WHO AM I

• John Villamil
  • Senior Research Scientist at Accuvant
• Previously at Matasano
• Started at Mandiant

• jvillamil@accuvant.com
• @day6reak
ONCE UPON A TIME…

- I asked a question on Reddit - [http://redd.it/smf4u](http://redd.it/smf4u)
- Got some amazing answers

a different approach, from someone with limited time and attention span:
pick one thing that really seems magical or interests you, and go at it, forgetting everything else
examples:

- what the heck is LLL doing? how is it possibly solving these knapsack instances?
- what is this SAT? could it help me in finding out XXX?
- how is it that some squiggy curve on a graph can be used to encrypt stuff?

the volume of available subjects and material is itself a hinderence; ignore it. collecting a bunch of foundational
material just adds to the size of the endeavor; ignore it. go to wikipedia immediately and read … some junk you
don't understand? cool now subject is more mysterious and challenging! it will be even more rewarding once you've
figured it out. let it haunt you while you drive, while you wait in line, etc.

experimenting and playing are more important than reading; download minisat and try to convert some equation to
dimacs and see it work right away; now tweak it, change it, play with it; now read a bit more; now play more

I started from scratch on the formal CS side, with an emphasis on program analysis, and taught myself the
following starting from 2007. If you're in the United States, I recommend BookFinder to save money buying these
things used.

On the CS side:

- Basic automata/formal languages/Turing machines; Sipser is recommended here.
- Basic programming language theory; I used University of Washington CSE P505 online video lectures and
  materials and can recommend it.
- Formal semantics; Semantics with Applications is good.
- Compilers. You'll need several resources for this; my personal favorites for an introductory text are Appel's
  ML book or Programming Language Pragmatics, and Muchnick is mandatory for an advanced understanding.
  All of the graph theory that you need for this type of work should be covered in books such as these.
- Algorithms. I used several books; for a beginner's treatment I recommend Dasgupta, Papadimitriou, and
  Vazirani; for an intermediate treatment I recommend MIT's 6.046J on Open CourseWare; for an advanced
treatment, I liked Algorithmics for Hard Problems.
THE NEED FOR VERIFICATION

• Verification shifted coding mentality
  • Before – fix it until it worked
  • After – prove it works the way it is intended

• Intended by whom? By what standards?
• What exactly is being verified?
• What does the verification rely upon? There must be some assumptions.
  • Other software
  • Instruction set
  • Hardware
Miscellaneous

- What kind of programs lend themselves to solver based bug finding?
- Solvers vs compression/encryption
- The amount of available research material is staggering and parsing it is time consuming
- My suggestion is start coding and get something crappy and broken up and running as fast as possible

"Use the force, Harry"

- Gandalf
ROBERT FLOYD

- The foundation of verification research was set in the 60s and 70s.
- “Assigning Meaning to Programs” – 1967
  - Made a strong relation between coding and math
  - Defines “preconditions”, “postconditions”, and “counterexamples”
- $Vc(P, Q)$
  - Verification condition holds if $P$ and $Q$ hold
  - There can be several $P$s and $Q$s for a program.
    - Multiple ways of reaching a conclusion
    - Multiple conclusions for a given precondition
2 YEARS LATER - HOARE LOGIC

- The Hoare Triple
  - \( \{P\}C\{Q\} \) originally written as \( P\{C\}Q \)
    - \( P \) – precondition
    - \( Q \) – postcondition
    - \( C \) – statement
  - Claims the following - If a precondition is satisfied for a specific state, then that state will satisfy the postcondition on termination
    - \( P \implies Q \)
  - Valid – \( \{Y+Z=1\} X=Y+Z \{X==1\} \)
  - Invalid – \( \{a=7\} a=a+3 \{a==8\} \)
  - To verify - negate \( Q \) and prove it is unsatisfiable. Why?
  - Hoare logic “flows”
    - The postcondition of a statement carries on to the precondition of the next statement
WEAKEST PRECONDITION – DIJKSTRA 75’

