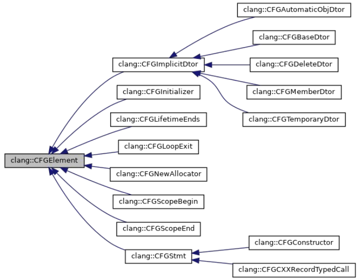
# Symbolic Analysis & Code Emulation

Micah Flack

# Analysis Methods



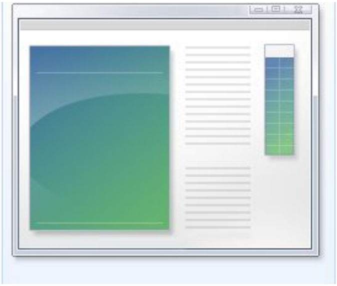

Typically the two main methods:

- Static Analysis, or...
  - Statically gathered indicators w/o runtime of sample
- Dynamic Analysis
  - Indicators gathered through runtime of sample

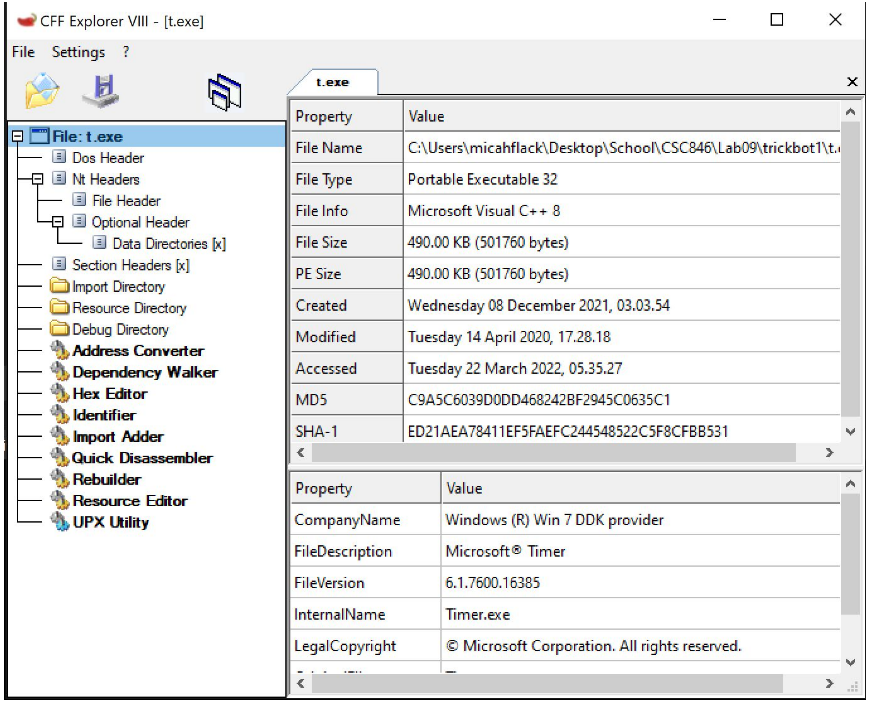
libemu
x86 emu

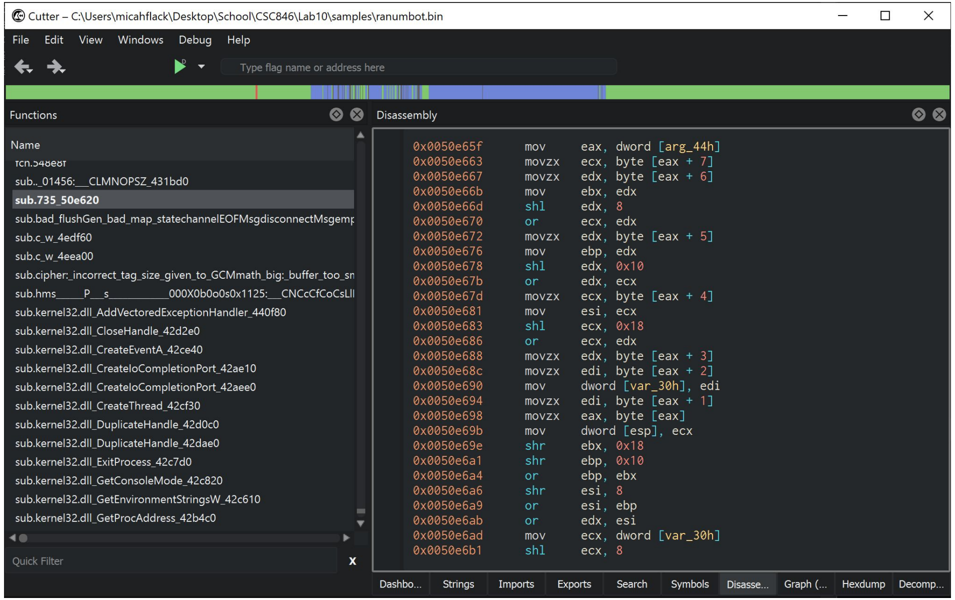
But there are actually more than that...

