The Automated Exploitation Grand Challenge

Tales of Weird Machines

Julien Vanegue

julien.vanegue@gmail.com

H2HC conference, Sao Paulo, Brazil
October 2013

Acknowledgements

I am indebted to many people for their work on Program Analysis and Exploitation:

Julio Auto, Thomas Ball, Dion Blazakis, Pascal Bouchareine, Sergey Bratus, Nicolas Brito, Michal Chmielewski, Gynvael Coldwind, Solar Designer, Mark Dowd, Thomas Dullien, Sergiusz Fonrobert, Mathieu Garcia, Thomas Garnier, Travis Goodspeed, Patrice Godefroid, Sean Heelan, Ronald Huizer, Vincenzo Iozzo, Barnaby Jack (R.I.P.), JP, Ken Johnson, Mateusz Jurczyk, Michel 'MaXX' Kaempf, Tim Kornau, Kostya Kortchinsky, Sebastian Krahmer, Joshua Lackey, Shuvendu Lahiri, Eric Landuiyt, Xavier Leroy, Felix 'FX' Lindner, Tarjei Mandt, Damien Millescamps, Matt Miller, John McDonalds, David Molnar, Julien Palardy, Enrico Perla, Paul @pa_kt, The PaX Team, Willem Pinckaers, Rolf Rolles, Gerardo Richarte, Dan Rosenberg, Sebastien Roy, Fermin Serna, Scut, Sysk, Alex Sotirov, Julien Tinnes, Richard Van Eeden, Ralf-Phillipp Weinmann, David Weston, Rafal Wojtczuk, Michael Zalewski.

You are an inspiration for the Automated Exploitation Grand Challenge

Thank you

What is Automated Exploitation?

- The ability to generate a successful computer attack with reduced or entirely without human interaction.
- It is important to understand the hardness of AE to measure the risk on critical infrastructure and online properties.
- There are many domains of attack: network, web, kernel, system, hardware, applications. Our focus today is on **software security.**

What are Weird Machines?

- Weird Machine (WM): “The underlying capacity of a program to perform runtime computations that escape the program specification”

(1) If extra computations are consistent with the intended program specification, the WM can lead to a covert execution of code within the program.

(2) If extra computations violate the intended program specification, the WM can lead to a security exploit (what we will talk about today).

Today’s exploits techniques

Modern history of exploit techniques :

- Code-reuse attacks: Computations without code injection
  - Started with “Return into Libc” (Solar Designer’s 1997)
  - Advanced by “Return into PLT” (Rafal Wojtczuk’s 1998)
  - Generalized by Chunk reuse (“borrowing”) technique (Richarte 2000, Krahmer 2005)
  - Since 2008, known as “Return Oriented Programming”
- Meta-data corruption
  - “Smashing C++ VPTRs” (Eric Landuiyt, 2000)
  - “V00d00 malloc tricks” (Michel Kaempf, 2001)
  - Many, many other papers.

Today’s exploits techniques (2)

- Information disclosure attacks
  - Format bugs (tf8 wu-ftpd 2.6 site-exec exploit, ~ 2000)
  - Weaknesses where content or address of target variables/functions can be read (BIND TSIG Exploit by LSD-PL, Openssl-too-open exploit by Sotirov, ~ 2001)
  - “Return into printf/send” (“Bypassing PaX ASLR protection”, Vanegue 2002)
- Heap chunks alignment techniques
  - “Advanced DL malloc exploit” (JP @ core-st , 2003)
  - “Heap Feng Shui”, (Sotirov, 2007)
- JIT attacks : make target generate “chosen” new code
  - “Pointer inference and JIT spraying” (Blazakis, 2010)

Exploit Mitigations

- Data Execution Prevention (DEP/Openwall/PaX/W^X/etc)
- Address Space Layout Randomization (ASLR)
- Control-Flow Integrity (CFI)
- Intra-modular displacement randomization (IDR)
- Heap randomization (non-deterministic fitness algorithms)
- Many others targeted protections (UDEREF, SEHOP, canary insertion, meta-data encoding, etc)

Full AE Models: The Automated Exploitation (AE) problem is solved if mitigations can be bypassed using minimal to no human interaction.