• \{P\}C\{Q\}
  • The least restrictive precondition
  • There are multiple “P” which satisfy “Q”
  • Weakest possible precondition is [True]
  • If Q \textit{implies} P then P is \textit{weaker} than Q

```cpp
if (x == y)
    y = x * 5 + 1;  // x*5+1 > 1 or \{x > 0\}
else
    y = x + 5;     // x+5 > 1 or \{x > -4\}
{y > 1}
```
WEAKEST PRECONDITION

- Great for ROP as seen in “Q: Exploit Hardening Made Easy”
- wp verification technique shows whether a gadget satisfies a precondition
- Equivalence checking is also used

```asm
addr 0x0 @asm "inc %eax"
label pc_0x0
T_t:u32 = R_EAX:u32
R_EAX:u32 = R_EAX:u32 + 1:u32
R_OF:bool = high:bool((T_t:u32 ^ -2:u32) & (T_t:u32 ^ R_EAX:u32))
R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EAX:u32 ^ T_t:u32 ^ 1:u32))
             R_EAX:u32 >> 1:u32 ^ R_EAX:u32)
R_SF:bool = high:bool(R_EAX:u32)
R_ZF:bool = 0:u32 == R_EAX:u32
addr 0x1 @asm "ret"
label pc_0x1
T_ra:u32 = mem:?u32[R_ESP:u32, e_little]:u32
R_ESP:u32 = R_ESP:u32 + 4:u32
```

Is eax_before equal to eax_after+1?
SYMBOLIC EXECUTION

• Instead of concrete values, variables represent all possible values of their type.
• Check out http://shell-storm.org/blog/Binary-analysis-Concolic-execution-with-Pin-and-z3/

```c
unsigned int foo(unsigned int bar){
    unsigned int a = bar;
    unsigned int y = 1;

    if (a < 100){
        y += a;
        return y;
    }

    else{
        return y;
    }
}
```

//Where foo is symbolic user input
//"a" is any possible value of an unsigned int
//A new state is created with the assertion of
//(a < 100).  y is now a symbolic value
//equal to (a + 1 < 100)
//This second state has the assertion (a >= 100)
//"y" is a concrete value of 1
CONSTRAINT PROGRAMMING

- SAT – boolean satisfiability problem
  - Can variables be assigned in such a way as to make a formula evaluate to true?
  - Uses logical operators $\land$, $\lor$, $\neg$
- Satisfiable is a truth assignment where the formula that evaluates to true
- Unsatisfiable if a solution cannot be found
- The basic and naive SAT algorithm is an unintelligent state exploration
  - If the algorithm can’t yet terminate with a TRUE or FALSE, reiterate and negate a variable
  - There are many optimizations and rules that speed it up
CNF AND DIMACS

• Conjunctive Normal Form
  • Just a normalized expression of the logical operators
  • Each clause is a disjunction
  • ie. \( \neg A \land \neg B \) instead of \( \neg (A \lor B) \)
  • Conversion to CNF can be automated
  • Simplifies SAT operations

• Dimacs
  • CNF ASCII file format most SAT solvers can parse

\[
\begin{align*}
(x(1) \lor (\neg x(3))) \\
\text{AND} \\
(x(2) \lor x(3) \lor (\neg x(1)))
\end{align*}
\]

• The above in dimacs looks like →

\[
\begin{align*}
c & \text{ simple_v3_c2.cnf} \\
p & \text{ cnf 3 2} \\
1 & -3 0 \\
2 & 3 -1 0
\end{align*}
\]
DPLL

- Optimization on the basic SAT algorithm
  - Boolean Constraint Propagation
  - Backtracking and Backjumping
  - Pure-Literal elimination
  - And more… See open source solvers for details
- Other optimizations have improved solver performance considerably
  - Smarter literal selection – scoring
  - Propagation of constraints to other clauses – clause learning
  - Random restarts – refreshing the state while retaining all the learned knowledge so far
- Conflict Driven Clause Learning (CDCL)
USING SAT

