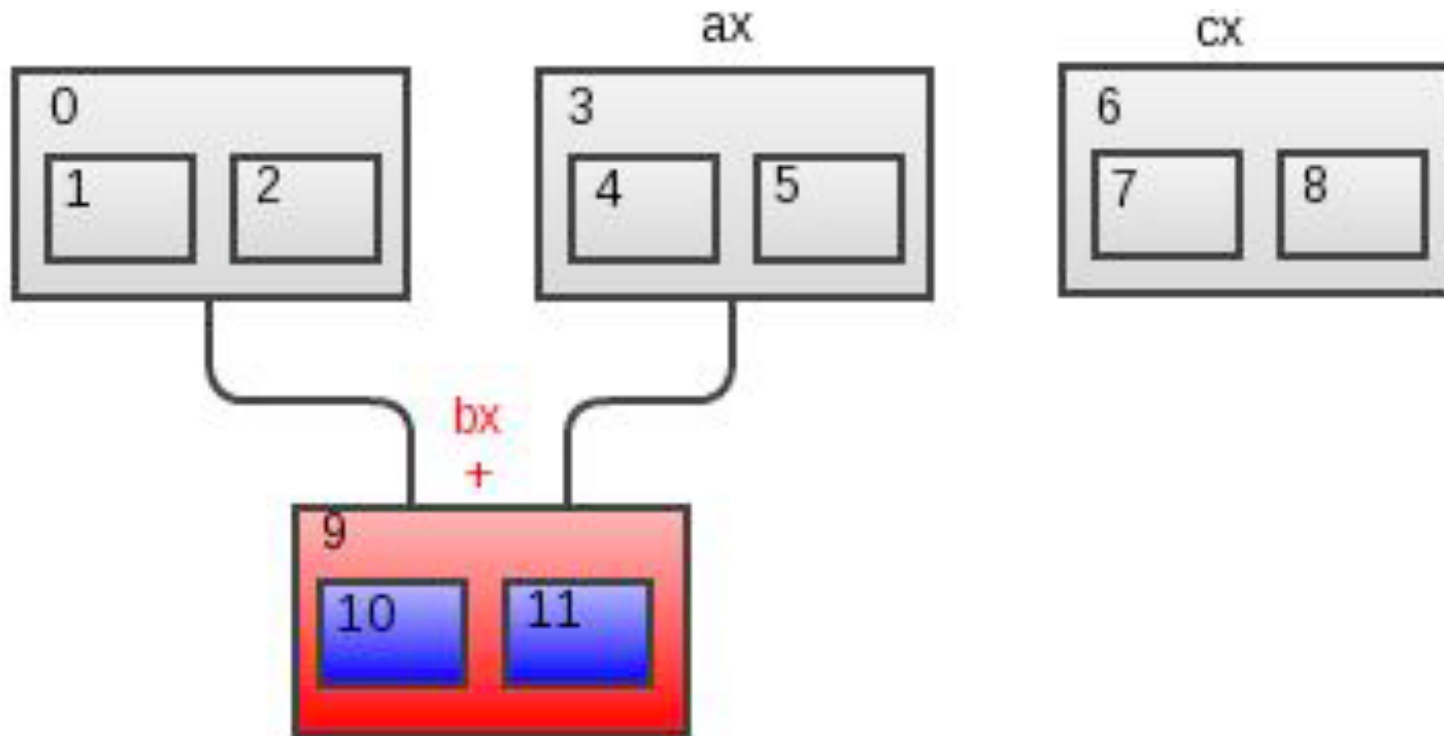
Dynamic Dataflow Analysis

- Essentially a question of expressing the dataflow semantics of an ASM instruction on an abstract model of a processes memory/registers
- Input – An ASM instruction, a model of the processes registers and memory
- Output – An updated model reflecting the effects of the instruction on our model
- In its pure form would provide a 'history' for every byte in memory in terms of all 'parent' bytes

