SYMBOLIC OF THE FUTURE

James Anderson

SYMBOLIC OF THE FUTURE

6 reasons why symbolic execution is better than IDA

Number 5 will blow you mind!

James Anderson

SYMBOLIC OF THE FUTURE

6 reasons why symbolic execution is better than IDA

Number 5 will blow you mind!

BUZZFEED EDITION

James Anderson

Get the presentation:

malwr.co/symbolic

(I swear it’s not malware)

Who am I?:

- Malware reverse engineer 5 years
- Security Engineer 2 years

#1

Reverse engineering is great but if it takes more than a day it takes **too long**.

Grab the pitchforks lads!

Grab the pitchforks!

Hear me out first...

(besides this is buzzfeed worthy clickbait so now you have to listen)

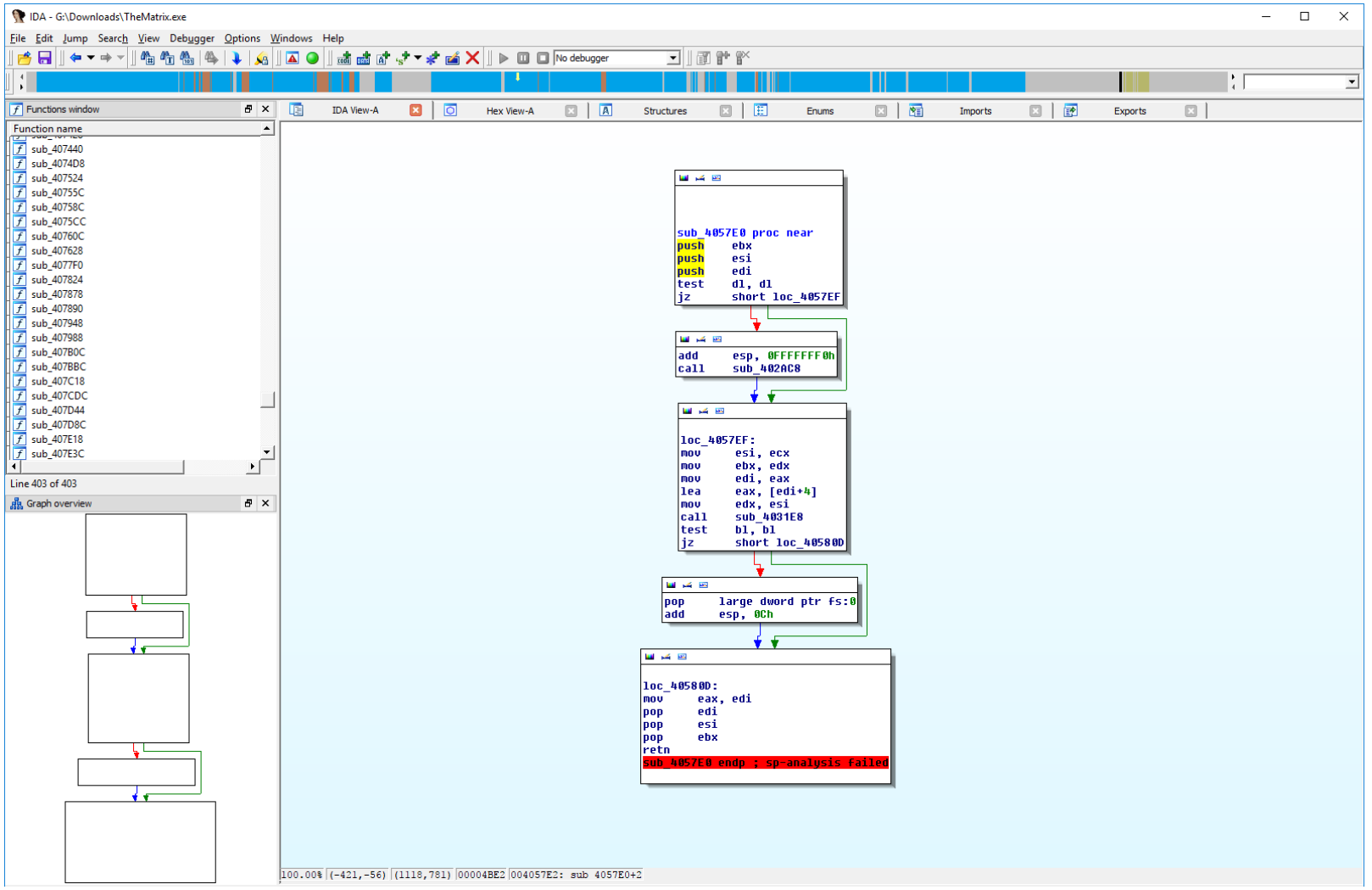
How every task starts

Let’s do this

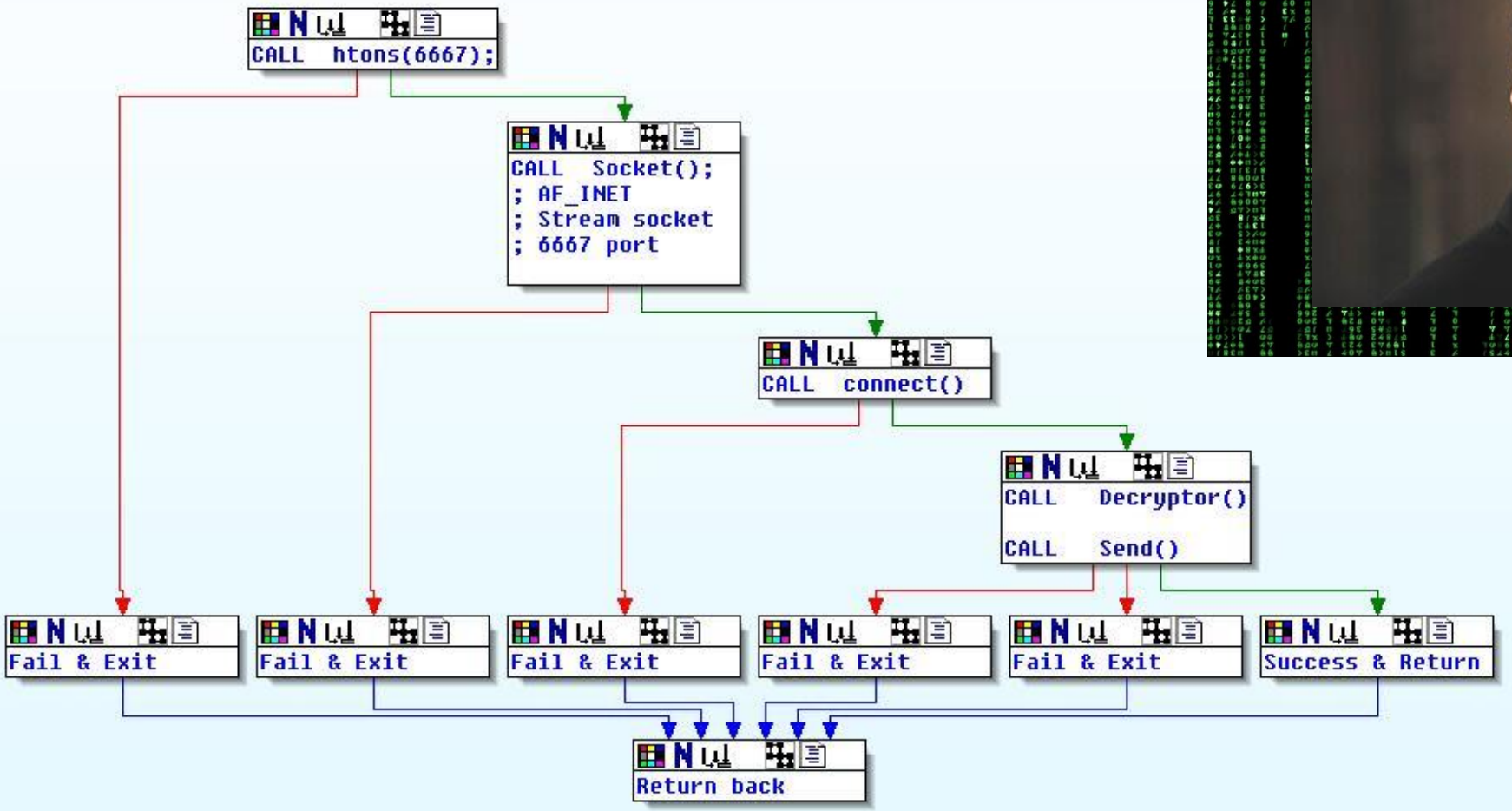

It's Saturday night,
I have no date,
a two-liter bottle of Shasta
and my all-Rush mix-tape...
Let's rock.
RUSH

