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:

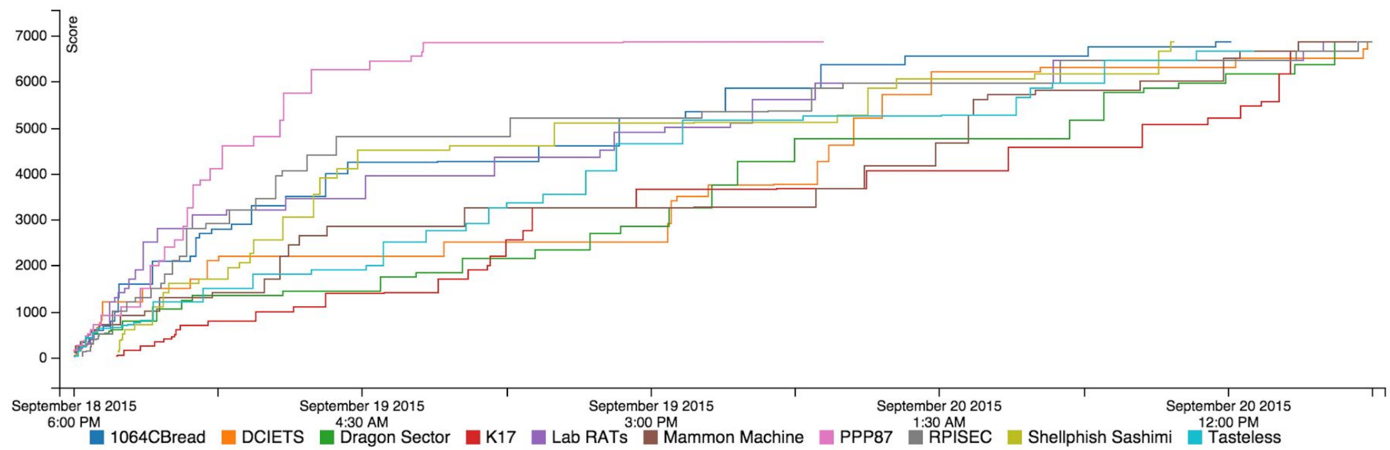
people get smarter and tools get better

- government research
- pentesters
- CTF!

CTF & Wargames

CSAW | Home | Rules | Judges | Teams | Scoreboard | Challenges | Statistics | CSAW.TV | Archives | Login

Scoreboard



All | Qualifying | High School | Industry | Undergraduate | Graduate | Other

Place	Team	Bracket	Country	Score
1	1064CBread	Undergraduate	United States	6860
2	DCIETS	Undergraduate Stacked	Canada	6860
3	RPISEC	Undergraduate Stacked	United States	6860
4	PPP1	Undergraduate	United States	5810
5	Shellphish Nigiri	Undergraduate Stacked	United States	5810

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

step...

step...

step...

step..

step...

step.

step..