- Symbolic Data Analysis (SDA)
- Concolic Data Analysis (CDA)
- Code/Library Emulation

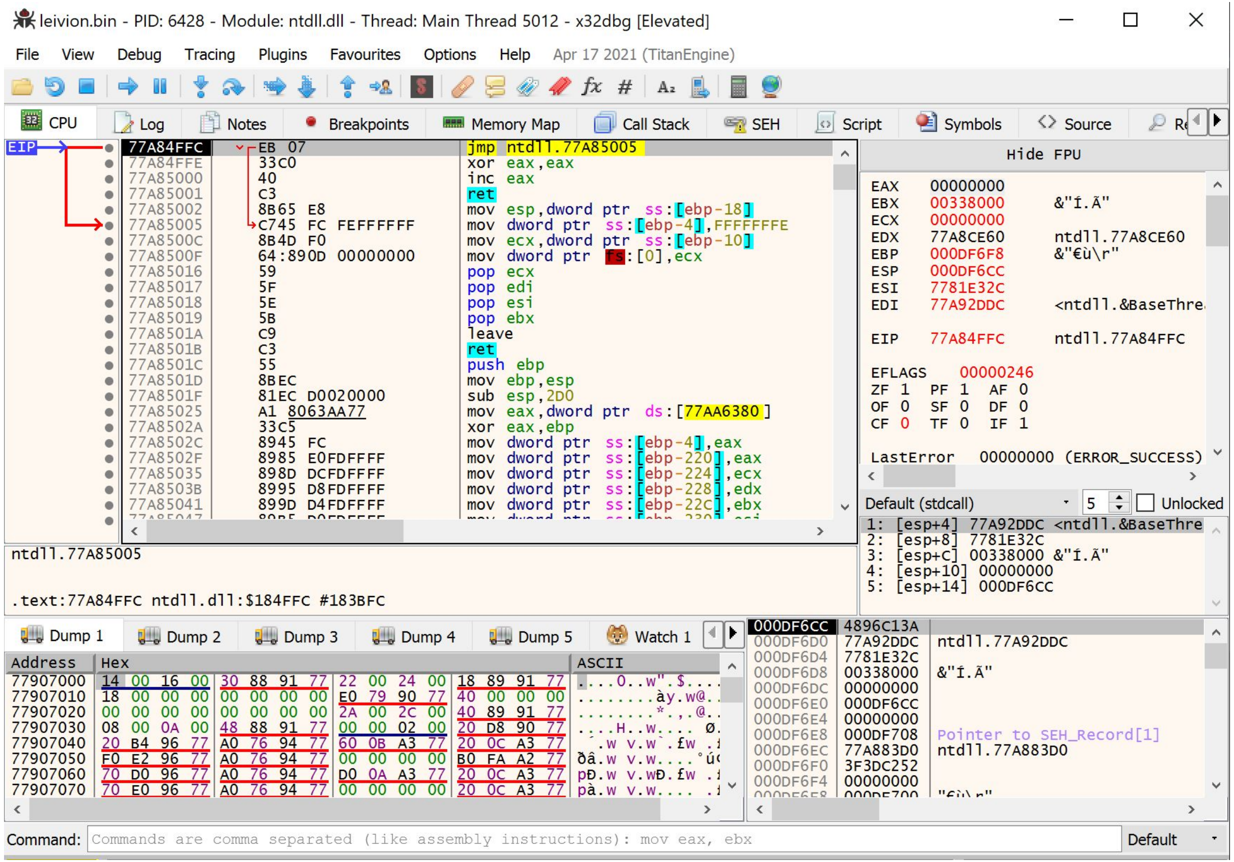


# Static Analysis

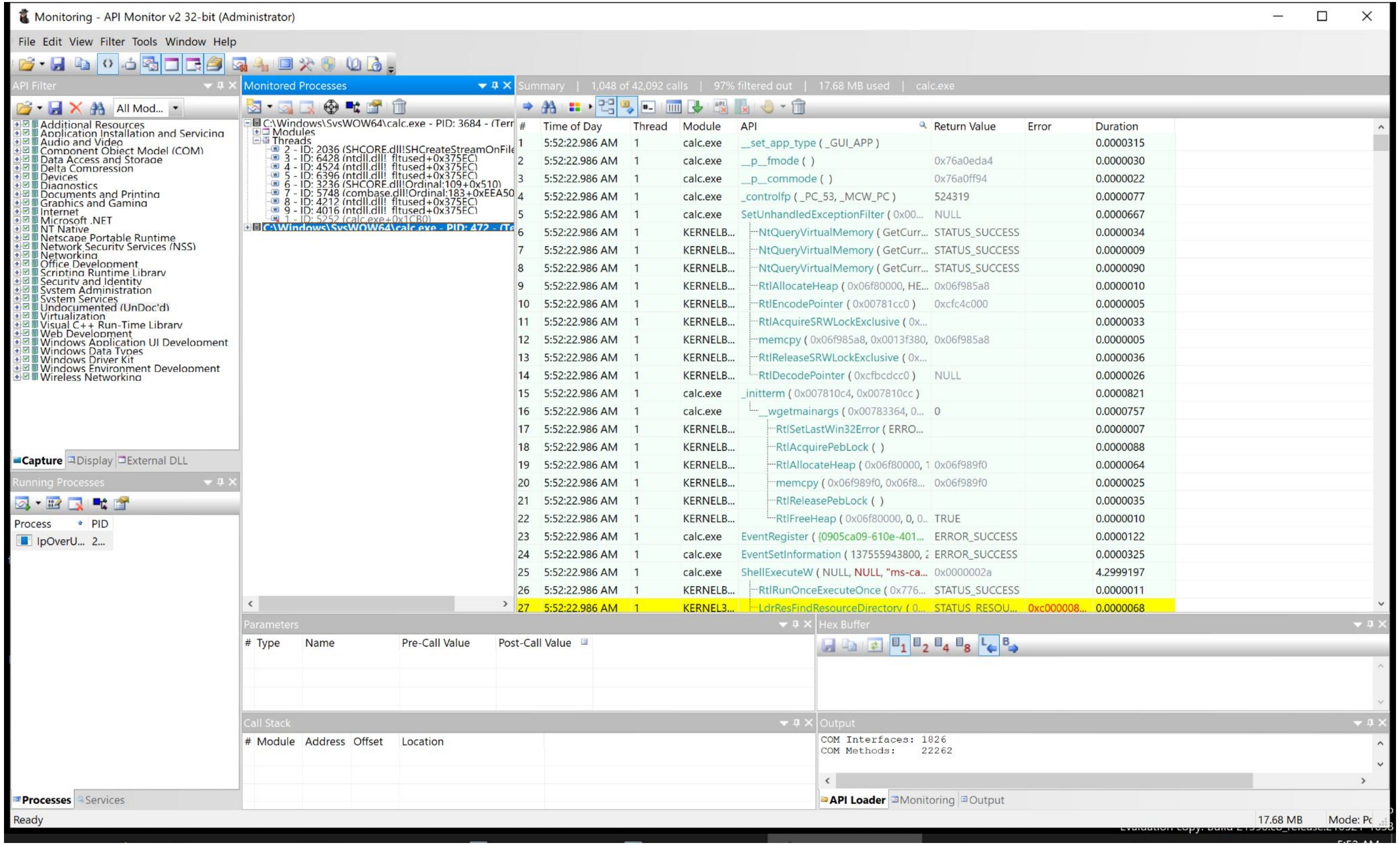


# Dynamic Analysis



# Dynamic Analysis



# What is Symbolic Execution?

Symbolic execution systematically explores as many possible execution paths at the same time without requiring concrete inputs.

Rather than taking on fully specified input values, the technique abstractly represents them as symbols, resorting to constraint solvers to construct actual instances that would cause property violations.