Restricted AE Models: Academic exercise where mitigations are ignored. Not the subject of this talk.

Control Flow Integrity

- Early implementation by Determina called "Program Shepherding" early 2000. Formalized by Martin Abadi et al. in 2005. A lot of work done at Microsoft, Intel, and more to make it practical – its hard.
- In a nutshell (idealized) :
  - Enforce strict transitions on the control flow graph, in particular between functions.
  - If A → B and B –ret→ A , then A → C is forbidden, so is B –ret→ D (for C and D any other two functions)
- Consequence: Exploit cannot easily corrupt a function pointer or a return address and execute a ROP payload.

Intra-modular Displacement Randomization

- [Miller, Johnson, Goel, Vanegue, 2011] at Microsoft Security. (http://ip.com/IPCOM/000210875)
- Core idea: randomize address space not only using module base address randomization, but also within a module (ex: between functions).
  - Ability to change function relative address from the base address every time a program is executed.
- Consequences:
  - Base address information disclosure is not enough to predict addresses of ALL gadgets in a module.
  - Attacker worst case: need one information disclosure per randomization point inserted in the module.

Exploit primitives

- Two major families of exploit primitives are write primitives (write address space) and read primitives (read address space).
- Early classification done by Gerardo Richarte at core-st : “About exploit writing”, 2002.
- Modern classification done by Matt Miller: “Modeling the exploitation and mitigation of memory safety vulnerabilities”, 2012.

Exploit write primitive

General form: *(basepointer + offset) = value

(1) Base, offset and values are attacker-controlled

→ Write controlled value at controlled location

(2) Base and offset are controlled

→ Write uncontrolled value at controlled relative location

→ With an information disclosure, can always be used to uncover new state space or elevate privileges

(3) Only RHS Value is controlled (totally or partially)

→ Write anything at fixed location

→ Can be useful if value is:

- Later used as base pointer, index, or offset (we fall into case 1 or 2)
- Used in a control predicate and can uncover new “weird” state space
- Controlling privilege level of application

Exploit read primitive

General form: value = *(basepointer + offset)

(1) Base, offset and values are attacker-controlled
→ Read value at desired location and store it at desired location

(2) Base and offset are controlled
→ Read value at desired location, store it at uncontrolled location
→ Only useful if uncontrolled location can be read by attacker

(3) Only LHS Value is controlled
→ Read internal program value and store it at desired location
→ Can be useful if value is:

- Internal program value is a direct code or data address
- Internal program value contains credentials (password, key, token, etc)
- Internal program value help deduce useful address info or credentials

Rising exploit techniques

- Data-only attacks (DOA)
  - Change internal program values to elevate privileges without changing Program control flow.
  - Infer address of data in program without direct memory read primitives.
- Program Likelihood Inference (PLI)
  - Probabilistic attacks: discover most likely executions to successful exploitation in non-deterministic environment.
  - Timing attacks: discover internal program information via run time execution measurements.

Tools Armory

Exploit Generation

- Automated Exploitation focuses on discovery and combination of write primitives and read primitives.
- Automated Exploitation in Full Model is a very hard problem. Anybody telling you otherwise is a fool or an impostor.
- Existing AE work focused on Restricted Models:
  - Sean Heelan's "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities" : http://www.cprover.org/dissertations/thesis-Heelan.pdf
  - David Brumley et al. (AEG, MAYHEM, etc)

http://users.ece.cmu.edu/~dbrumley/pubs.html

Analysis and Exploit Automation

- Compilers (Program transformation)
- Fuzz testers (Input generation)
- SMT solvers (Symbolic reasoning)
- Model Checkers (State space exploration)
- Symbolic Execution Eng (Path generation)
- Emulators (Machine modeling)
- Abstract interpreters (Abstraction)

SMT solvers

SMT = Satisfiability Modulo Theories

- Give it a list of variables and constraints on them, will tell you whether the set of constraints is satisfiable.
- A good representation to reason about a program (e.g. translate a program into an SMT formulae)
- Can track feasibility of predicates, eliminate impossible program paths, etc.