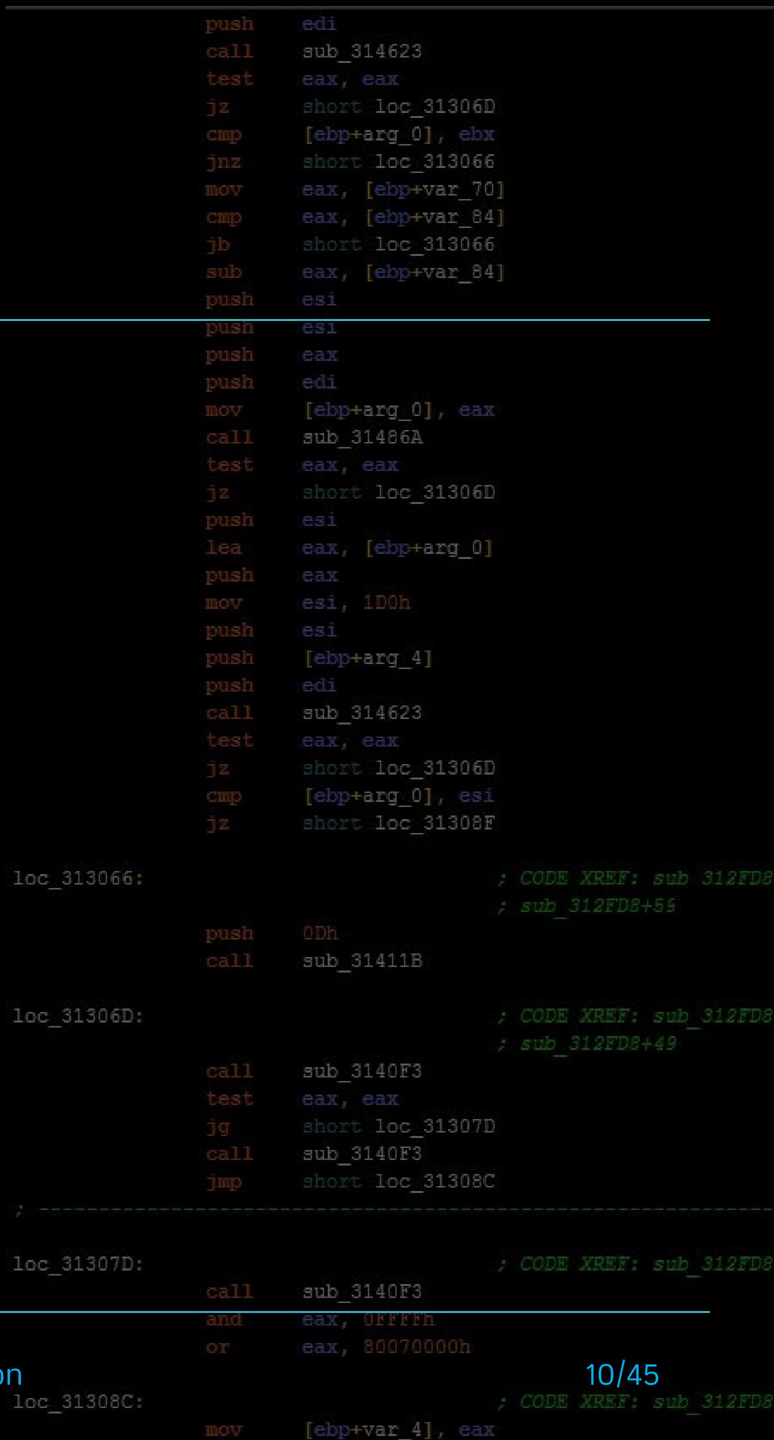
step...

step...

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.

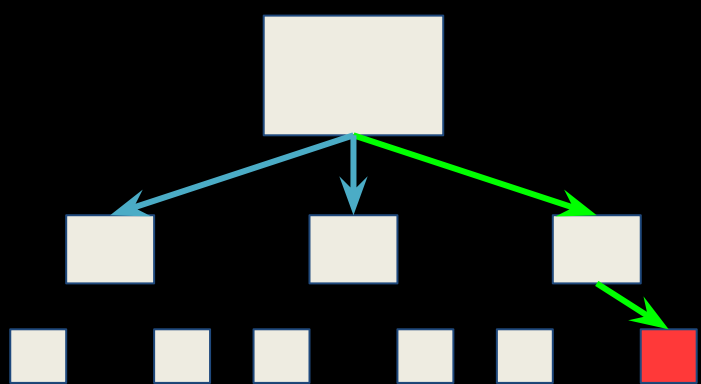
- Focus on **discovery** and **combination of write and read** primitives