- See https://tuts4you.com/download.php?view.3293 by Dcoder and andrewl
- Kao’s Toy Crackme

```c
void expand(u8 B[32], const u8 A[32], u32 x, u32 y)
{
    u32 i;
    for(i=0; i < 32; ++i)
    {
        out[i] = (in[i] - x) ^ y;
        x = ROL(x, 1);
        y = ROL(y, 1);
    }
}
```

- Advantages of using a solver here
CONSTRAINT PROGRAMMING

- SAT is still the basis
- SMT – Satisfiability Modulo Theories
  - SAT with domain theories – integers, arrays, bitvectors
  - No more need for CNF! SMT-LIB(2) is the standard
- See http://rise4fun.com/z3/tutorialcontent/guide for tips on using the grammar
  - Be aware they extend the SMT-LIB2 grammar – some features aren’t found in other solvers.
EXAMPLES
EXAMPLES

• Formulas have…
  • Declarations
  • Precondition
  • Postcondition
• Formula are resolved with a…
  • Sat
  • Unsat
  • Unknown
EXAMPLES

- Polish Notation
- Z3py output is interfix notation
- Most things are functions, even constants

```
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const d Real)

(assert (> a (+ b 2)))
(assert (= a (+ (- 3 c) 10)))
(assert (<= (- c b) 1000))
(assert (> (to_real a) d))
(assert (< a (+ d (to_real b))))

(check-sat)
(get-model)
```
EXAMPLES

- Modeling is not programming
- No code is executed
- ‘=’ states equivalence as opposed to variable assignment

(declare-const x Int)
(declare-const y Int)
(assert (= x 1))
(assert (= x y))
(check-sat)
(get-model)

$z3 -smt2 sat.txt
sat
(model
  (define-fun y () Int 1)
  (define-fun x () Int 1)
)

EXAMPLES

- Bitvectors
- http://smtlib.cs.uiowa.edu/theories/FixedSizeBitVectors.smt2
- And of course http://rise4fun.com/z3/tutorialcontent/guide

```
(set-logic BV)
(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))
(declare-fun z () (_ BitVec 32))

(assert(
    and (= x y) (= z (bvsub x y)))))

(assert(bvugt z #x00000005 ))

(check-sat)
(get-model)
```
EXAMPLES

- Bitvector types cannot be mixed
- BVs must be the same size
  - Zero Extend – ZeroExt()
  - Concat – Concat()
  - Sign Extend – SignExt()
  - Extract – Extract()
- Small demo using bvs in Z3py
- Other data types – Quantifiers, Arrays,
TOWARD WHITEBOX FUZZING
SEMI-WHITEBOX

- American fuzzy lop from lcamtuf
  - https://code.google.com/p/american-fuzzy-lop/
- Bunny-the-fuzzer
  - https://code.google.com/p/bunny-the-fuzzer/
- Flayer
  - https://code.google.com/p/flayer/
- Roll your own with PIN or DynamoRIO
  - Modify a code coverage tool
WHITEBOX

- Best of both worlds – verification and fuzzing
- Implementation challenges
  - Tracking untrusted data propagation through a program
  - Checking if new paths can be taken
    - Path explosion
  - Modifying untrusted data to take those new paths
- Requires significant engineering
  - There are a lot of tools that can help!
OPEN SOURCE

• Avalanche
  • Uses Valgrind
  • Taintgrind to track user data
    • Based on Flayer
  • Libvex for an intermediate representation
  • Parses VEX IR and constructs a model for STP
  • Not a fuzzer
  • Checks for null derefs but also uses memcheck to find errors
OPENSOURCE

- Fuzzgrind
  - Uses Valgrind
  - Plugin to track tainted data
    - Based on Flayer
  - Parses VEX IR and constructs a model for STP
  - Checks if new path can be taken
  - Fuzzes input file for paths it can satisfy
GENERIC WORKFLOW
TAINT TRACING