Pros, there is no…

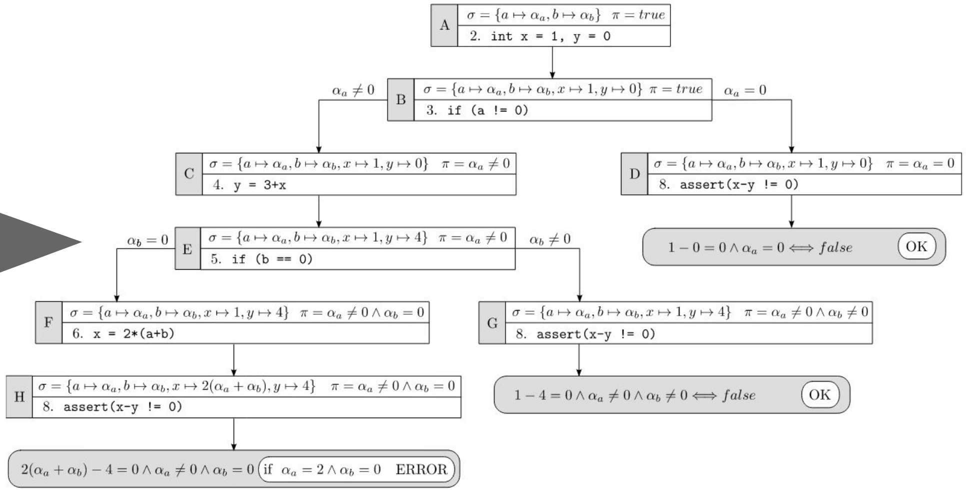
- … division by zero
- … NULL pointer ever dereferenced
- … backdoor that can bypass authentication
  - versus blindly debugging or detonating the sample

Cons:

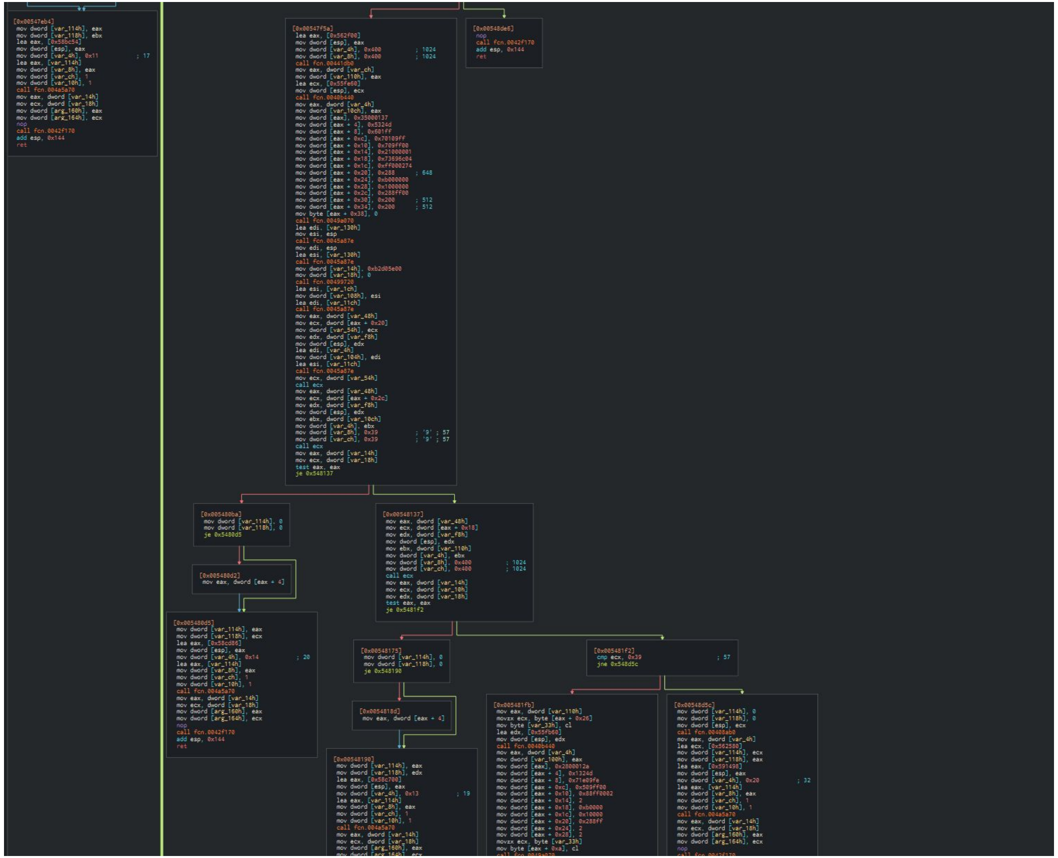
- No automated way to determine JMP table results

# For example...

```
void foobar(int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3+x;
        if (b == 0)
            x = 2*(a+b);
    }
    assert(x-y != 0);
}
```



# More familiarity with…



# Symbolic/Concolic Analysis

Essentially, the provided sample is disassembled and then static analysis is performed symbolically or concretely.

