An Introduction to Dynamic Analysis for R.E.

Alan Cao

Objectives

- Program Analysis Review
- Tools of the Trade: Debuggers
- Fuzzing
- Symbolic Execution
- Demonstrations
- Other Resources
- Concluding Thoughts / Questions

Program Analysis Review

- **program/binary analysis**

  “How does this program behave under the hood?”

  - Analyzing and reasoning with *properties* and *behaviors* of a computer program, in order to gain a deeper intrinsic understanding
    - Often times with compiled programs, we are stuck without source!

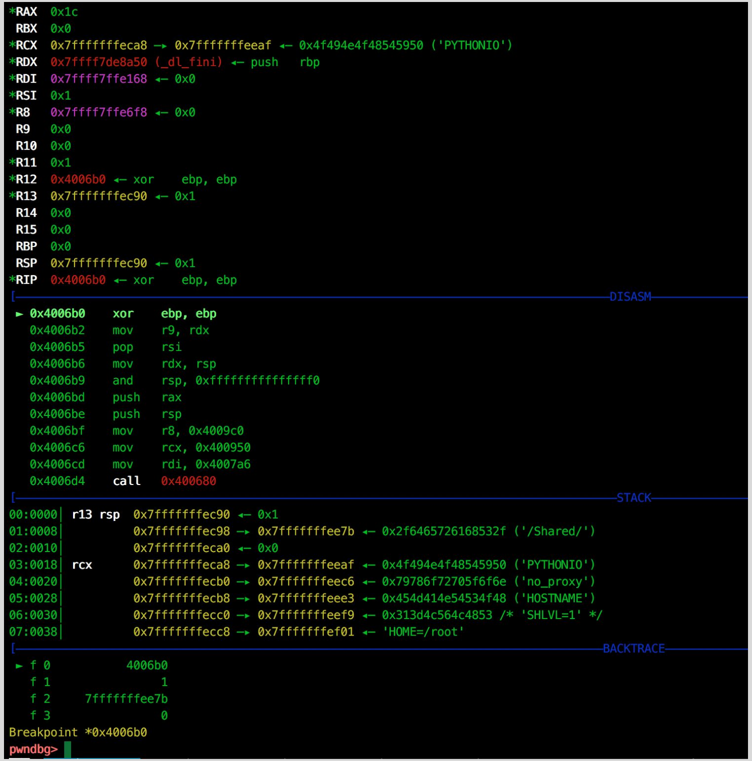
  - Static vs dynamic analysis; *Can someone explain the difference?*
    - **Static** - recovering *properties* and extrapolating information from a program
    - **Dynamic** - gaining an understanding of how a program *behaves* during its runtime

Dynamic Analysis

- Limitations of static analysis *Can someone name a few?*
  - Does not account and anticipate for behaviors occurring during execution
  - Optimal for more manual reverse engineering
- Dynamic Analysis Tools
  - Hook onto a program, observe their behavior when run with an input, and reason about their functionality during execution
- Limitations of dynamic analysis
  - Can be slower than static analysis
  - Won’t provide most insight for environmental side effects (program interacting with operating system facilities)
  - *Need program coverage!
    - We can therefore explore how program behaves *under every single path*

Tools of the Trade: Debuggers

- Debuggers allow you to *introspectively trace* through the execution of a program
  - Understand program crashes
  - Reason about functionality at a machine-level
  - Help write exploits 😈
- Important functionality include:
  - Inspecting memory regions and offsets
  - Pause during runtime by setting *breakpoints*
  - Read register values
  - Trace child processes
  - Provide an interface to build analysis plugins to gain automate meaningful debugging
- Let’s take a look at gdb / pwndbg!



Pwndbg (http://pwndbg.re/)

Using `pwndbg`

- Let’s define our workflow:
    1. Do an initial **static analysis** and recover basic information.
    2. **Trace** through the execution of the program from the entry point.
    3. **Find** our memory offsets of interest.
    4. **Analyze** behavior when EIP is at points of interest
    5. **Craft** an appropriate payload and/or build up your exploit
    6. **PWN**!