EXAMPLE 1: (B >= A) && (A <= B) is SAT

EXAMPLE 2: A && B && NOT(A&&B) is UNSAT

An open-source SMT solver : Z3

- Z3 is a state-of-the-art SMT solver developed in Microsoft Research RiSE group.
- Understand equalities, arrays, bitvectors, uninterpreted functions, and custom theories.
- Makes SMT a good symbolic representation to reason about programs (e.g. by translating it into SMT formulae).

Try by yourself on **http://rise4fun.com/Z3/**

Example of translation from C to SMT

```
F(int c)
{
  int ret;
  if (c < 10)
     ret = 1;
 else
   ret = 2;
}
```

```
(declare-fun x () Int)
(declare-const ret Int)
(declare-const c Int)
(assert (=> (>= c 10) (= ret 2)))
(assert (=> (<  c 10) (= ret 1)))
(assert (= ret 1)) // check me!
(check-sat)
(get-model)
```

Output model for constraint set

- A model is a valuation of the variables for which a (SMT) formula is true.
- In this example, the constraints set is satisfiable if variable C = 9 and RET = 1
- Change the last assertion of previous slide and see what happens to the model.

Z3 output:

```
sat (model
  (define-fun c () Int 9)
  (define-fun ret () Int 1) )
```

HAVOC: static analysis for C/C++

- HAVOC: Verifier for C(++) programs
  http://research.microsoft.com/en-us/projects/havoc/
- Translate C/C++ code to Boogie IR
  (Open source at: http://boogie.codeplex.com)
- Boogie IR is then translated to SMT formulae understood by Z3, which performs SMT check and give you a model.
- At Microsoft, HAVOC helped found 100+ security vulnerabilities in Windows and Internet Explorer.
- Experiments documented in: “Towards practical reactive security audit using extended static checkers” (Vanegue / Lahiri, 2013) http://research.microsoft.com/pubs/185784/paper.pdf

Problem: non-deterministic programs

Assume an attacker can overflow chunk 1 and chunk 3 is a target:

Heap in 90% of executions of program P :

S1 =

Chunk 1, Size S1, Addr A1

Chunk 2, Size S2, Addr A2 = A1 + S1

Chunk 3, Size S3, Addr A3 = A2 + S2

Heap in 10% of executions of program P :