- 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

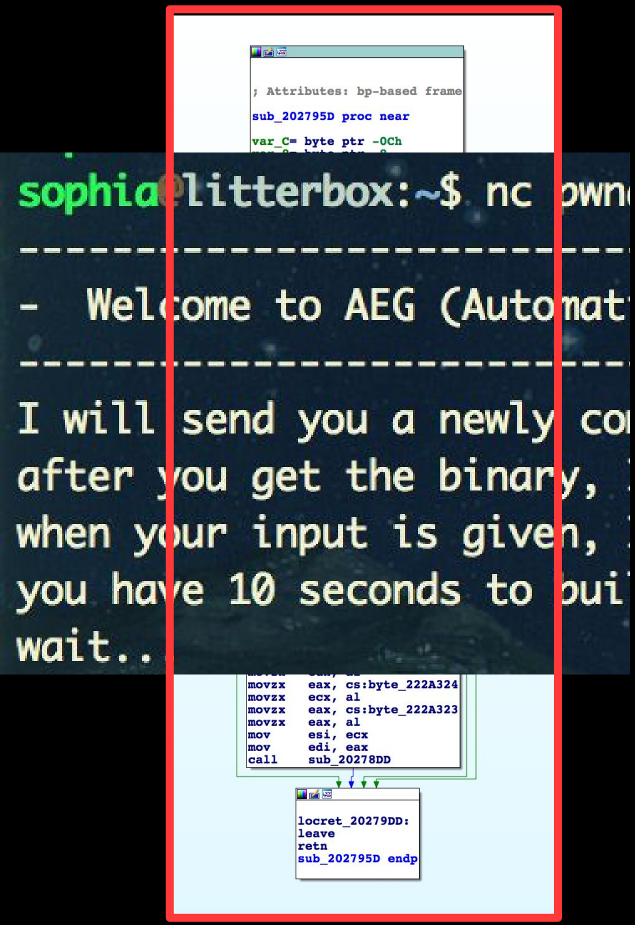


- Both are hard
- Work being done in both areas
- Focus today on first problem

Automating Exploitation

PWNABLE.KR

AEG - pwnable.kr



Program Operations

Get random binary, pwn it in 10 seconds.

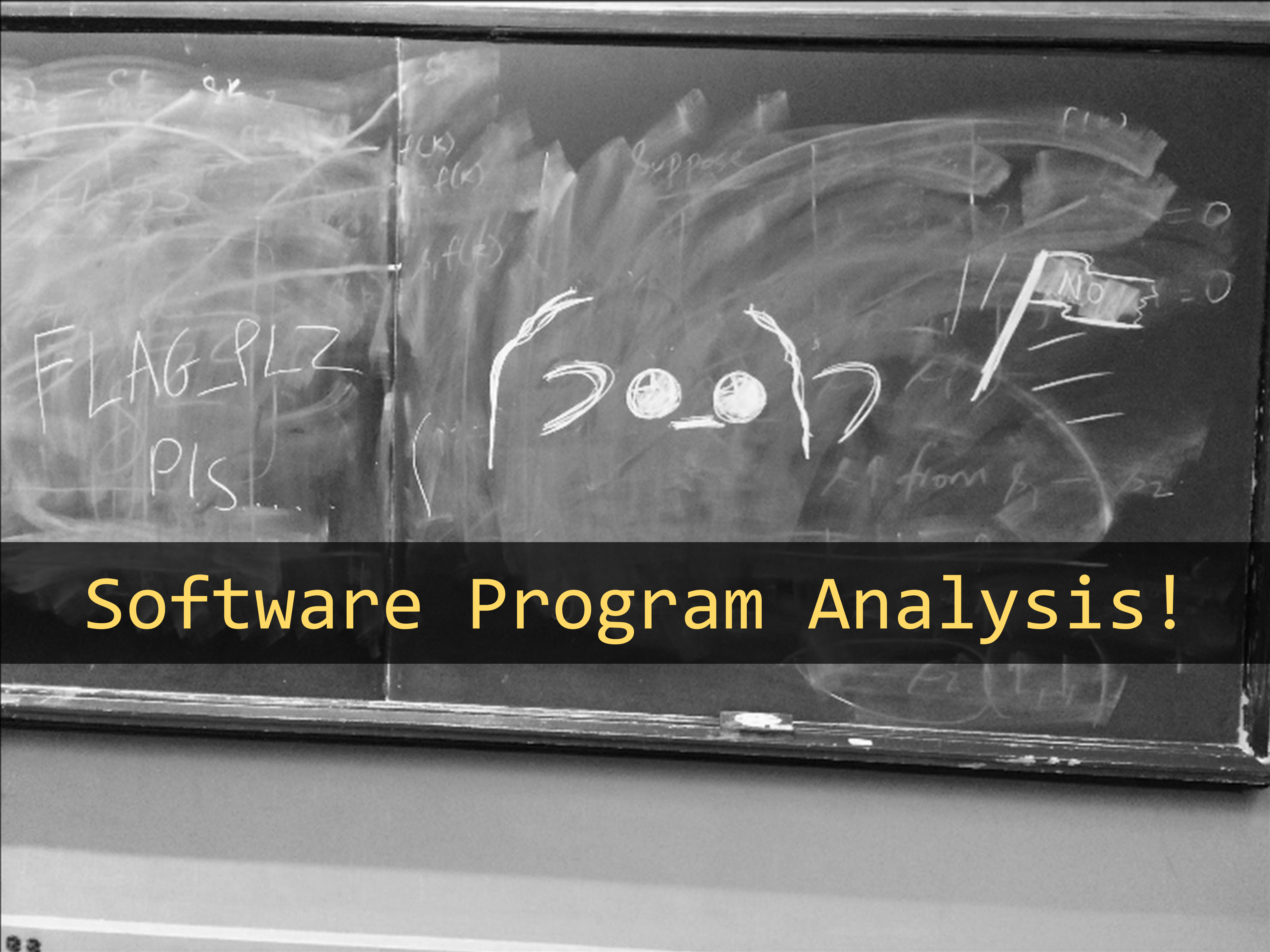
1) Takes input at argv[1]
2) Does some decode & operations on it
3) Calls sequence of 16 functions
4) Each function checks 3 characters of input sequentially
5) If you pass them all, you get to the exploitable memcpy!