Functionality	**Command**	**Usage Example**
Examine Memory	x <addr>	`x/i 0xdeadbeef`
Set Breakpoint	b <symbol> or b <addr>	`b my_function`
Look at memory offsets	vmmap or info proc map	`vmmap`
Step Through Program	si or ni or c	`si`
Attack process	attach <pid>	`attach 5431`
Set register values	set <reg> = value	`set $rip = 0x1000`

Demo

Let’s examine how a license validator works under the hood!

https://github.com/osirislab/Hack-Night/blob/master/Rev/dynamic/license_validator/

Other Notable Debugging Tools

- strace / ltrace
  - Run a program, get every system call (strace) and library call (ltrace) made during its execution
- Valgrind
  - Run a program and check for memory violations
- `exploitable`
  - GDB plugin that can determine possible exploit primitives during runtime
- rr (record-replay)
  - Tool with GDB extensions that allow one to record and replay snapshots of a program being run

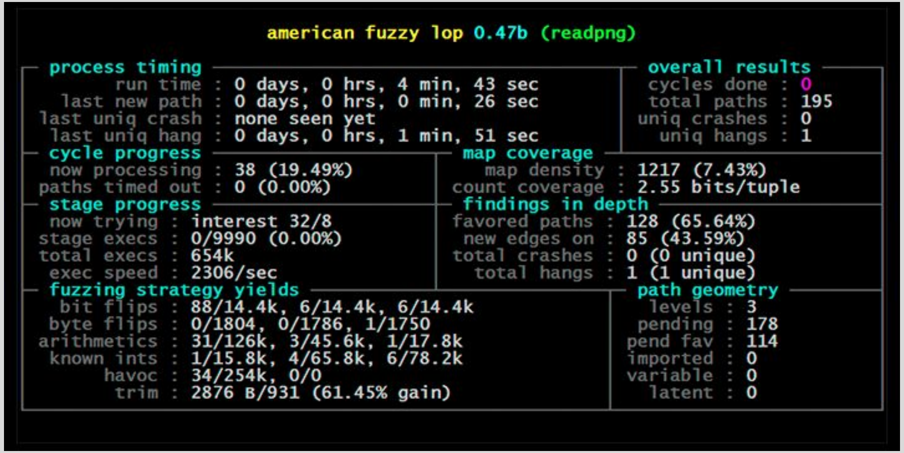
Dynamic Analysis Techniques

- Fuzzing
- Symbolic Execution

Fuzzing - *The "Dumb" Method**

- Throw garbage at a program, and see what crashes we can get
  - (Hopefully they are exploitable!)
- Mutational vs Generational fuzzing
- *Dumb, but *surprisingly effective.*

```
int status = myFunctionality(input);
if (status < 1) {
     // Can we find an input that can
     // hit this condition?
     fail_and_panic();
}
```