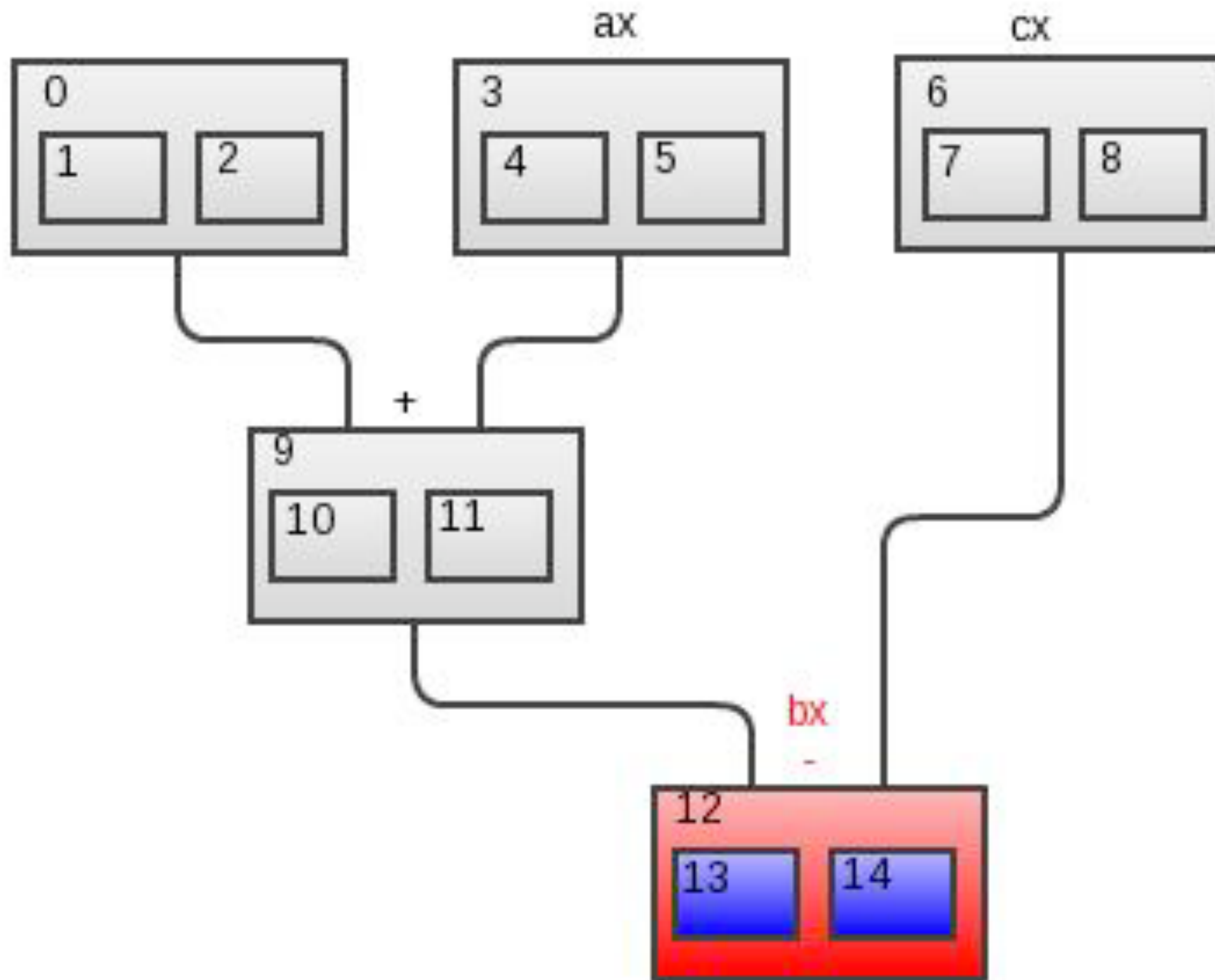
Basic Dataflow Example



add bx, ax



sub bx, cx



Taint Analysis

- DFA over all bytes in memory and all instructions is neither necessary nor practical
- Taint analysis is a more useful form
  - Tracking values under the influence of an attacker
- Our abstract model of memory/registers is essentially two disjoint sets mapping addresses/registers to TAINTED/UNTAINTED

Initialising the Tainted Set

- Hook read/recv/recvfrom etc system calls
- Alternatively (and preferably in many cases)
  - Model/Hook higher level wrappers that read in attacker data e.g. libc wrappers
- Tainting at a byte level
  - Every byte 'tainted' by user input is added to our TAINTED set
  - Why/why not bit level?
- Flags and Indirect tainting (is the return value of strlen(tainted_data) tainted?)