- Based on simple rules
- Track registers and memory accesses
- If source of an operation is tainted, the destination becomes tainted
- Implementations:
  - BitBlaze
  - Dytan
  - PrivacyScope
  - Libdft
  - Minemu
  - Many others
- Lot of research papers and info out there
TAINT TRACING

- PIN is popular
  - Dynamic Binary Instrumentation tool (DBI)
    - Much cleaner API compared to Dynamo RIO but slower in some cases
  - XED2 engine builds a structure of information for each disassembled instruction
  - Simple to query to see instruction type, operands, memory references etc.
  - Facilitates syscall and function hooking
**Taint Tracing**

- Build a list of function pointers indexed by the XED2 instruction type for a fast lookup.
  
  ```c
  array_of_ins_types[XED_ICLASS_ADD] = &your_ADD_function;
  ```

- Call your taint function with that instruction’s info
  
  ```c
  INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(your_ADD_TAINT_func),
                 IARG_ADDRINT, INS_OperandReg(ins, 0),
                 IARG_PTR, dest,
                 IARG_END);
  ```

- Use global data types for simplicity, “map” works great for registers and memory
IR TRANSLATION

- Libvex
- BAP
  - Toil to lift assembly to IR
  - Iltrans to modify, simplify, optimize the IR

```
addr 0x5 @asm "sub %ecx,%eax"
label pc_0x5
R_EAX_129:u32 = R_EAX_118:u32 - R_ECX:u32
R_EAX_160:u32 = R_EAX_129:u32
R_ESP_165:u32 = R_ESP_124:u32
addr 0x7 @asm "neg %eax"
label pc_0x7
R_EAX_170:u32 = 0:u32 - R_EAX_160:u32
R_CF:bool = if R_EAX_160:u32 == 0:u32 then false else true
R_EAX_200:u32 = R_EAX_170:u32
R_ESP_204:u32 = R_ESP_165:u32
addr 0x9 @asm "sbb %eax,%eax"
label pc_0x9
R_EAX_249:u32 = R_EAX_212:u32
R_ESP_253:u32 = R_ESP_204:u32
```
HANDLING THE IR

- The goal is to construct a model that can be used by a solver
- Negate constraints for untraversed paths

```c
 case Ist_Store:
  data = st->Ist.Store.data;
  tl_assert(isIRAtom(data));
  tl_assert(isIRAtom(st->Ist.Store.addr));

  if (data->tag == Iex_RdTmp) {
    j = bb->tyenv->types[data->Iex.RdTmp.tmp];
    size = (j != Ity_I1) ? sizeofIRType(j) * 8 : 1;
  }
  else { // data->tag == Iex_Const
    j = typeOfIRConst(data->Iex.Const.con);
    size = (j != Ity_I1) ? sizeofIRType(j) * 8 : 1;
  }

  add_dirty2(helperc_store,
              (st->Ist.Store.addr->tag == Iex_Const) ? mkIRExp
              mkIRExpr_HWord((data->tag == Iex_RdTmp) ? data->
              break;
```

Fuzzgrind
fz_translate.c
VEX details in libvex_ir.h
TAINT NO FUN

- tainting ffmpeg –i <infile> <outfile>
  - On 1mb input file produces 3.4gig of tainted trace output
  - Takes 22 minutes to complete compared to 0.4s without tracing
- Good hardware is usually thrown at the problem
- In general, whole program taint propagation and constraint solving was not intended for fuzzing
  - It takes too much…
    - Space
    - Speed
    - Symbolics
- Every byte can be treated as a symbol
KEEP IT SIMPLE

• Work iteratively – parse paths one by one – no need to taint propagate the entire program at once
• Have a threshold – if the number of new instructions exceeds the threshold then it’s a sample!
• Customize your tools for the target

Live Long, and Prosper
- Han Solo