How every task starts

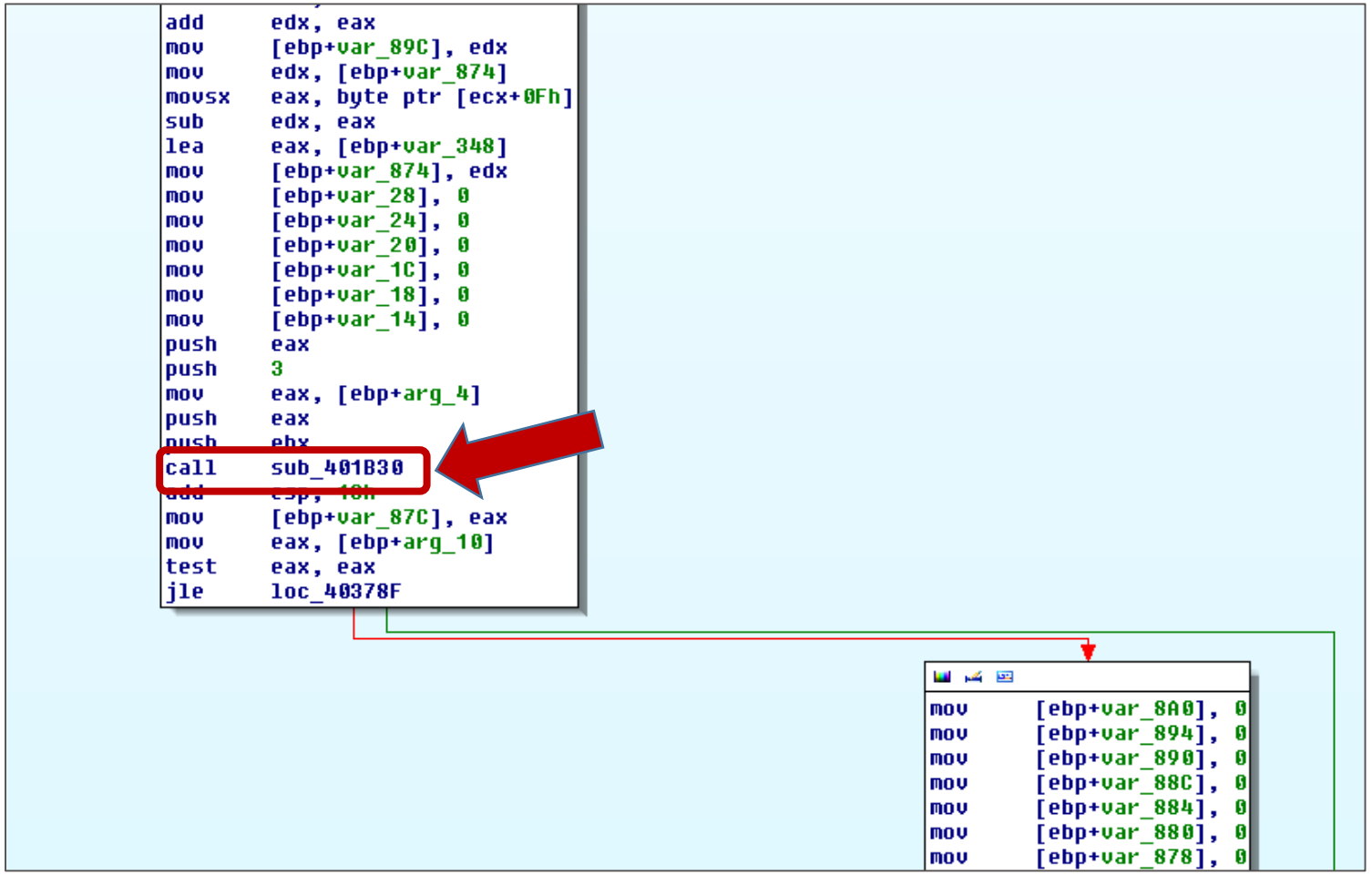
Looks pretty straight forward



I’m seeing into the matrix…



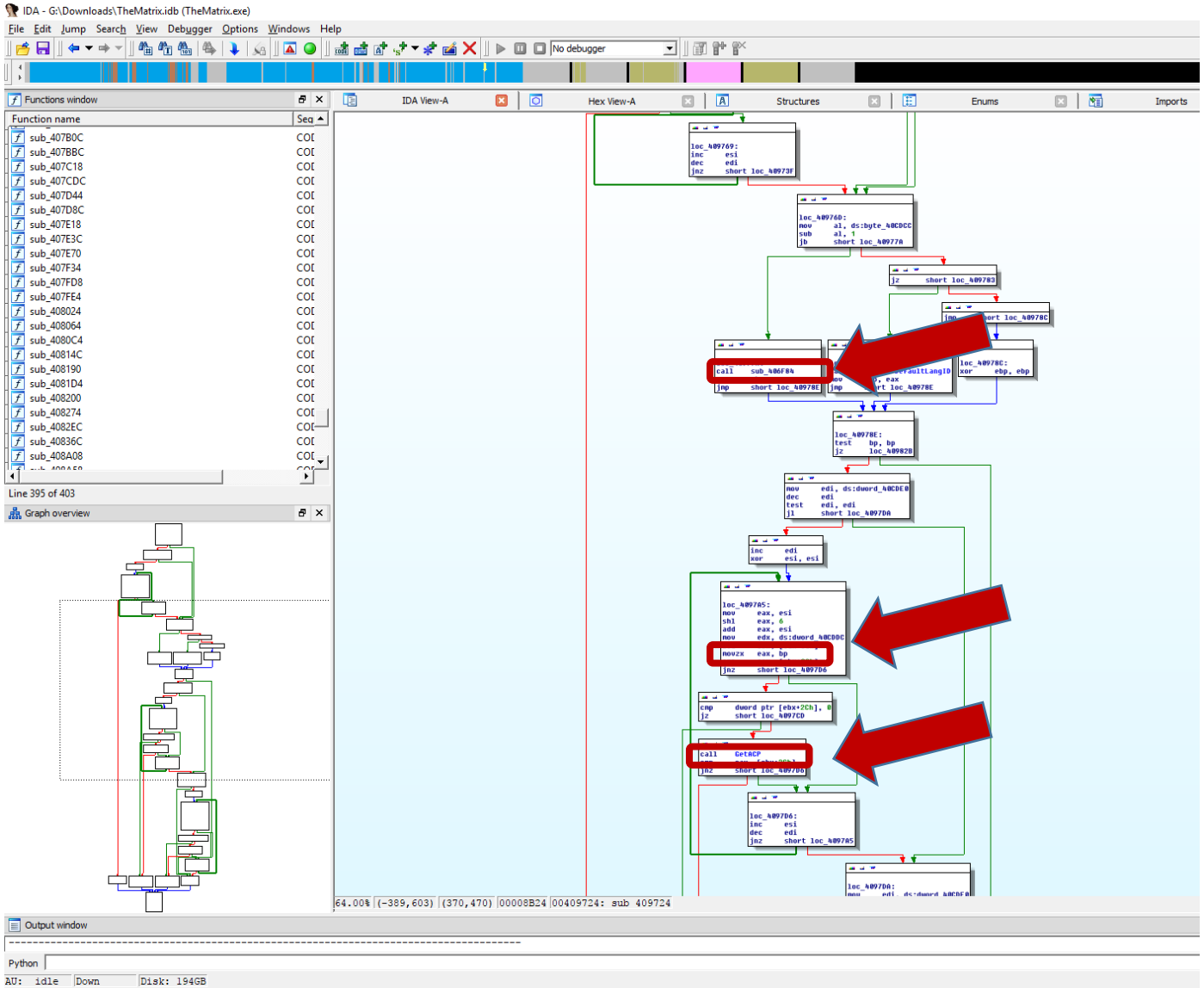
The Investigation



Wait a second!

What does this call do?