American Fuzzy Lop (AFL) - https://lcamtuf.coredump.cx/afl/

Fuzzing

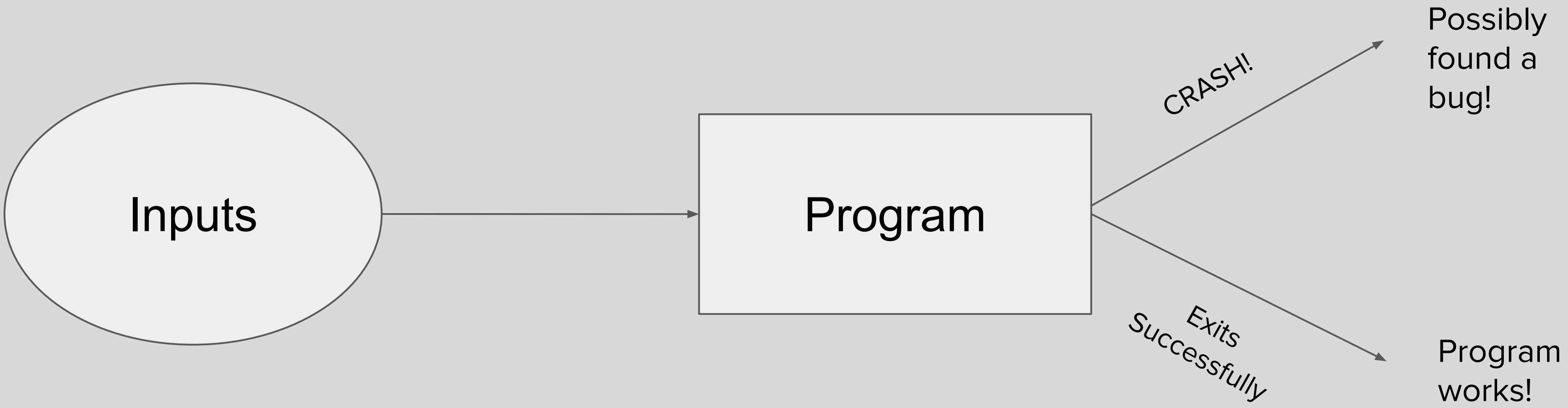
Fuzzing

- Reads and validates your input
- Parses your input
- Perform some functionality, and validate its execution

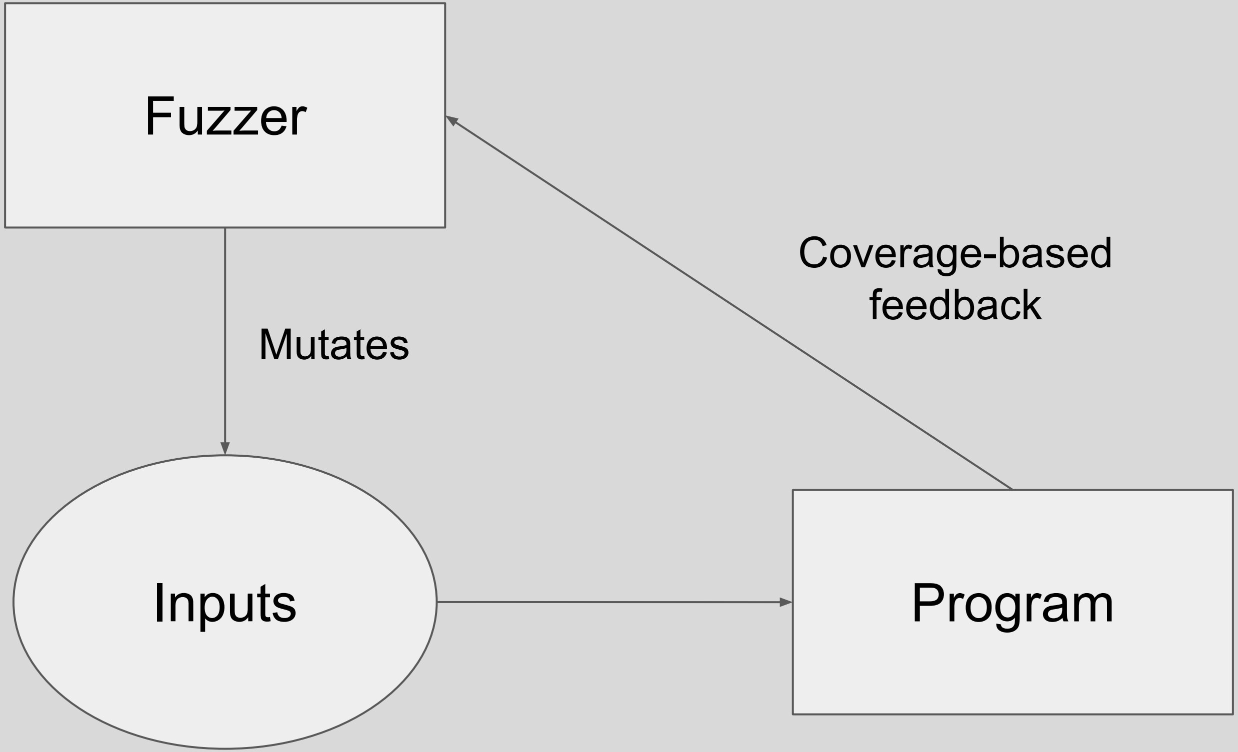
```c
#include <my_json_impl.h>


/*
 * JSON file -> Object -> JSON dump 1
 *
 *  Check if JSON file == JSON dump
 *
 */

int main(int argc, char *argv[])
{
    /* `argv[1]` is a JSON input test we read from */
    char *input = readFromFile(argv[1]);
    size_t size = getFileInputSize(argv[1]);

    /* Parse our input, and an initial validation check */
    json_t *object = json_parse(input, size);
    object.validate();

    /* Dump back as a string */
    char *dump_output = object.dump();
    size_t dump_size = object.dump_size();

    /* Check against our original input! */
    CHECK(input, dump_output);
    exitSuccessfully();
}
```