Propagating Taint Information

- Given an instruction *i*, a memory location or register *x* and the set of tainted locations *T*
  - Add *x* to the tainted set *T* *iff*

$$x \in i_{dsts} \wedge (y \in i_{srcs[x]} \mid y \in T) \neq \varnothing$$

  - *dsts* is the set of destinations for an instruction
  - *srcs[x]* is the set of sources affecting dst *x*

Propagating Taint Information

- Given an instruction *i,* a memory location or register *x* and the set of tainted locations *T*
  - Remove *x* from the tainted set *T iff*

$$x \in i_{dsts} \wedge (y \in i_{srcs[x]} \,|\, y \in T) = \varnothing$$

Adding to the Tainted Set

- We are not merely maintaining a set
- Remember the DFA example
- For every addition to this set we record a precise representation of the arithmetic relationship between the memory location and its ‘parents’

Um..wait..what?

- Where do *dsts* and *srcs* come from?
- Where does this ‘precise arithmetic relationship come from’?

ASM and Intermediate Representations

Modelling Dataflow Semantics

- We need an exact expression of the relationship between the sources and destinations of every instruction
- Can't automatically build this from parse tables etc
- What to do?
  - Model each and every ASM instruction (or until we run out of energy/will to live)

Intermediate Representations

- Writing instruction set specific analysis code is a bad idea for a number of reasons
  – Implicit operations mean repetitive work and potential inaccuracy e.g. updates to flags and other ‘side-effects’
  – Rewriting analysis code for every new instruction set doesn’t seem like fun
- We can create our IR such that it has properties not found in the original representation
  – Static single assignment form
  – Functional semantics

Intermediate Representations

```
For example, consider this x86 instruction:

  addl %eax, %ebx

One Vex IR translation for this code would be this:

  ------ IMark(0x24F275, 7) ------
  t3 = GET:I32(0)                 # get %eax, a 32-bit integer
  t2 = GET:I32(12)                # get %ebx, a 32-bit integer
  t1 = Add32(t3,t2)               # addl
  PUT(0) = t1                     # put %eax
```

From the Valgrind sources VEX/pub/libvex_ir.h

Properties of a typical IR

- Reduced instruction set
  - Intel x86 has > 200 instructions
  - REIL (Zynamics) has 17
- All implicit side effects of each instruction made explicit e.g. flag updates
- One-to-many relationship between native instructions and IR instructions
- Syntactic component vs. semantic component

Syntactic component

```
439B1250    ieproxy.dll::sub_439B1250
439B126C    add           esi, 4
439B126F    jmp           byte cs:loc_439B125B
```

REIL IR ->

```
439B126C00: and 4, 2147483648, t0
439B126C01: and esi, 2147483648, t1
439B126C02: add 4, esi, t2
439B126C03: and t2, 2147483648, t3
439B126C04: bsh t3, -31, SF
439B126C05: xor t0, t1, t4
439B126C06: xor t4, 2147483648, t5
439B126C07: xor t0, t3, t6
439B126C08: and t5, t6, t7
439B126C09: bsh t7, -31, OF
439B126C0A: and t2, 4294967296, t8
439B126C0B: bsh t8, -32, CF
439B126C0C: and t2, 4294967295, t9
439B126C0D: bisz t9, , ZF
439B126C0E: str t9, , esi
439B126F00: jcc 1, , 1134236251
```

Semantic component

- The syntactic component makes instruction effects explicit. We need a semantic component to interpret these on a model of memory/registers
- Every time a new variable is created we record its sources, whether they are tainted and the operation performed on these sources as an arithmetic or logical primitive
  - e.g. ASSIGN, AND, OR, NOT, ADD, SUB etc