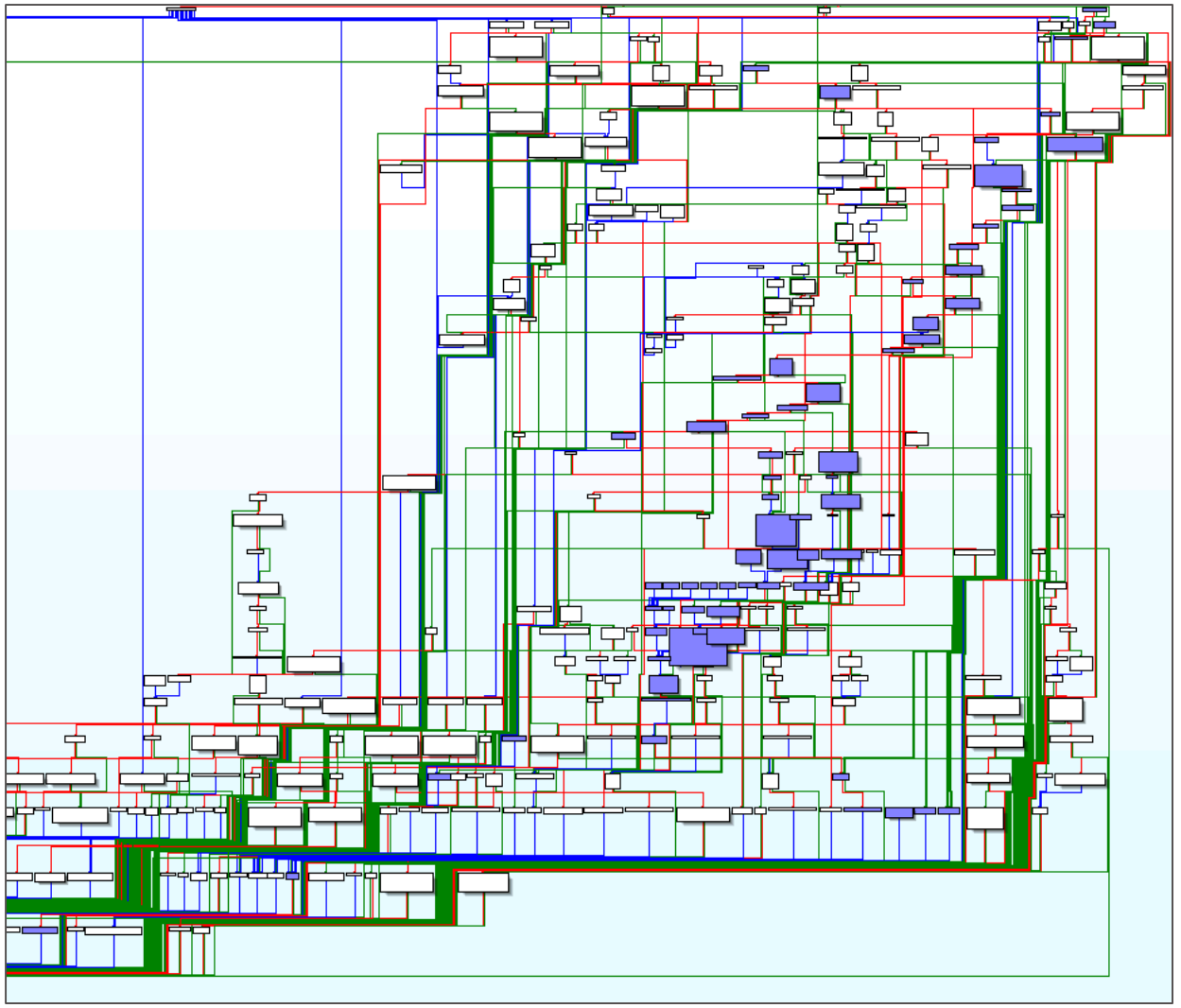
The Investigation



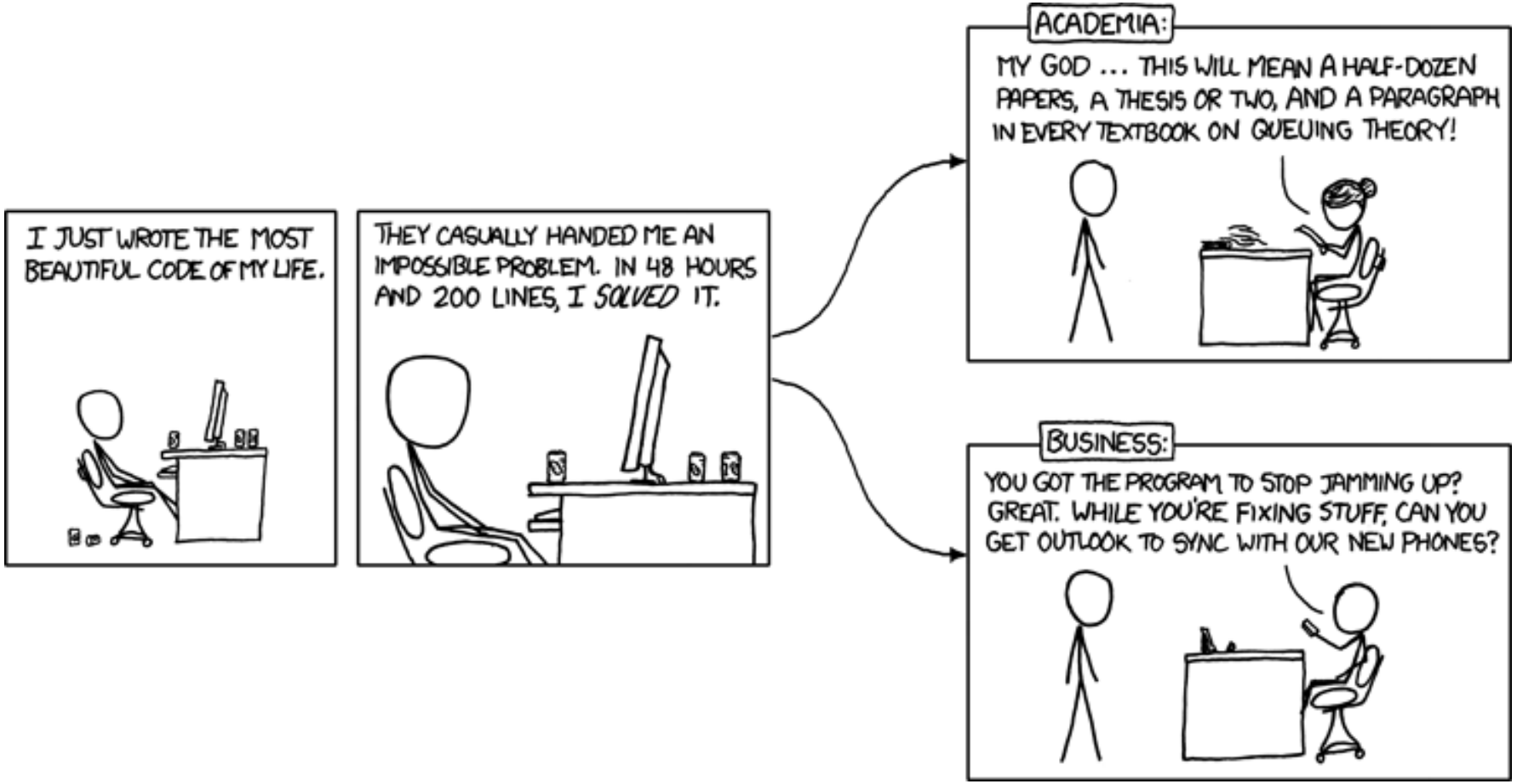
What is this!
Even more calls

Plz Help...

- Plz halp, I've gone too deep.