Automated Exploit Generation

1) Generate input to get to vulnerability
2) Generate payload to exploit and get shell

AEG - pwnable.kr

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

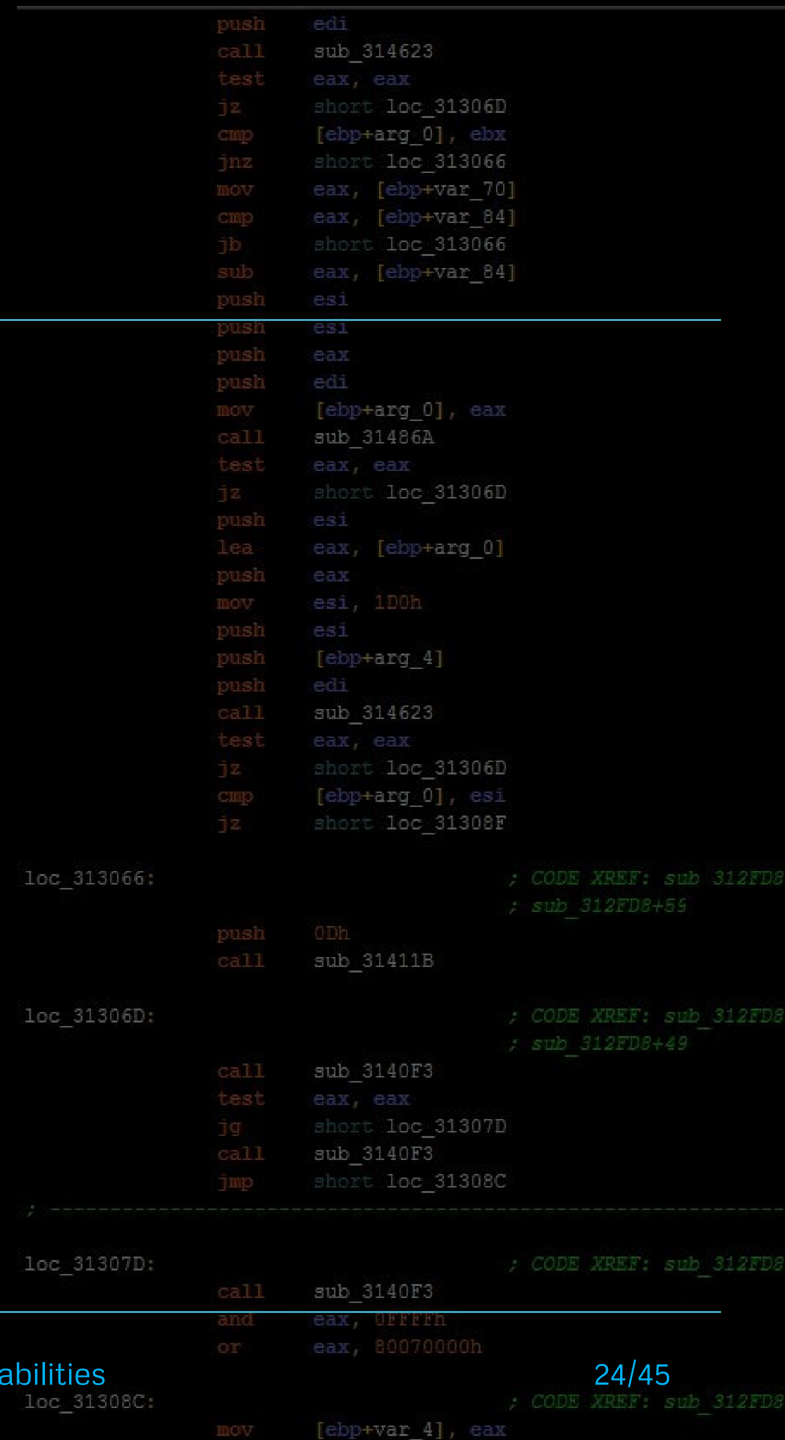
[STILL BRUTE FORCE]