Semantic component

```
VOID
x86Simulator::simMov_RM(ADDRINT memW, ADDRINT memWSize, UINT32 regId,
        THREADID tid, ADDRINT pc)
{
    SourceInfo si;
    if (!tmgr.isRegTainted(regId, tid)) {
        tmgr.unTaintMemLoc(memW, memWSize);
        return;
    }

    si.type = REGISTER_LOC;
    si.loc.regId = regId;

    vector<SourceInfo> sources;
    sources.push_back(si);

    TaintInfoPtr tiPtr;
    try {
        tiPtr = tmgr.createNewTaintInfo(sources, (unsigned)memWSize,
                DIR_COPY, X_ASSIGN, tid);
    } catch (IgnoreRegisterException &x) {
        tmgr.unTaintMemLoc(memW, memWSize);
        return;
    }
    tmgr.updateTaintInfoM(memW, tiPtr);
}
```

Analysis flow

Executing program ->

Instrumentation layer ->

Syntactic ASM transform ->

Application of IR semantics to memory model

Querying memory model -> ???

And this is useful because?

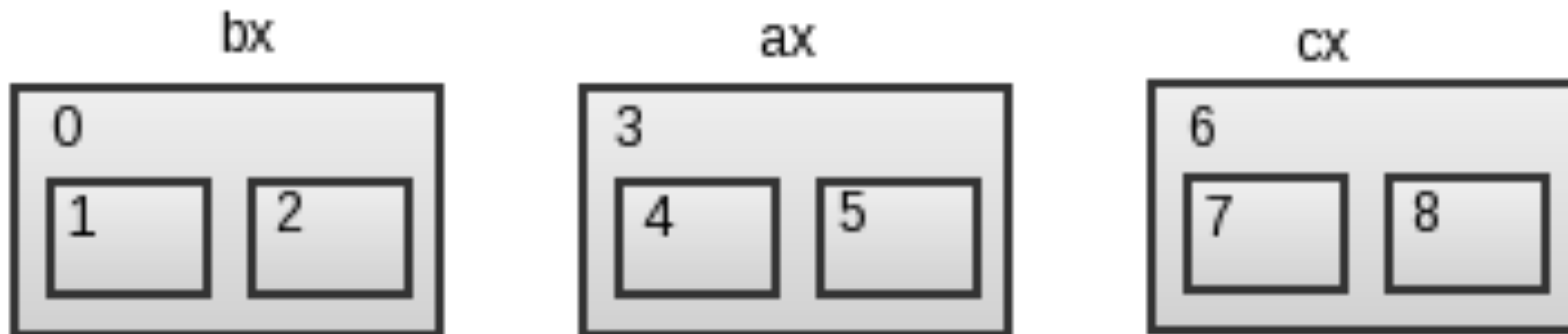
- We can answer the first question:
  - What locations are *tainted* by user input?
- Info is available to answer the next three with some processing:
  - What exact bytes of user input?
  - What computations were done on these bytes?
  - What conditions have been imposed on these bytes?

Post-Execution Processing

Building a Path Condition

- A path condition is a logical representation of the executed code (including conditionals)
- Essentially a formula relating input data to live memory locations or registers
- Built from the semantic analysis of each executed instruction
- This will express the answer to these questions:
  - What exact bytes of user input?
  - What computations were done on these bytes?