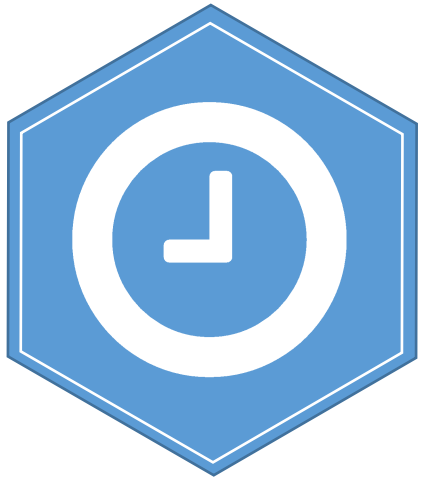


Speed is the key



https://xkcd.com/664/

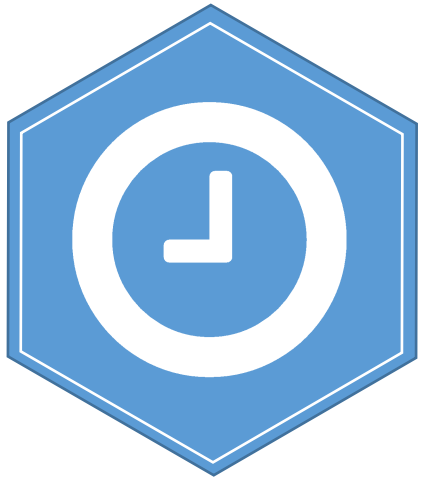
What was I looking at? Ah nevermind...



