Automatic Exploit Generation
an Odyssey

Sophia D’Antoine
Hack.lu 2015
Introduction

Programs have become increasingly difficult to exploit

- larger, changing surface area
- mitigations
- more bytes to siphon through
Introduction

Reaction:
people get **smarter** and **tools** get better

- government research
- pentesters
- CTF!
CTF & Wargames

Scoreboard

<table>
<thead>
<tr>
<th>Place</th>
<th>Team</th>
<th>Bracket</th>
<th>Country</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1064CBread</td>
<td>Undergraduate</td>
<td>United States</td>
<td>6860</td>
</tr>
<tr>
<td>2</td>
<td>DCIETS</td>
<td>Undergraduate Stacked</td>
<td>Canada</td>
<td>6860</td>
</tr>
<tr>
<td>3</td>
<td>RPISEC</td>
<td>Undergraduate Stacked</td>
<td>United States</td>
<td>6860</td>
</tr>
<tr>
<td>4</td>
<td>PPP1</td>
<td>Undergraduate</td>
<td>United States</td>
<td>5810</td>
</tr>
<tr>
<td>5</td>
<td>Shellphish Nigiri</td>
<td>Undergraduate Stacked</td>
<td>United States</td>
<td>5810</td>
</tr>
</tbody>
</table>

10/22/2015
The Past

Manual labor

- **static** analysis
- **dynamic** analysis
Dynamic Analysis

Definition:
- **Running** it (concrete execution)
- Collecting/ observing environment changes

Popular Uses:
- dump VM memory & grep
- record/ replay & manual analysis
- gdb (debuggers) & run
Dynamic Analysis

Common tools:
- gdb, windbg, cdb
- python brute force (blind fuzzing)
Example: Dynamic Analysis
Automated Exploitation
Agenda

1. Intro
2. Automating Exploitation
   a. what, how?
   b. the target
3. Program Analysis
   a. background
   b. types we care about
   c. how this helps with AEG
4. Application
   a. tools
   b. demo
5. Conclusion
Some Background

What is Automated Exploitation?
The ability to generate a successful computer attack with reduced or entirely without human interaction.

- Focus on **discovery** and **combination** of write and read primitives

- Existing AE work focused on Restricted Models:
  - Sean Heelan’s “Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities”
  - David Brumley (@ Carnegie Mellon) et al. (AEG, MAYHEM, etc)
  - *Cyber Grand Challenge!* (CGC)
Automating Exploitation

Break up AEG into 2 parts:

- Generating **input** to get to vulnerability
- Generating “**payload**” to profit from vulnerability

- Both are hard
- Work being done in both areas
- Focus today on first problem
Automating Exploitation

PWNABLE.KR
AEG - pwnable.kr

Program Operations

Get random binary, pwn it in 10 seconds.

1) Takes input at argv[1]
2) Does some decode & operations on it
3) Calls sequence of 16 functions
4) Each function checks 3 characters of input sequentially
5) If you pass them all, you get to the exploitable memcpy!

Automated Exploit Generation

1) Generate input to get to vulnerability
2) Generate payload to exploit and get shell
AEG - pwnable.kr

Shell we play a game?

input argv[1]

fail ...

3 checks

fail ...

... 15 more functions ...

memcpy
How can AEG solve for this path in the CFG?
Software Program Analysis!
Agenda

1. Intro
2. Automating Exploitation
   a. what, how?
   b. the target
3. Program Analysis
   a. background
   b. types we care about
   c. how this helps with AEG
4. Application
   a. tools
   b. demo
5. Conclusion
What is program analysis

The process of **automatically** analyzing the behavior of applications

- In terms of a **property**:
  - program **correctness**
  - set of paths == expected paths
  - program **optimization**

- minimum expense => expected paths
How This Helps with AEG

Analysis helps us **hunt** for bugs automatically.

- **Fuzzing/ Instrumenting**
- **Symbolic Execution**
- **Concolic Execution**

**===> Pro move:** **combine analyses**
Types we care about.
Dynamic Binary Instrumentation

Definition:
- ‘Hijacked’ environment, binaries, or source
- Monitor specific system artifacts
- Attempts at complete (concrete) execution

Popular Uses:
- Force program states
- Gather and report observations at runtime
- Types of hooking: source & binary
Example: DBI

```
$pin -t inscount0.so -- binary
```

[BINARY LEVEL]
- Inject increment after each instruction

[STILL BRUTE FORCE]
- Return total instructions for fuzzed input
- Only true for that 1 executed path
  (the possible CFG space may be very large)
Example: DBI

```assembly
sub $0xff, %edx
sub %0xff, %edx
jle ncount++
mov %0x1, %edi
add $0x10, %eax
jle
icount++
mov $0x1, %edi
icount++
add $0x10, %eax
```

Program Analysis to Find Vulnerabilities
Symbolic Execution

Definition:
- Generate 1 sym path for a set of paths (could still be extremely expensive)
- Satisfies path conditions
- Composed of some concrete values