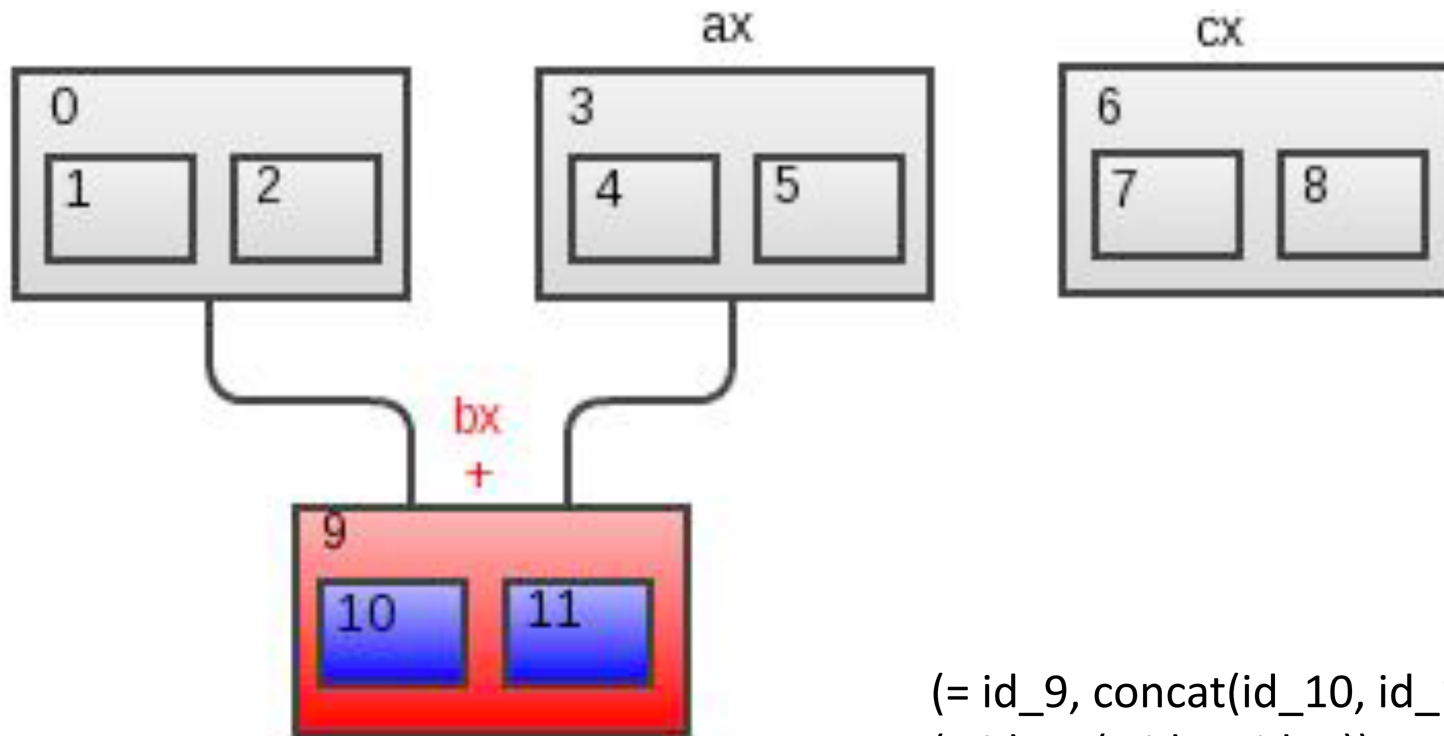
Building a Path Condition



Declare id_1, id_2, … as BitVector[8]
Declare id_0, id_3, … as BitVector[16]