Time Matters

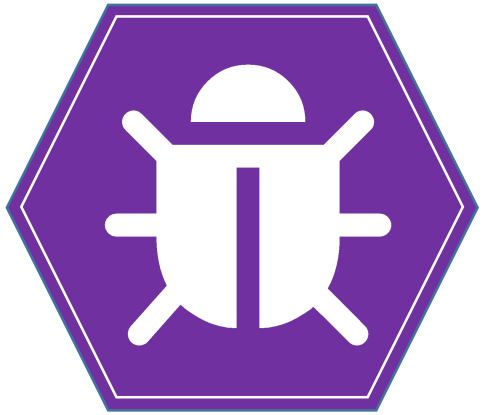
If others are waiting on the results then they will move on quickly

What was I looking at? Ah nevermind...



Time Matters

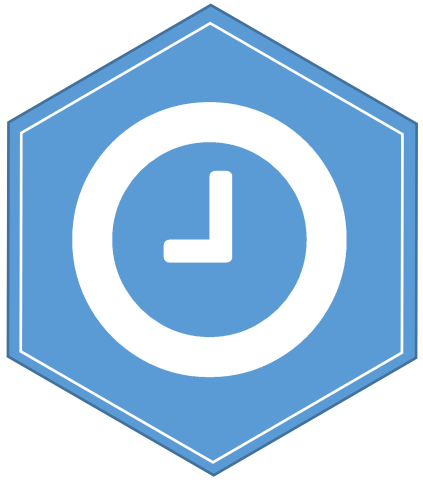
If others are waiting on the results then they will move on quickly



Tracking Bugs

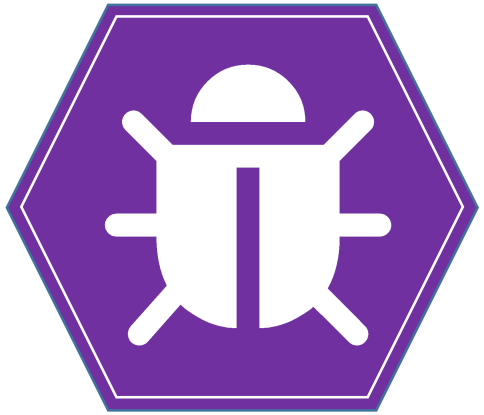
How long can we do it for? I want results but how soon?

What was I looking at? Ah nevermind...



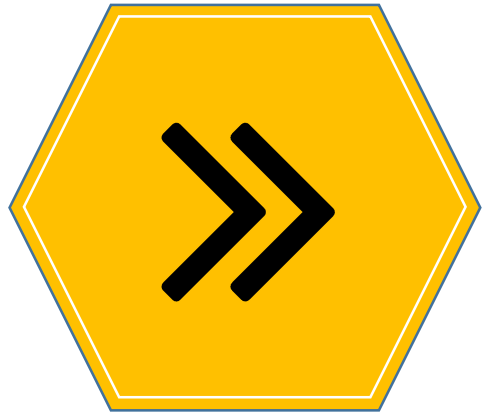
Time Matters

If others are waiting on the results then they will move on quickly



Tracking Bugs

How long can we do it for? I want results but how soon?



Faster at the game

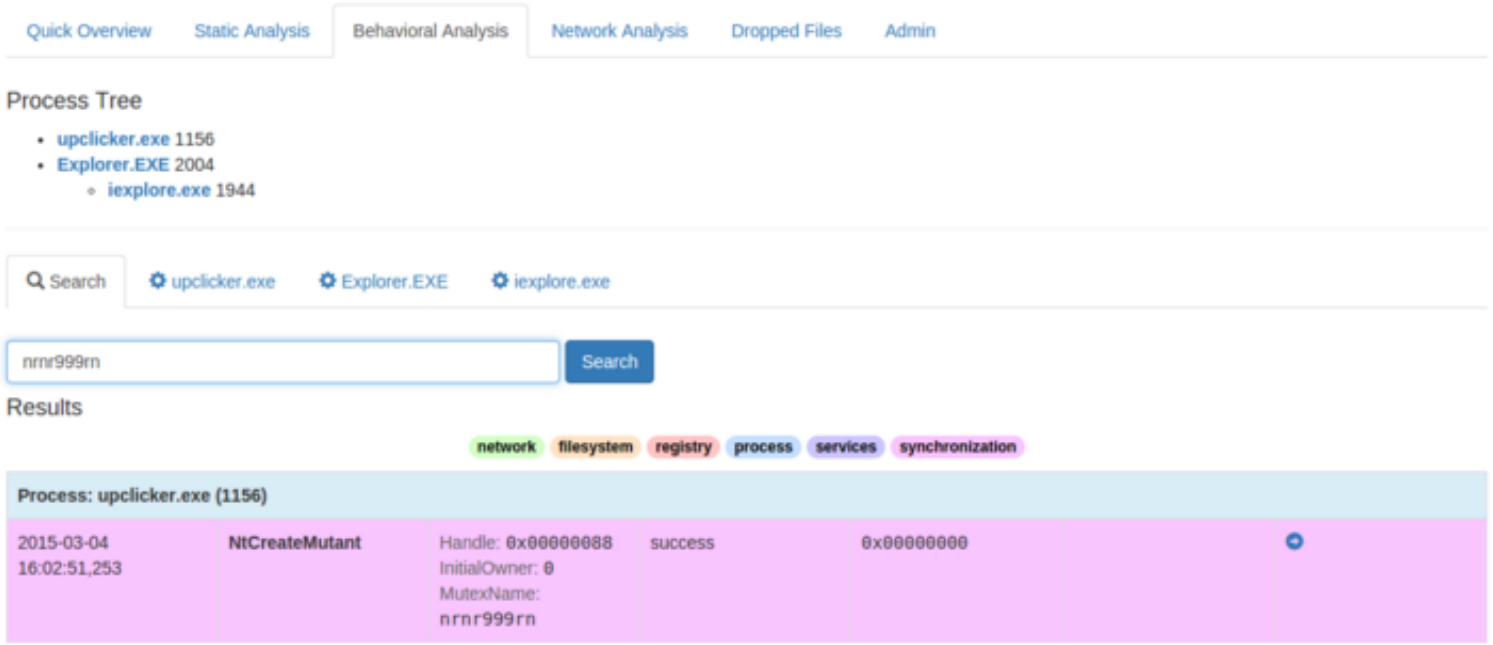
If others are waiting on the results then they will move on quickly

#2

Dynamic is fast and efficient, but you always lose out on the details.