S2 =

Chunk 1, Size S1, Addr A1

Chunk 3, Size S3, Addr A2 = A1 + S1

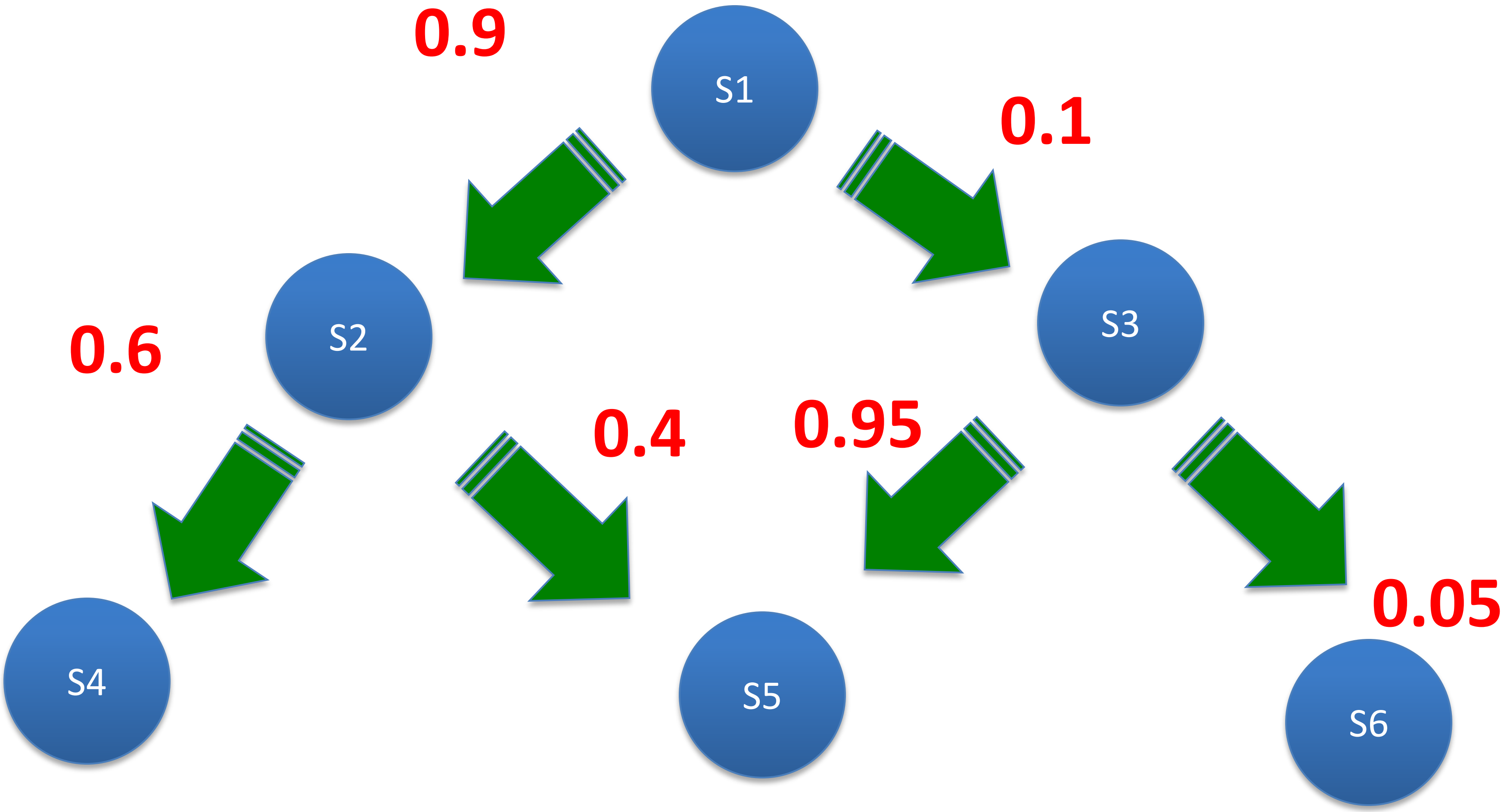
Chunk 2, Size S2, Addr A3 = A2 + S3

SMT solvers are unable to reason about non-determinism

Idea: Markov exploits

- Andrei Markov (1856-1922)
- Systems (Programs) may seem to act randomly, but have a hidden probabilistic regularity.
- Instrument program and deduce from sampling which paths have most chance to bring the heap in a desired exploitable state.

Markov transition system



The transition system models the set of all possible random walks.

Markov transition system

Previous slide explained:

- We computed the probability of reaching every heap states in a maximum of two heap interactions (malloc, free, etc)
- Probability of reaching S4 is:
  P(S4) = P(S4|S2) * P(S2|S1) = 0.6 * 0.9 = 0.54 (54%)
- Probability of reaching S5 is:
  P(S5) = P(S5|S2) * P(S2|S1) + P(S5|S3) * P(S3|S1)
  = 0.9*0.4 + 0.95*0.1 = 0.455 (45.5%)
- Probability of reaching S6 is:
  P(S6) = P(S6|S3) * P(S3 | S1) = 0.1 * 0.05 = 0.005 (0.5%)

Assuming S5 and S6 are the only two desired exploitable states, the most exploitable random walk ends in S5.