JSON Example (https://github.com/osirislab/Hack-Night/blob/master/Rev/dynamic/fuzz_example.c)

Fuzzing


Inputs
Program
CRASH!
Possibly found a bug!
Exits Successfully
Program works!

Fuzzing



```c
#include <my_json_impl.h>


/*
 * JSON file -> Object -> JSON dump 1
 *
 *  Check if JSON file == JSON dump
 *
 */

int main(int argc, char *argv[])
{
    /* `argv[1]` is a JSON input test we read from */
    char *input = readFromFile(argv[1]);
    size_t size = getFileInputSize(argv[1]);

    /* Parse our input, and an initial validation check */
    json_t *object = json_parse(input, size);
    object.validate();

    /* Dump back as a string */
    char *dump_output = object.dump();
    size_t dump_size = object.dump_size();

    /* Check against our original input! */
    CHECK(input, dump_output);
    exitSuccessfully();
}
```

- **What are testing?** Serialization and deserialization for a parser library
    - We want to find inputs that *make our library crash.*
    - Crashes may mean *edge cases* that the library did not account for.

- **How are we testing?** We check to see if the library properly serializes an input back to the original
    - The code demonstrates *good coverage* of the functionality, especially as we validate along the way

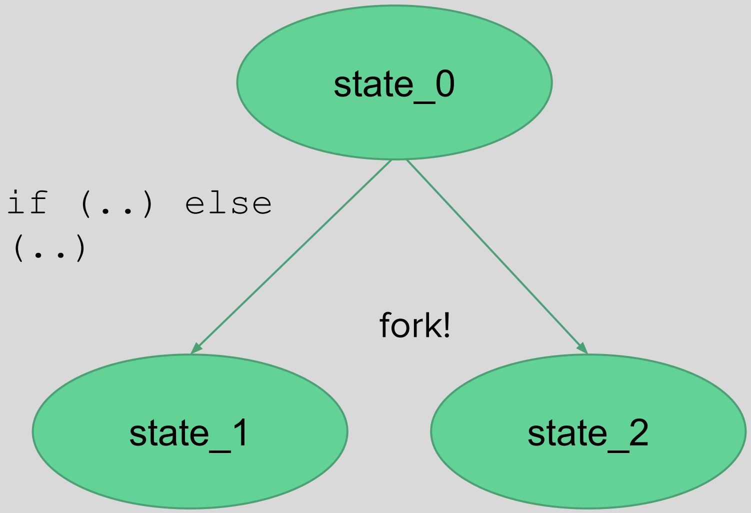
Demo

Finding and Exploiting CVE-2014-0160

https://github.com/osirislab/Hack-Night/blob/master/Rev/dynamic/heartbleed_fuzz/

Symbolic Execution - *The "Smarter" Method*

- Can we automatically generate all possible input test cases that can reach this point of program execution?
- Analyzing a program under a *symbolic model / representation* to logically reason about execution
  - We represent conditional forks as path constraints using a *logical representation* called SMT (Satisfiability Modulo Theorem)
  - Find solutions and generating interesting inputs == solving path constraints



```c
void foo(int z)
{
    fail("%d is less than 10! Bad!", z);
}


void bar(int z)
{
    doSomething(z);
    exitSuccessfully();
}


int main(int argc, char *argv[])
{
    int x = int(argv[0]);
    int y = int(argv[1]);

    int z = x + y
    if (x < 5 && y < 5)
        foo(z)
    else
        bar(z)
}
```

```c
void foo(int z)
{
    fail("%d is less than 10! Bad!", z);
}


void bar(int z)
{
    doSomething(z);
    exitSuccessfully();
}


int main(int argc, char *argv[])
{
    int x = int(argv[0]);
    int y = int(argv[1]);

    int z = x + y
    if (x < 5 && y < 5)
        foo(z)
    else
        bar(z)
}
```