No one is surprised by this

Dynamic execution like cuckoo gives us some of the answers but it only gets us part of the way.



No one is surprised by this

While sandboxes give good detail they lose the details.

- How are the comms encrypted?
- What other persistence methods are there?
- Does it have any anti-vm/ anti debugger techniques
- What did we miss?

Static gives the detail but can be slow

- Many unresolved symbols
- Debugger side by side to help analysis

What if there was another way?

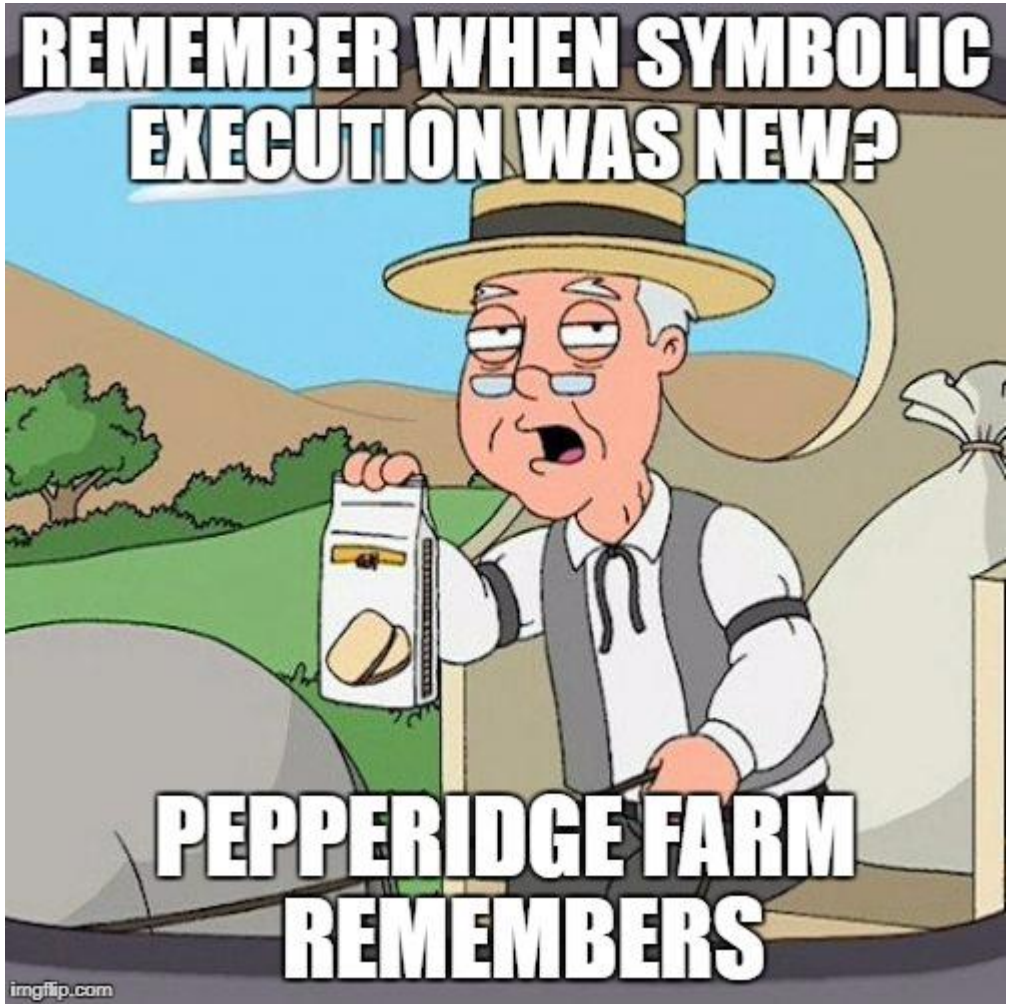
Symbolic Execution

Symbolic execution is testing technique to aid the generation of test data and in proving the program quality

Symbolic execution is simply making some data ‘symbolic’ in the same way we do for formulas.

$$2x + 7 = 23$$

This is nothing new...



#3

Symbolic execution is the 80% solution you need.

Symbolic Execution

Remember we are trying to get answers fast

- We aren't trying to work out everything
- Just the core details
- URL's, comms , persistence, lateral movement, second stage

If we can get the detail sooner even if we don't fully understand it we still get what we (and others) need.

Symbolic Execution

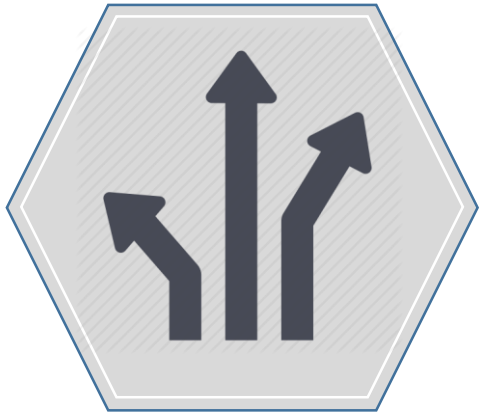
Concrete Execution

Standard execution of a program with defined variables (i.e. x=5)

Symbolic Execution

Concrete Execution

Standard execution of a program with defined variables (i.e. x=5)