Markov Exploit Food for thoughts

- Paths exploration strategy can be static or dynamic (planned, or constructed on the fly)
- If one creates an accurate heap manager specification, heap state measurement could be static, but this is a very hard and allocator-dependent task.
- Most likely, one needs to execute program and instrument debugger to measure heap state when heap operations are performed.
- After monitoring, one can construct the Markov transition system based on sampled program paths. More samples means heap model is more accurate.

Markov Exploit Food for thoughts (2)

- Determine list of possible heap interactions (malloc, free, etc) sequences in a given program. A single unique sequence may be represented by multiple random walks due to non-deterministic heap manager behavior.
- Determine sequence maximizing probability of reaching desired heap state in a minimum amount of steps. A SMT solver can be used to craft corresponding input based on encountered path predicates.
- A range of Markov models can be used to facilitate encoding of heap structure into a probabilistic transition system (Markov chain, Markov network, etc)

Challenge problems

Hilbert’s program

- In 1900, German mathematician David Hilbert formulates a list of 23 hard problems touching the foundations of mathematics. Five of these problems remain unsolved today.

  http://en.wikipedia.org/wiki/Hilbert's_program

A Program for Automated Exploitation

- Inspired by David Hilbert and many ones after him, we define a list of problems whose solutions pave the way for years to come in the realm of automated low-level software analysis.
- The Grand Challenge consists of a set of 11 problems in the area of vulnerability discovery and exploitation that vary in scope and applicability.
- Most problems relate to discovering and combining exploit primitives to achieve elevation of privilege.

Exploit challenges are not new

- Gerardo Richarte’s insecure programming (from 10 years ago!) constitutes great training for manual exploit writing:

  http://community.coresecurity.com/~gera/InsecureProgramming/

- Many of the “Capture the Flag” events are, in essence, manual exploit challenges.
- In this challenge, we expect exploits to be generated automatically instead of written manually.

Nature of Grand Challenge problems

- Exploit Specification problem (A, H)
- Input generation problems (B, C, D, E)
- Exploit Primitive composition problem (F)
- Environment determination (I, J, K)
- State space representation (G)

Not all problems need to be resolved for a given target as different problems cover different exploit scenarios.

Grand Challenge Evaluation

Two main problems of Automated Exploitation are **Vulnerability Discovery** and **Vulnerability Exploitation.** Solutions to challenge problems must be evaluated on their varying degree of:

- Soundness (Precision and Signal/Noise ratio)
- Expressivity (Applicable domain and Configurability)
- Scalability (Automation and Performance)
- Completeness (Coverage)
- Resilience (to Environment and Exploit Mitigations)

Exploit specification

Problem A: Given a program P, determine the set of assertions S for which satisfying any a in S is equivalent to corrupting the program.

In other words,

what is the program P **anti-specification ?**

Problem A code

```
F(int x, int y)
{
   int loc[4];
   int idx = G(x, y);
   if (idx > 4)
     return -1;
   assert(idx >= sizeof(*loc)); // do infer assertion
   loc[idx] = 0x00;
}
```

Pre/post-conditions inference

Problem B: Given a program function and an assertion in the function, determine the necessary and sufficient pre/post conditions such as the assertion is true if and only if the pre/post conditions is true.

This is equivalent to the input generation problem (we start with loop-free programs).

Note: May need to walk over call graph to resolve problem transitively from entry point to assertion.

Problem B code

PRECOND (?)