Symbolic Representation

$\pi := T$

$x := \alpha$

$y := \beta$

```c
void foo(int z)
{
    fail("%d is less than 10! Bad!", z);
}


void bar(int z)
{
    doSomething(z);
    exitSuccessfully();
}


int main(int argc, char *argv[])
{
    int x = int(argv[0]);
    int y = int(argv[1]);

    int z = x + y
    if (x < 5 && y < 5)
        foo(z)
    else
        bar(z)
}
```

Symbolic Representation

$\pi := T$

$x := \alpha$

$y := \beta$

$\pi := T$

$z := x + y$

$z := \alpha + \beta$

```c
void foo(int z)
{
    fail("%d is less than 10! Bad!", z);
}


void bar(int z)
{
    doSomething(z);
    exitSuccessfully();
}


int main(int argc, char *argv[])
{
    int x = int(argv[0]);
    int y = int(argv[1]);

    int z = x + y
    if (x < 5 && y < 5)
        foo(z)
    else
        bar(z)
}
```

Symbolic Representation

$\pi := T$
$x := \alpha$
$y := \beta$

$\pi := T$
$z := x + y$
$z := \alpha + \beta$

$\pi := \alpha < 5 \wedge \beta < 5$

Fail branch!

$\pi := \alpha \geq 5 \wedge \beta \geq 5$

Success branch!

```c
void foo(int z)
{
    fail("%d is less than 10! Bad!", z);
}


void bar(int z)
{
    doSomething(z);
    exitSuccessfully();
}


int main(int argc, char *argv[])
{
    int x = int(argv[0]);
    int y = int(argv[1]);

    int z = x + y
    if (x < 5 && y < 5)
        foo(z)
    else
        bar(z)
}
```

Symbolic Representation

$\pi := T$
$x := \alpha$
$y := \beta$

$\pi := T$
$z := x + y$
$z := \alpha + \beta$

$\pi := \alpha < 5 \land \beta < 5$

Fail branch!

$\pi := \alpha \geq 5 \land \beta \geq 5$

Success branch!

Symbolic Executor Outputs:

- Success Branch: x = 6, y= 5
- Fail Branch: x = 0, y = 0

Symbolic Execution

- Current standard tooling:
  - Angr
  - Manticore
  - KLEE
- Still being heavily researched and developed
  - Slow
  - Path explosion problem!
- Why do we care if fuzzing can find a lot of bugs?
  - Extract fine-grained test cases from complexity
  - SE can help provide *verification*- useful for *mission-critical* systems

Manticore
(https://github.com/trailofbits/manticore)

Angr (https://angr.io)

Demo

Let’s go back and break our license key validator!

https://github.com/osirislab/Hack-Night/blob/master/Rev/dynamic/license_validator

Other Cool Tools and Platforms

- ~~radare2~~
- Microsoft Security Risk Detection (SAGE)
  - https://www.microsoft.com/en-us/security-risk-detection/
- BAP (CMU's Binary Analysis Platform)
  - https://github.com/BinaryAnalysisPlatform/bap
- PANDA (NYU/MIT/NU's whole-system malware analysis sandbox)
  - https://github.com/panda-re/panda
- Cyber Reasoning Systems (CRSes)
  - Mechanical Phish - https://github.com/mechaphish

Other Resources

- awesome-dynamic-analysis
  - https://github.com/analysis-tools-dev/dynamic-analysis
- /r/ReverseEngineering
  - https://www.reddit.com/r/ReverseEngineering/
- Google's work in fuzzing
  - https://github.com/google/fuzzing
- Andriesse, Dennis. *Practical Binary Analysis*
  - https://nostarch.com/binaryanalysis
- Related Concentration: Malware Analysis

Closing Thoughts

- Other dynamic analysis techniques to explore:
  - Dynamic taint analysis (DTA)
  - Program slicing
  - Dynamic binary instrumentation (DBI)
  - … and so on!
- Use in tandem with static analysis tools and plugins for effective analyses!
- Program analysis R&D is valuable to industry!

Questions?