- Return total instructions for fuzzed input
- Only true for that 1 executed path
  (the possible CFG space may be very large)

Example: DBI

```
sub $0xff, %edx
cmp $0xff, %edx
jle
mov $0x1, %edi
add $0x10, %eax
```

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



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

```
[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

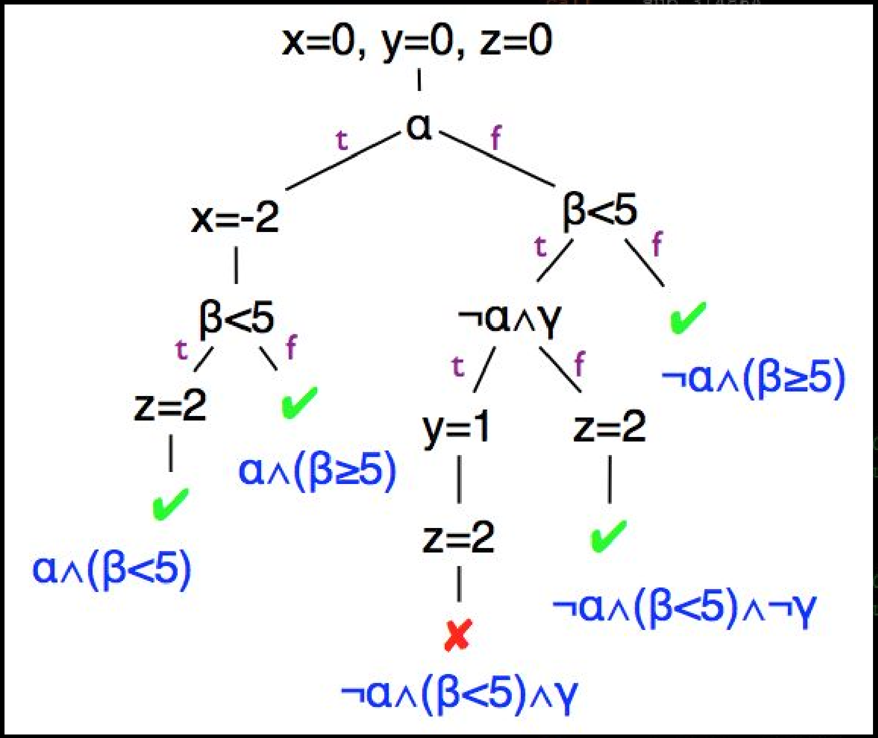
[WARNING] inputs unbounded!

Example: Symbolic Execution