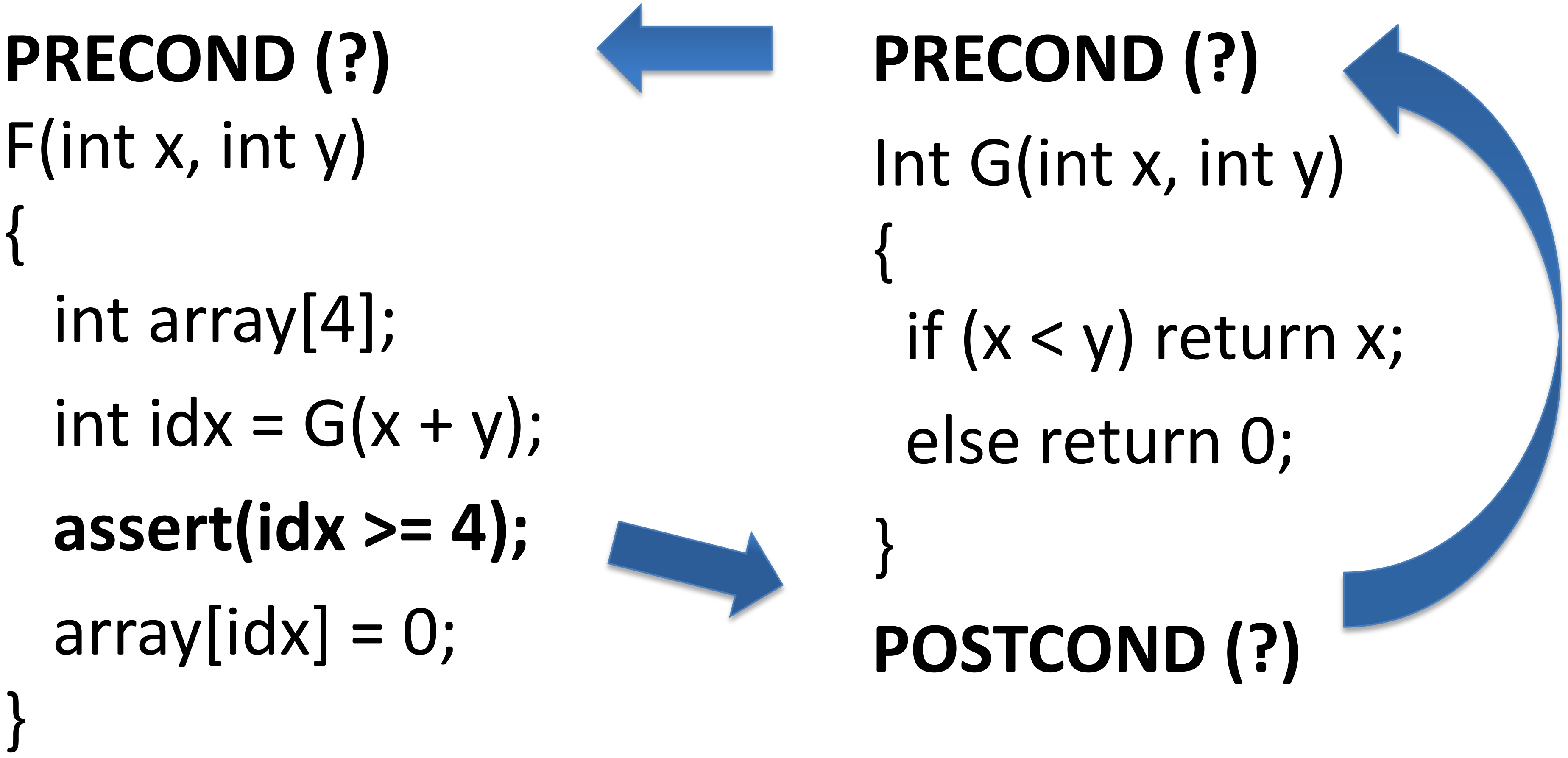
```
F(int x, int y)
{
  int array[4];
  int idx = G(x + y);
  assert(idx >= 4);
  array[idx] = 0;
}
```

PRECOND (?)

```
Int G(int x, int y)
{
  if (x < y) return x;
  else return 0;
}
```

POSTCOND (?)