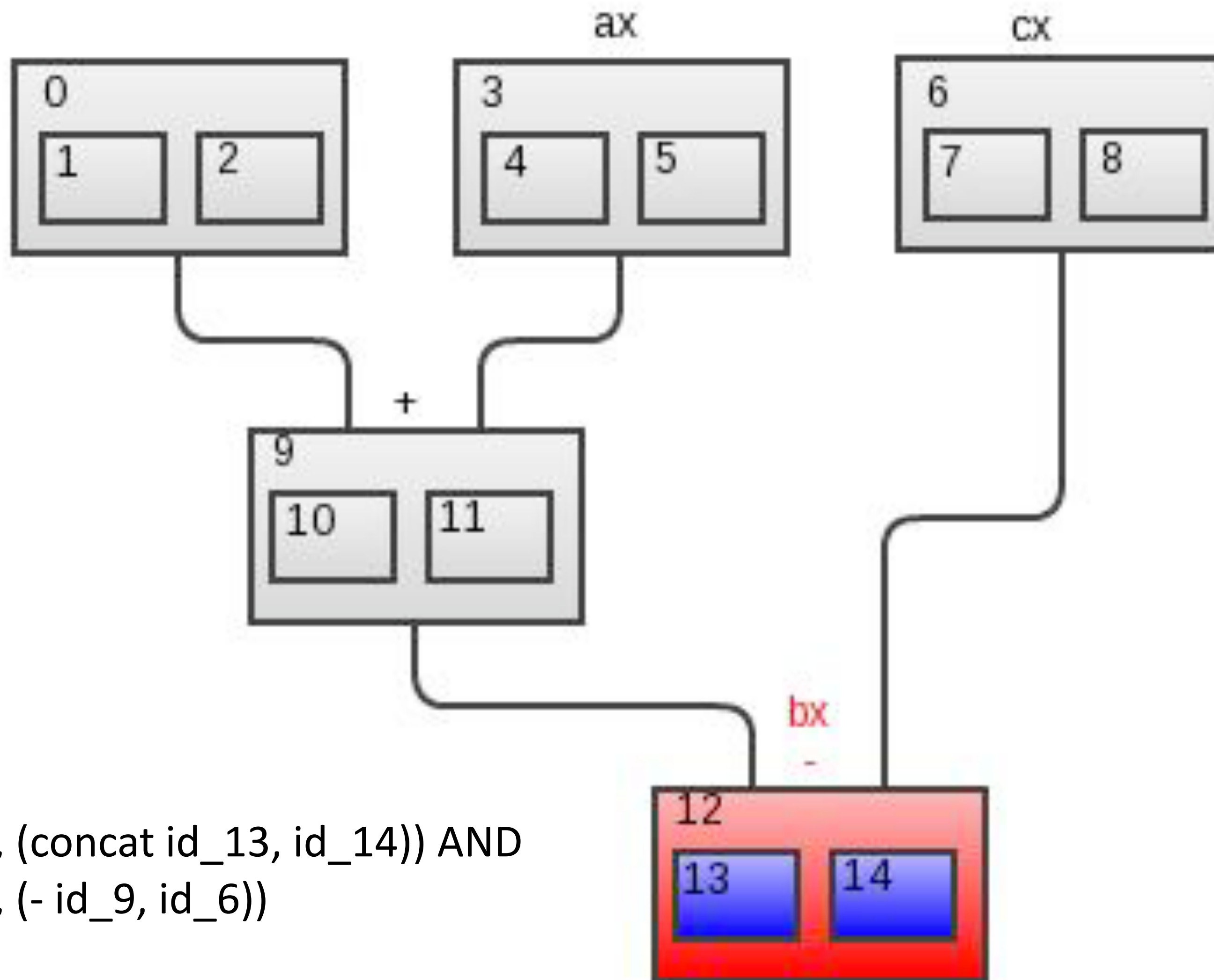
(= id_0, (concat id_1, id_2)) AND
(= id_3, (concat id_4, id_5)) AND
(= id_6, (concat id_7, id_8))

add bx, ax



(= id_9, concat(id_10, id_11)) AND
(= id_9, (+ id_0, id_3))

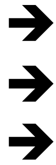
sub bx, cx



(= id_12, (concat id_13, id_14)) AND
(= id_12, (- id_9, id_6))

Dataflow as a ‘formula’

add bx, ax
sub bx, cx



Declare id_1, id_2, … as BitVector[8]
Declare id_0, id_3, … as BitVector[16]

(= id_0, concat id_1, id_2)) AND
(= id_3, concat id_4, id_5)) AND
(= id_6, concat id_7, id_8)) AND
(= id_9, concat(id_10, id_11)) AND
(= id_12, concat id_9, id_6)) AND
(= id_9, (+ id_0, id_3)) AND
(= id_12, (- id_13, id_14))

Playing with Formulae

- We'll get to solvers and how they work soon
- For now lets assume we have a black box
  - INPUT: A formula with zero or more unbound variables
  - OUTPUT:
    - True/False depending on whether the formula is satisfiable
    - If 'True' then an assignment to all unbound variables that makes the formula satisfiable

What can we do with this formula?