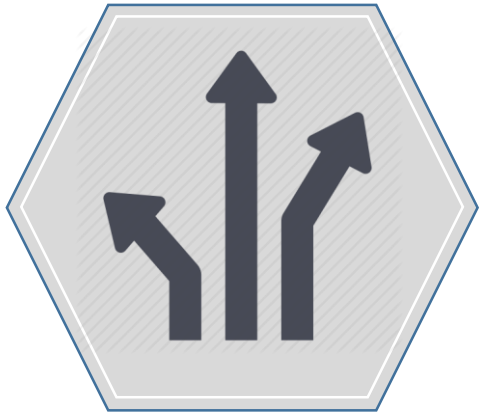
Symbolic Execution

Execute a program through all possible execution paths, thus achieving all possible path conditions

Symbolic Execution

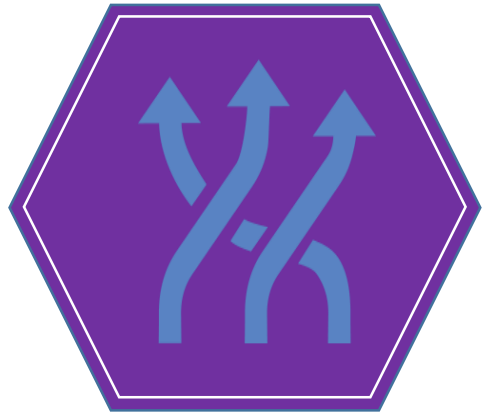
Concrete Execution

Standard execution of a program with defined variables (i.e. x=5)



Symbolic Execution

Execute a program through all possible execution paths, thus achieving all possible path conditions



Concolic Execution

Concolic execution is a mix between CONCrete execution and symbOLIC execution, guiding it through a specific execution path

Concrete Execution

Concrete execution

- Perform actions based on the hard data that is passed in.
- X and y are hard values (i.e x =5, y=7)

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

Symbolic Execution - Primer

Symbolic Execution

Maintains a symbolic states of registers and part of memory at each program point.

- a table of symbolic registers states
- a map of symbolic memory states
- a global set of all symbolic references

Dynamic forward symbolic execution

Dynamic forward symbolic execution builds a logical formula describing a program execution path

Dynamic forward symbolic execution

Dynamic forward symbolic execution builds a logical formula describing a program execution path

Symbolic execution proceeds along both branches, by "forking" two paths. Each path gets assigned a copy of the program state at the branch instruction as well as a path constraint

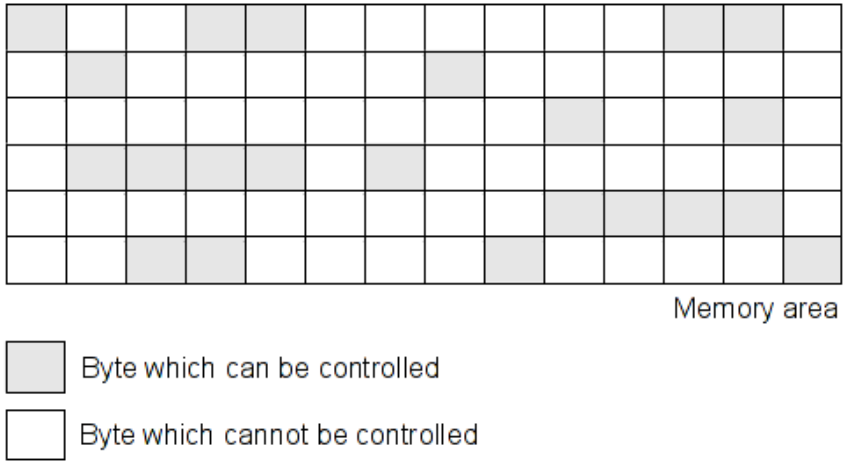
λ * 2 == 12 for the then branch and λ * 2 != 12

```
 1  int f() {
 2
 3   y = read();
 4   z = y * 2;
 5   if (z == 12) {
 6     fail();
 7   } else {
 8     printf("OK");
 9   }
10 }
```

Taint Analysis

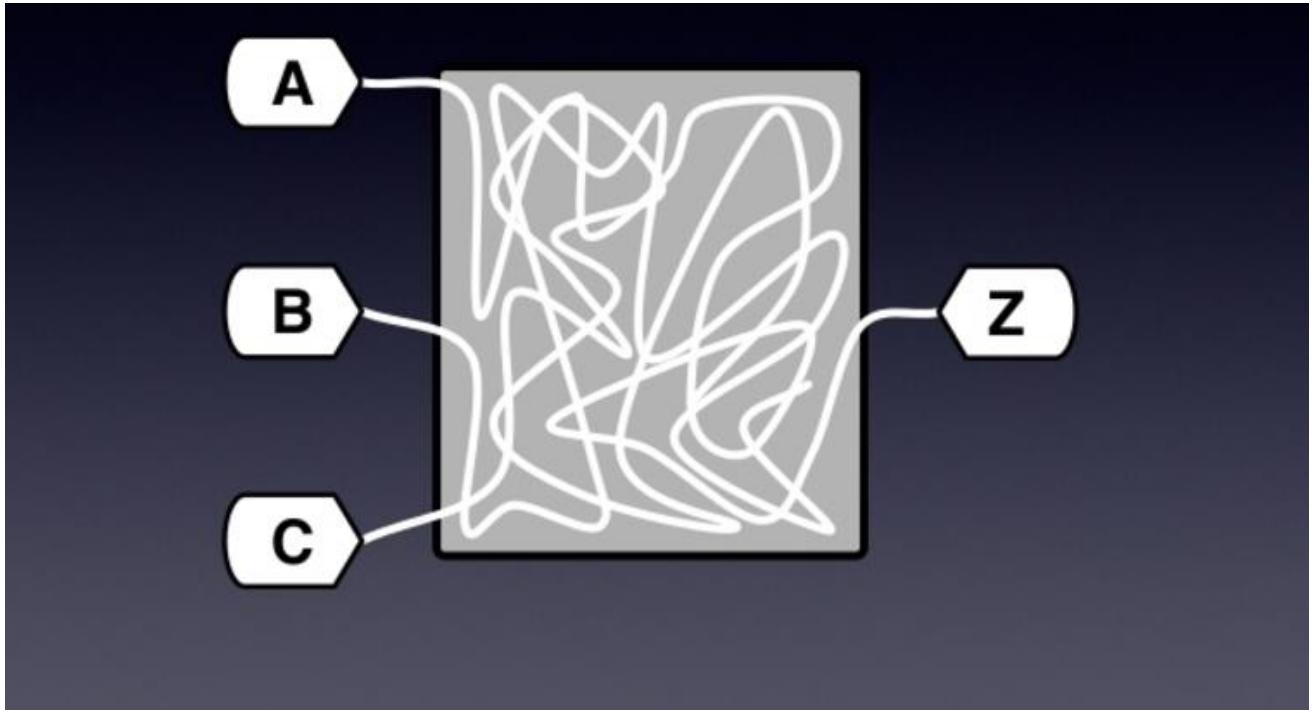
The purpose of dynamic taint analysis is to track the information flow from the sources (usually user inputs) to the targets (such as control-flow value).