Problem B code



PRECOND (?)

```
F(int x, int y)
{
  int array[4];
  int idx = G(x + y);
  assert(idx >= 4);
  array[idx] = 0;
}
```

PRECOND (?)

```
Int G(int x, int y)
{
  if (x < y) return x;
  else return 0;
}
```

POSTCOND (?)

Loop assertion inference

Problem C: Given a program loop and an assertion A1 within or at the loop exit-node, determine loop-assertion A2 such as A1 is true if and only if A2 is true.

Note: A loop invariant is an assertion that must be verified at every iteration of the loop. Given that we work on a program anti-specification, the desired exploit loop assertion may not be necessarily a loop invariant (it could just be true at some iterations).

Problem C code

```
F(char *buf, int bufsz)
{
  int limit = bufsz;
  int idx = 0;
  loop_assertion(?)
  while (i < limit)
  {
     if (buf[i] == ‘{’) limit++;
     else if (buf[i] == ‘}’) limit--;
     i++;
  }
  assert(i >= sizeof(buf));
  buf[i] = 0;
}
```

Exploit input definability

<u>Problem D:</u> Given an initial state I of a program P with functions and loops, exhibit an algorithm converging to a desired sink state.

A desired sink state can be defined as an assertion in the program (more weakly: as a set of chosen variables values).

Problem D code

```
Precondition(?)                    // D = A + B + C
F(int x, int y)
{
   int loc[4];
   int idx = G(x, y);
   if (idx > 4)
      return -1;
   while (x < y) idx++;
   assert(idx >= sizeof(loc));   // how to reach this?
   loc[idx] = 0x00;
}
```

Exploit derivability

Problem E: Given a concrete program input and associated program crash/log, find the longest crash trace prefix from which the desired exploitable program state can be reached.

The available program crash/log can be:

(1) Full (unlimited access to all values ever)

(2) Partial (only active values are tracked)

(3) Control-only (ex: a stack or instructions trace)

Problem E code

```
/* Crash possibly generated by fuzz testing */
F(int x, int y)
{
  int loc[4];
  if ((x + y) > sizeof(loc))
     return (loc[x]);                      // program crash here
  else if ((x + y) <= sizeof(loc)) {
     x = G(x, y);
     loc[x] = 0;                           // how to reach here?
     return (0);
   }
}
Int G(int x, int y) { while (x < y) x++; return (x); }
```

Multi-interaction exploit

Problem F: Given a program initial state I, a desired program state U unreachable from I within any single program interaction R, determine all intermediate states T such as multiple interactions Ri can be composed to reach U as in : R1(I,T) + R2(T,U)

Transitive decomposition: determine minimum number of interactions to reach U from I .

Problem F code

```
Char *glob;
F(int x, int y)                     // Ex: F and G are syscalls
{
  glob = malloc(x + y);             // integer overflow
}
G()
{
  glob[x] = 0;                      // array OOB access
}
```

How to construct Trigger() = { F(); G(); } ?

Minimal concurrent exploit

Problem G: Given a program P, a desired exploit state S, and a thread count C, find the minimal state space representation to reach S in some execution of P **while retaining ability to generate corresponding concrete input.**

Note 1: *Partial Order Reduction* is a generic framework that can help control state space explosion.

Note 2: Minimal state space representation is dependent on desired sink state (as in *Abstract Interpretation*).

Example of research in this area: “Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns” (Jurczyk / Coldwind, 2013)

Problem G code

```
/* Example of basic TOC/TOU vulnerability */
/* ptr holds a valid non-volatile pointer */
F(unsigned int *ptr)
{
  if (*ptr > 0x10) return;
  global->ptr = malloc(*ptr + 1);
  if (global->ptr == NULL) return;
  global->ptr[*ptr] = 0x00;      // double-fetch!
}
```

If *ptr* is "modified under" by another thread, the second array access can go OOB.

Privilege Separation Inference

<u>Problem H:</u> Given a program P, determine code privilege partitioning. For each partition, determine entry points.