- Answer questions on output values given we control input values

```
(= id_0, XXX) AND (= id_3, 4) AND (= id_6, 8) AND
(= id_9, (+ id_0, id_3)) AND (= id_12, (- id_9, id_6))
```

- No real advantage to solving this formula with a solver versus running the code on a CPU though

What can we do with this formula?

- Query input values required for a given output value

```
(= id_9, (+ id_0, id_3)) AND (= id_12, (- id_9, id_6)) AND
(= id_12, 10)
```

- More interesting than the previous case as we can't really do this without a solver of some kind

Adding Conditional Instructions

- Conditional jumps essentially introduce inequalities into our formula
- Necessary for accurate solutions
- Simple to derive if you have an IR
  - Flag modifications are explicit in our IR therefore we can track the exact variables involved in setting them

(For our sanity and brevity we won’t be using a full IR in the following examples)

Adding Conditional Instructions

```
add bx, ax
sub bx, cx
cmp bx, 10
jg target
…
target:
```

```
(= id_9, (+ id_0, id_3)) AND (= id_12, (- id_9, id_6)) AND
(= id_12, 10) AND (> id_12, 10)
```

Incomplete Transition Tables

```
add bx, ax
sub bx, cx
cmp bx, 10
jg target
    mov ax, 0
    jmp exit
target:
    mov ax, bx
exit:
    …
```

```
(= id_9, (+ id_0, id_3)) AND (= id_12, (- id_9, id_6)) AND
(= id_12, 10) AND (<= id_12, 10) AND (= id_15, 0)
```

```
(= id_9, (+ id_0, id_3)) AND (= id_12, (- id_9, id_6)) AND
(= id_12, 10) AND (> id_12, 10) AND (= id_15, id_12)
```

Incomplete Transition Tables

- Essentially we have no representation of what occurs on the untaken side of conditions
- One of the main drawbacks of purely dynamic analysis
- If our appended constraints require such a path to be taken the solver will return ‘unsatisfiable’
- Solving this problem dynamically is messy

Using a Solver to Drive Execution

- So we've no idea what happens on the other side of that condition....
- What if we use the following to generate an input?

```
(= id_9, (+ id_0, id_3)) AND (= id_12, (- id_9, id_6)) AND
(= id_12, 10) AND (<= id_12, 10)
```

See SAGE research from Microsoft and FuzzGrind (open source)

Solving Formulae

- By creating and solving formulae we therefore can produce answers to the following:
  - Give me the input values a, b, c such that the output variables have values x, y, z, etc.
  - Give me the output values for variables x, y, z were I to restrict the input variables a, b, c to A, B and C
  - Give me an input that takes a different path at condition C
- How do we solve these formulae?

Theorem Proving

Solving Formulae/Theorem Proving

- We've been glossing over some details :)
  - How does one represent these formulae?
  - How do you solve non-toy examples? e.g A thousand variables and ten thousand clauses
  - How do we interact with these solvers?
- But first… a brief diversion into 1st year logic :)

Propositional logic

- Punctuation e.g. ()
- Propositional symbols e.g. *p, q, r, s* etc
- Connective symbols e.g. $\wedge, \vee, \neg, \rightarrow$
- Syntactic rules e.g. a proposition or a formula must occur on both sides of the symbol 'v'
- Axioms e.g. $\phi \rightarrow \phi \vee \chi$
- Transformations rules – replacement/ detachment

Truth tables

- The interpretation of boolean symbols can be defined via truth tables

p	q	p ^ q
T	T	T
F	T	F
T	F	F
F	F	F

Truth/Satisfiability

- Is there an assignment to the variables to make the following formula true (satisfiable)?

$$(a \vee b \vee \neg c) \wedge (b \vee c) \wedge \neg c$$

- How did you decide?