In symbolic analysis, symbolic execution traverses the code considering symbolic input variables in place of concrete (static) values.

- Complete exploration of Control Flow Graph (CFG) and all paths

In concolic (CONC-rete symb-OLIC) analysis, the execution trace of the disassembled code is guided by provided contextual information and therefore has similar drawbacks to dynamic analysis.

- Exploration is limited within CFG by the provided context or input.

# Using angr for symbolic analysis

angr is a multi-architecture binary analysis toolkit, with the capability to perform dynamic symbolic execution and various static analyses on binaries.

Some alternatives you might have used before for binary analysis:

- Pefile
- LIEF

angr is more powerful though because we can symbolically solve constraints within the binaries given limited complexity

- Single block, high recursion samples would cause most analyses to fail

# How are binaries read

angr uses the CLE loader to provide a representation of binary files into a virtual address space.

```
>>> proj.loader
<Loaded true, maps [0x400000:0x5004000]>

>>> proj.loader.shared_objects # may look a little different for you!
{'ld-linux-x86-64.so.2': <ELF Object ld-2.24.so, maps [0x2000000:0x2227167]>,
 'libc.so.6': <ELF Object libc-2.24.so, maps [0x1000000:0x13c699f]>}

>>> proj.loader.min_addr
0x400000
>>> proj.loader.max_addr
0x5004000

>>> proj.loader.main_object  # we've loaded several binaries into this project. Here's the main one!
<ELF Object true, maps [0x400000:0x60721f]>

>>> proj.loader.main_object.execstack  # sample query: does this binary have an executable stack?
False
>>> proj.loader.main_object.pic  # sample query: is this binary position-independent?
True
```

Typically some type of address rebasing for the entrypoint is used to load bins @ 0x400000

# Solver Engine

The part that we are most interested in is the solver engine methods used by angr

When a binary is loaded and the project object is provided, we not only have the symbolic states of each function but also their symbolic values. Rather than being limited to concrete, static values angr uses symbolic values (names) as placeholders - which we can then solve for as arithmetic problems given the provided abstract syntax tree (AST).

# Example of Constraint Solving

Registers and memory spaces are treated the same way - allowing us to enumerate the different paths, the provided constraints, and their possible solutions.

```
>>> state.solver.add(x > y)
>>> state.solver.add(y > 2)
>>> state.solver.add(10 > x)
>>> state.solver.eval(x)
4
```

# Potential Applications

Not limited to, but some of the ways you might leverage angr/symbolic analysis

- Code Audits
- Exploitation Development
- Fuzzing
- Binary Capture-The-Flags
- Malware Analysis

# Using angr for symbolic analysis

https://github.com/jakespringer/angr_ctf/tree/master/00_angr_find

# Code/Library Emulation

Instead of statically inspecting a sample, using somewhat complicated symbolic analysis, or even dynamic analysis that requires detonating a sample, you can use libemu emulation library.

Libemu makes it possible to emulate API functions from a program w/o risking infection; it can even be done with Windows shellcode on Linux.

Original project ➔ https://git.carnivore.it/libemu

Adopted and reworked by David Zimmer as scdbg.exe

- https://github.com/dzzie/scdbg_unicorn
- Already included with FlareVM under debuggers

# Example...

```
FLARE Tue 03/22/2022 13:47:28.50
C:\Users\micahflack\Desktop>scdbg /f ..\Downloads\micah
Loaded 209 bytes from file ..\Downloads\micah
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000

401043  LoadLibraryA(urlmon.dll)
401084  URLDownloadToFileA(https://github.com/micahflack/scripts/raw/main/notif.exe, not_malware.test)
4010a5  WinExec(.\not_malware.test)
4010b4  Sleep(0x5000)
4010c5  DeleteFileA(.\not_malware.test)
4010ce  ExitProcess(0)

Stepcount 103425


FLARE Tue 03/22/2022 13:47:39.00
C:\Users\micahflack\Desktop>_
```