(1) Determine variables guarding privilege level (PL)

(2) Partition functions so that all elements of a given partition share the same PL. If static partitioning does not exist, determine parameters of dynamic partitioning.

Partitioning can determine multi-stage exploits paths:

- Remote → Local → Kernel
- Remote → Sandboxed → Unsandboxed
- Remote → Non-authenticated → Authenticated

Problem H code

```
bool authenticated = false;
Int F()
{
  authenticated = check_creds();
  // execute at authenticated level
  if (authenticated)
  {
    bool res = serve_client();
    if (!res) return (send_error(E_FUNC));
    return (0);
  }
 // execute at non-authenticated level
 return (send_error(E_AUTH));
}
```

Note: send_error() can execute at multiple privilege levels.

Heap likelihood inference

Problem I: Given a program P using a non-deterministic heap allocator, determine most exploitable random walk(s) for P to reach "aligned" exploitable heap state.

1. Assume existence of heap corruption C in P
2. Identify set S of exploitable heap states w.r.t. C
3. Minimize steps to reach any element of S

See previous Markov exploit description. This problem is particularly relevant in presence of heap randomization.

Problem I code

```
Struct s1 { int *ptr; } *s1a = NULL, *s1b = NULL, *s1c = NULL;
Struct s2 { int authenticated; } *s2 = NULL;

F() {
  s1a = (struct s1*) malloc(sizeof(struct s1));
  s1b = (struct s1*) malloc(sizeof(struct s1));
  s1c = (struct s1*) malloc(sizeof(struct s1));
}
G() { s2 = (struct s2*) malloc(sizeof(struct s2)); }
H() { free(s1b); }
I() { memset(s1a, 0x01, 32); }
J() { if (s1 && s1->authenticated) puts(“you win”); } // Print this
K() { if (s2 && s2->ptr) *(s2->ptr) = 0x42; }           // Avoid crash here
```

Iff allocator reuses s1b’s memory to allocate s2 with max probability:

Automate best walk = { F(); H(); G(); I(); J(); }

Generalized program timing attack

Problem J: Define the necessary and sufficient execution time analysis conditions to infer value, size, or location of:

(1) A program control structure
- Return address, Function Pointer, Exception Handler, etc.

(2) A program data structure
- Heap chunk, Stack Frame, Global variable, etc.

(3) A program code fragment
- Instruction, Function, Method, etc.

In other words, automate program time inference to defeat address space randomization.

Problem J examples

The problem is stated in very generic terms on purpose.

Resolution depends on target-specific implementation.

For two great starting point on timing inference, see:

Cryptographic timing attacks on DH, RSA, DSS and other systems
(Paul C. Kocher, 1996)
http://www.cryptography.com/public/pdf/TimingAttacks.pdf

Program timing attacks on Firefox hash tables
(Paul @pa_kt, 2012)
http://gdtr.wordpress.com/2012/08/07/leaking-information-with-timing-attacks-on-hashtables-part-1/

Indirect information disclosures

Problem K: Define the necessary and sufficient conditions to infer the value or address of a variable without a direct read primitive, such as:

(1) Data reuse attacks
(example: partial pointer overrides)

(2) Pointer value prediction attacks
(example: pointer inference)

Problem K examples

Resolution of Problem K depends on target-specific implementation.

Prior art on Indirect information disclosures includes:

Flash Pointer Inference (Blazakis, 2010)
http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf

Garbage Collection marking attack (Blazakis, 2013)
http://www.trapbit.com/talks/Summerc0n2013-GCWoah.pdf

Conclusion

- We decomposed the problem of Automated Exploit Generation in a set of challenges with clear intermediate assumptions.
- Resolving one such sub-problem is a step towards automated end-to-end solutions of larger and larger sub-classes of exploits.
- Even though Automated Exploitation is an undecidable problem, observing that most vulnerabilities are shallow allows the problem to be approached.

Questions / Discussion

- Thanks for attending H2HC's 10th anniversary

- Questions and feedback welcomed by email