Popular Uses:
- Determine program state at particular basic block
- Create ‘equation’ to feed to SAT/SMT solvers
- Faster than brute forcing all conditions
Example: Symbolic Execution

```c
[INT] a, b, c
[INT] x, y, z = 0;

fun( int a, b, c ){
    if (a) {
        x = -2;
    }

    if (b < 5) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x+y+z!=3)
}
```

Old Method:
Try all inputs until assert

[WARNING] inputs unbounded!
Example: Symbolic Execution

```plaintext
[SYMBOL] a, b, c

[INT] x, y, z = 0;

if (a) {
    x = -2;
}

if (b < 5) {
    if (!a && c) {
        y = 1;
    }
    z = 2;
}

assert(x+y+z!=3)
```
Concolic Execution

Definition:
- Dynamic symbolic execution
- Instrumentation of symbolic execution as it runs
- One path at a time to maintain concrete state underneath symbolic variables

Popular Uses:
- Concretization (replace symbols with values to satisfy path condition)
- Handle system calls & library loading
- Cases which SMT can’t solve
Example: Concolic Execution

```c
[INT] a, b, c
[INT] x, y, z = 0;

fun( int a, b, c ){
    if (a) {
        x = -2;
    }
    if (b < 5) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x+y+z!=3)
}

fun( 0, 3, 1);

Old Method:
Try all inputs until assert

[WARNING] inputs unbounded!
```
Example: Concolic Execution

```
[INT & SYMBOL] a, b, c

[INT] x, y, z = 0;

if (a) {
    x = -2;
}

if (b < 5) {
    if (!a && c) {
        y = 1;
    }
    z = 2;
}

assert(x+y+z!=3)
```

**STEPS**

**ONE**
concrete execution of function

**TWO**
while building symbolic path model

**THREE**
constraints on input are modeled

**FOUR**
models used to generate concrete input
Creating a Feedback Loop

In practice using the results of different analyses finds bugs quicker.

Example Pairing:
- Concrete execution
- Fuzz input
- Symbolic/Concolic execution
- Examine results
- Craft new input
Agenda

1. **Intro**
2. **Automating Exploitation**
   a. what, how?
   b. the target
3. **Program Analysis**
   a. background
   b. types we care about
   c. how this helps with AEG
4. **Application**
   a. tools
   b. demo
5. **Conclusion**
Dynamic Binary Instrumentation

Common tools:
- PIN Tool
- Valgrind (before/during runtime)
- DynamoRIO
- Qemu
Example: Flare-on Challenge 9


- Pintool instruction count
- More instructions == Closer to correct input

Input:
AAAAA...
Symbolic Execution

Common tools:
- KLEE (runs on LLVM bc)
- SAGE (MS internal tool)

feed it to z3 to solve
Concolic Execution

Common tools:
- Angr
- Pysymemu
- Triton
AEG Demo: Assumptions

[ Assumptions ]
- Space of potential vulnerabilities too large
- Need to write tools to hunt for subset
  - Target memory corrupt (memcpy)
- ROP from there...

[ Dynamically Acquire ]
- Path to target
- Solve for constraints
- Addresses of gadgets for ROP

[ Statically (Pre) Acquired ]
- Semantics of target & gadgets
Using the structure of the binary:

- **Dominator Tree**
  - Longest path of CFG is the “winning” path
- **Use-def chain**
  - Each cmp of this path comprises the “constraints”

⇒ "**Flow-sensitive constraint analysis**"

**LLVM:**

- Makes this analysis easier
  - DomTree & Use-def construction
  - **Semantics of cmp and vars** easy to pull out
  - Runs statically over bitcode (lift with Mcsema)
  - Fast
Angr Script

... acquire binary & some conditions ....

```python
b = angr.Project("aeg")
ss = b.factory.blank_state(addr=entry_func)
ss.options.discard("LAZY_SOLVES")
ss.se._solver.timeout=10000
ss.memory.store(argv1_buff, ss.BV("input", 50*8))
pg = b.factory.path_group(ss, immutable=False)
angr.path_group.1.setLevel("DEBUG")
pg.explore(find=vuln_addr[0], avoid=fail_bbs)
argv1_win = pg.found[0].state.se.any_str(pg.found[0].state.memory.load(argv1_buff, 50))
```
Demo
Conclusion: The Future

[ What We are (still) Working With ]

- Binaries
- Source is nice
  - Need to lift bins to IR for LLVM
  - Most concolic exec. tools would need to compile it.

[ Difficulty ]

- Know how to express our targeted vulnerability
- Semantics for UAF, Memory Corruption, etc....
Finding (More) Bugs

**Automatic** program analysis
- translate program (IR)
- define program in-correctness

goal: proving **existence** or **absence** of bugs
Acknowledgements

- Trail of Bits
- RPISEC
References

[Good Course Material]
http://homepage.cs.uiowa.edu/~tinelli/classes/seminar/Cousot.pdf

[Site for Tool Documentation]
https://github.com/angr/angr-doc  
https://github.com/llvm-mirror/llvm

[Other Good Resources]
http://www.grammatech.com/blog/hybrid-concolic-execution-part-1  
Any Questions?

IRC: quend
email: sophia@trailofbits.com

10/22/2015