```
[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

```
[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

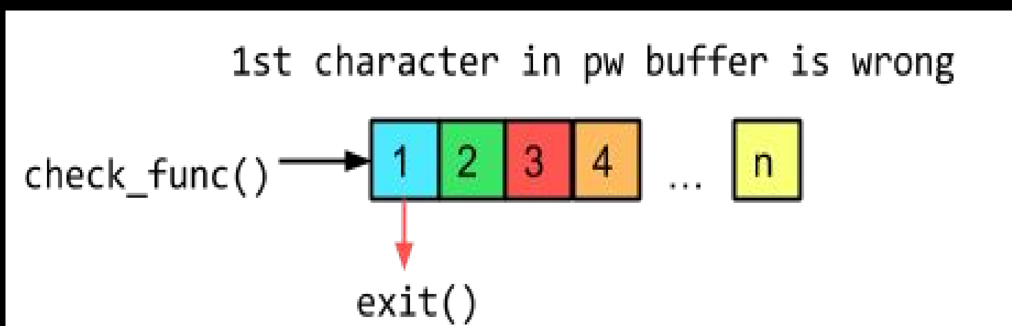
Common tools:

- PIN Tool
- Valgrind (before/during runtime)
- DynamoRIO
- Qemu

Example: Flare-on Challenge 9

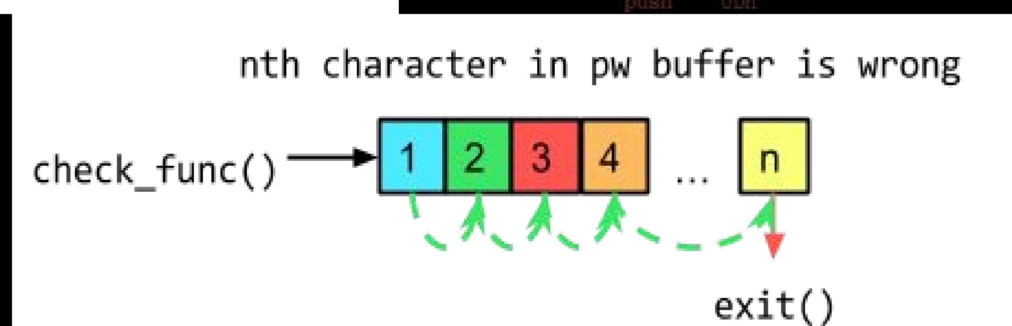
[http://blog.trailofbits.com/2015/09/09/flare-on-reversing-challenges-2015/]

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



Input:
AAAAAAAA...

Input:
FLAGAAAA...



Symbolic Execution

Common tools:

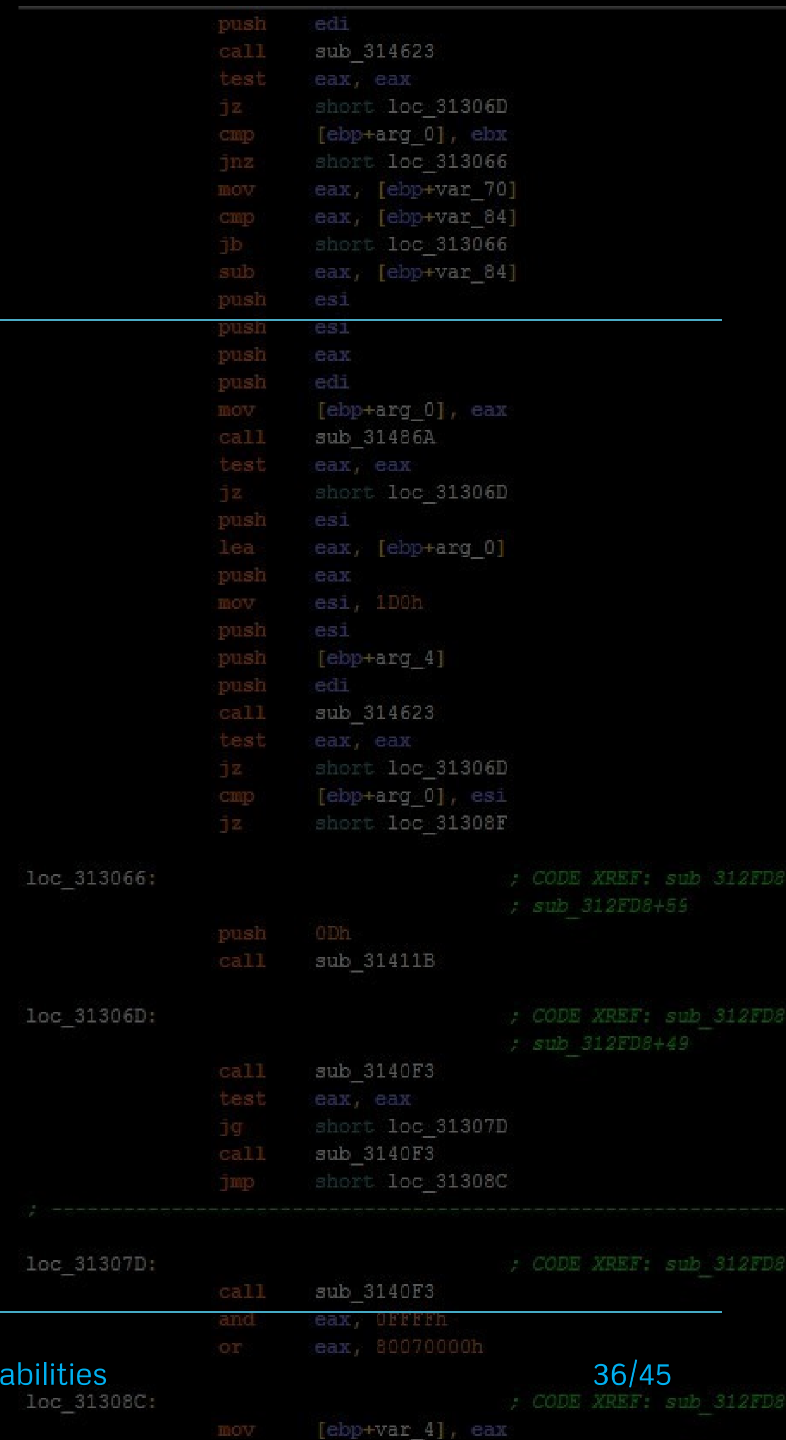
- KLEE (runs on LLVM bc)
- SAGE (MS internal tool)

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

LLVM Pass

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 ....

```
b = angr.Project("aeg")                                  #setup env

ss = b.factory.blank_state(addr=entry_func)
ss.options.discard("LAZY_SOLVES")
ss.se._solver.timeout=10000                   #fake input with no value
ss.memory.store(argv1_buff, ss.BV("input", 50*8))

pg = b.factory.path_group(ss, immutable=False)
angr.path_group.l.setLevel("DEBUG")
pg.explore(find=vuln_addr[0], avoid=fail_bbs)   #target & bad branches, 4 speed

argv1_win = pg.found[0].state.se.any_str(pg.found[0].state.memory.load(argv1_buff, 50))

                           #solved for path to target, dump memory
```

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]

https://www.cs.umd.edu/class/spring2013/cmsc631/lectures/symbolic-exec.pdf
https://www.utdallas.edu/~zxl111930/spring2012/public/lec4.pdf
http://web.mit.edu/16.399/www/lecture_01-intro/Cousot_MIT_2005_Course_01_4-1.pdf
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
http://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf

Any Questions?

IRC: quend
email: sophia@trailofbits.com