A Basic Approach

- From a formula with N variables there are 2^N possible interpretations
- This set is recursively enumerable therefore the solution is effectively computable
- Obvious solution? Truth tables

F: $(a \vee b \vee \neg c) \wedge (b \vee c) \wedge (\neg c)$

a	b	c	F
T	T	T	F
T	T	F	T
T	F	T	F
T	F	F	T
F	T	T	F
F	T	F	T
F	F	T	F
F	F	F	T

The DPLL algorithm

- The previous approach is provably correct but quite useless for real problems
- The DPLL algorithm provides the base for most modern solvers
- Essentially a heuristic search through a MASSIVE state space
  - For details ask me later or check out the links at the end

Um…

- Our formula is quite obviously not in propositional logic

```
(= id_9, (+ id_0, id_3)) AND (= id_12, (- id_9, id_6)) AND
(= id_12, 10) AND (> id_12, 10)
```

- We have a propositional skeleton but the rest will require a higher order logic

SMT Solvers

- DPLL algorithm with a theory specific solver
  - e.g. the theory of linear arithmetic, theory of arrays/lists, theory of bit-vectors
- The theory specific solver handles conjunctions of clauses in its theory when requested by the DPLL algorithm
- Essentially we now know that our formulae can actually be solved given an implementation of DPLL(T)

Analysis flow

Executing program ->

Instrumentation layer ->

Syntactic ASM transform ->

Application of IR semantics -> memory model

Querying memory model ->

SMT-LIB formula

$$(A = B) \wedge (C = 10) \wedge (D = A + C) \wedge (E = D)$$

```
(benchmark test
:status unknown
:logic QF_BV

:extrafuns ((a BitVec[8])(b BitVec[8])(c BitVec[8])
  (d BitVec[8])(e BitVec[8]))

:assumption (= a b)
:assumption (= c bv10[8])
:assumption (= d (bvadd a c))
:assumption (= e d)

:formula (= e bv20[8])
)
```

Solver(formula) -> satisfying assignment

```
$ ./yices -e -smt < new.smt
sat
(= b 0b00001010)

(= i0 0b11101011)
(= i1 0b00011000)
(= i2 0b01011110)
(= i3 0b10001001)
...
```

Exploit Development

Detecting Memory Corruption

- Other ways to do this (PageHeap etc) but usually sufficiently imprecise to miss subtle cases
- Directly tainted EIP
  - Probably a good sign mischief is afoot
- Tainted read/write addresses
  - False positives?
    - Let the solver take care of that

Locating Potential Shellcode Buffers

- Can track arbitrary input and dump lists of potential buffers at any point in programs execution
- We also have access to the complete history of every byte in each buffer
- Simple to find the least restricted/mangled buffer of user controllable input
  - Consider the RE effort involved in doing this manually

Rewriting Shellcode to Undo Mangling

- We can use a solver to ‘undo’ arithmetic mangling quite easily
- Given shellcode S, user input X and mangling function M we want M(X) = S
- Simple case
  - A loop containing add x, 4 for all bytes x in X
  - Given the constraint M(X) = S a solver will produce (x – 4) for all x in X

Exploit Generation

- A subset of exploits can be concisely expressed by appending conditions to a formula built as previously described and automatically generated
- Constraining write/read/return addresses
- Constraining the shellcode

http://www.cprover.org/dissertations/thesis-Heelan.pdf

Conclusion

Summary

- By tracking tainted data we can make reverse engineering of running/crashing programs a lot easier
- Tracking tainted data is a pretty simple matter
  - Instrumentation + IR + Dataflow Semantics
- Post-processing of the tracked data allows us to build formulae representing instruction semantics
- Solving formulae is useful for a bunch of fun stuff :)

Annoyances

- Dynamic dataflow analysis
  - Quite slow
  - By its nature leaves us with an incomplete picture
- Theorem proving
  - Can take several hours to terminate (assuming we can even guarantee completeness) for certain tasks
- Infrastructure
  - Until someone releases a more complete/integrated set of tools there's quite a lot of setup

Future Work

- Combining dataflow analysis/theorem proving with existing tools e.g. Immunity Debugger
- Integration with static analysis toolkits will make for better dynamic and static analysis
  - e.g. using dynamic analysis to reduce false positives and using static analysis to optimise dynamic tracing
- Hopefully more useful/ambitious tools in general (See William Whistlers talk later today)

Questions

sean@immunityinc.com http://twitter.com/seanhn

Links

- http://www.unprotectedhex.com/psv
- http://www.reddit.com/r/reverseengineering