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/seanhhn
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
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>708FC961</td>
<td>mov</td>
<td>ds:[edx], eax</td>
</tr>
<tr>
<td>708FC963</td>
<td>jmp</td>
<td>byte cs:loc_708FC967</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>708FC967</td>
<td>cmp</td>
<td>ds:[edx], 0</td>
</tr>
<tr>
<td>708FC96A</td>
<td>jz</td>
<td>byte cs:loc_708FC976</td>
</tr>
</tbody>
</table>
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 $T \iff$

    $$x \in i_{dsts} \land (y \in i_{srcs[x]} | y \in T) \neq \emptyset$$

  – $dsts$ 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

\[
x \in i_{\text{dsts}} \land (y \in i_{\text{srcs}[x]} \,|\, y \in T) = \emptyset
\]
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:

\[ \text{addl \%eax, \%ebx} \]

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

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.
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]

$(= id_0, (\text{concat } id_1, id_2))$ AND
$(= id_3, (\text{concat } id_4, id_5))$ AND
$(= id_6, (\text{concat } id_7, id_8))$
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’

\[
\begin{align*}
\text{add bx, ax} & \rightarrow \\
\text{sub bx, cx} & \rightarrow \\
\text{Declare id}_1, \text{id}_2, \ldots \text{ as BitVector}[8] & \rightarrow \\
\text{Declare id}_0, \text{id}_3, \ldots \text{ as BitVector}[16] & \rightarrow \\
(= \text{id}_0, \text{concat id}_1, \text{id}_2) & \text{ AND} \\
(= \text{id}_3, \text{concat id}_4, \text{id}_5) & \text{ AND} \\
(= \text{id}_6, \text{concat id}_7, \text{id}_8) & \text{ AND} \\
(= \text{id}_9, \text{concat(id}_10, \text{id}_11) & \text{ AND} \\
(= \text{id}_12, \text{concat id}_9, \text{id}_6) & \text{ AND} \\
(= \text{id}_9, \text{+ id}_0, \text{id}_3) & \text{ AND} \\
(= \text{id}_12, \text{- id}_13, \text{id}_14) & \text{ AND}
\end{align*}
\]
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

\[
(= \text{id}_0, \ XXX) \ \text{AND} \ (= \text{id}_3, 4) \ \text{AND} \ (= \text{id}_6, 8) \ \text{AND} \\
(= \text{id}_9, (+ \text{id}_0, \text{id}_3)) \ \text{AND} \ (= \text{id}_12, (- \text{id}_9, \text{id}_6))
\]

• 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

\( (= \text{id}_9, (+ \text{id}_0, \text{id}_3)) \ \text{AND} \ (= \text{id}_{12}, (- \text{id}_9, \text{id}_6)) \ \text{AND} \ (= \text{id}_{12}, 10) \)

• 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

```plaintext
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:
...

\[(= \text{id}_9, (+ \text{id}_0, \text{id}_3)) \text{ AND } (= \text{id}_12, (- \text{id}_9, \text{id}_6)) \text{ AND } (= \text{id}_12, 10) \text{ AND } (\leq \text{id}_12, 10) \text{ AND } (= \text{id}_15, 0)\]
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?

\[(= \text{id}_9, (+ \text{id}_0, \text{id}_3)) \text{ AND } (= \text{id}_{12}, (- \text{id}_9, \text{id}_6)) \text{ AND } (= \text{id}_{12}, 10) \text{ AND } (\leq \text{id}_{12}, 10)\]

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 1\textsuperscript{st} year logic :)
Propositional logic

- Punctuation e.g. ()
- Propositional symbols e.g. $p, q, r, s$ etc
- Connective symbols e.g. $\land, \lor, \neg, \rightarrow$
- Syntactic rules e.g. a proposition or a formula must occur on both sides of the symbol ‘$\lor$’
- Axioms e.g. $\phi \rightarrow \phi \lor \chi$
- Transformations rules – replacement/detachment
Truth tables

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

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p ^ q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Truth/Satisfiability

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

\[ (a \lor b \lor \neg c) \land (b \lor c) \land \neg c \]

- 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
\( F: (a \lor b \lor \neg c) \land (b \lor c) \land (\neg c) \)
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

\[(= \text{id}_9, (+ \text{id}_0, \text{id}_3)) \text{ AND } (= \text{id}_12, (- \text{id}_9, \text{id}_6)) \text{ AND } (= \text{id}_12, 10) \text{ AND } (> \text{id}_12, 10)\]

- 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
\[(A = B) \land (C = 10) \land (D = A + C) \land (E = D)\]

(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

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