It is thus capable of analyzing which region of the memory and registers are controllable by user inputs.



Taint Analysis

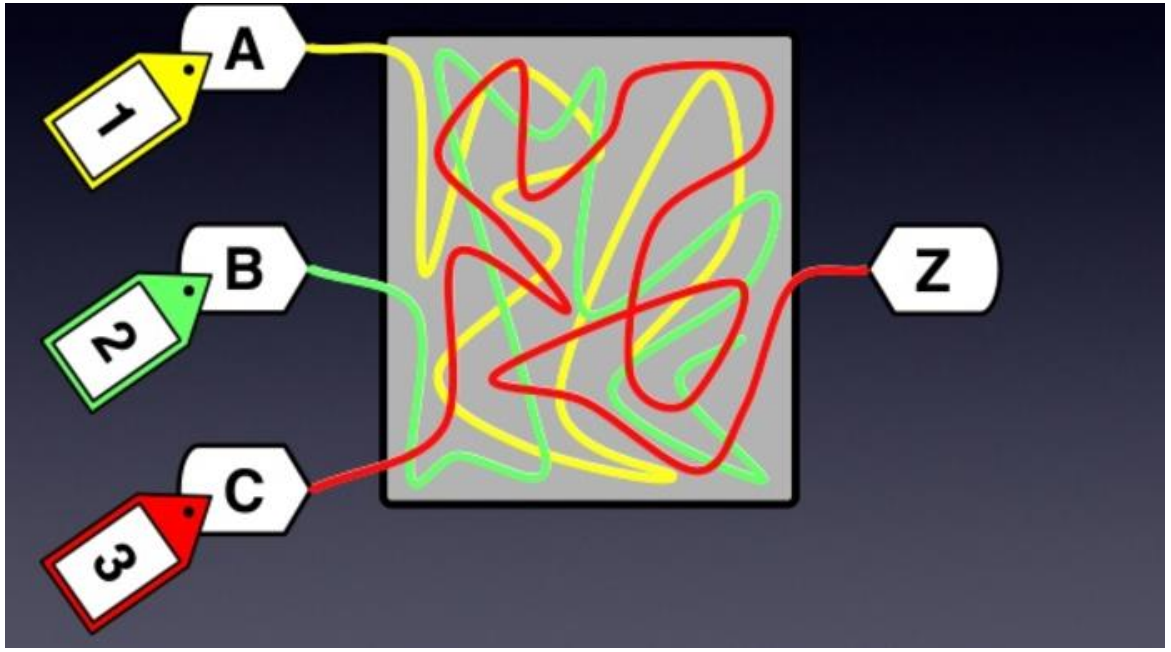
With this method it is possible to check the registers and the memory areas which can be controlled by the user when a crash occurs



https://www.slideshare.net/jimclause/penumbra-automatically-identifying-failurerelevant-inputs-issta-2007

Taint Analysis

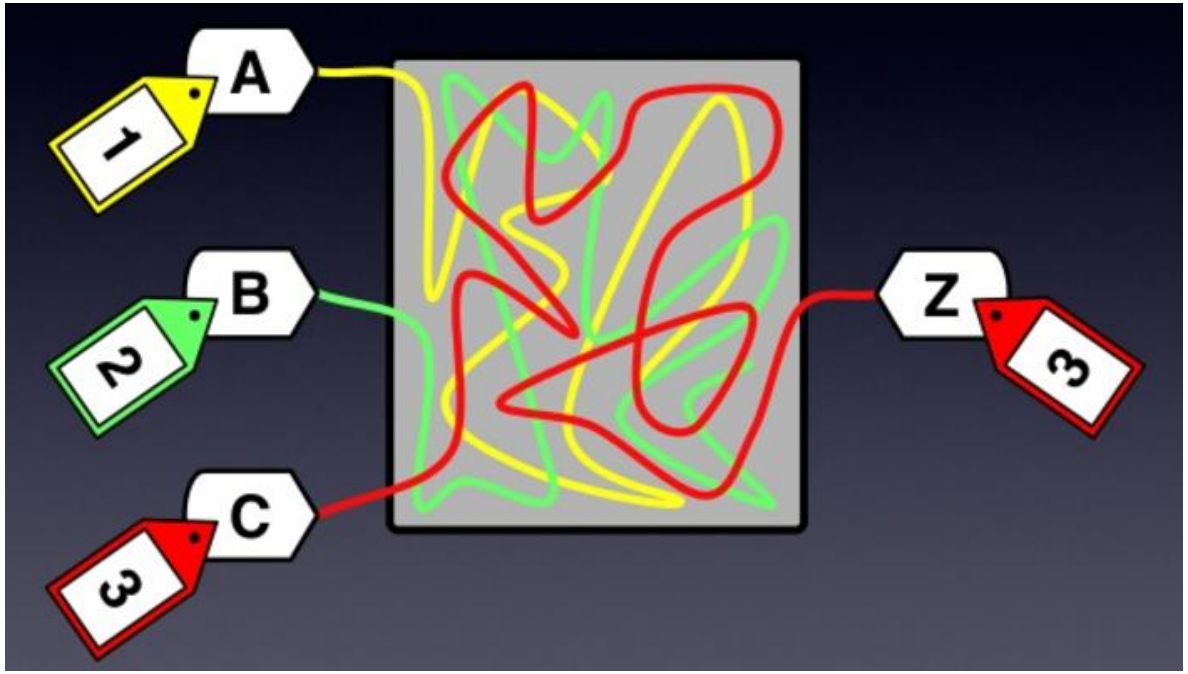
With this method it is possible to check the registers and the memory areas which can be controlled by the user when a crash occurs



https://www.slideshare.net/jimclause/penumbra-automatically-identifying-failurerelevant-inputs-issta-2007

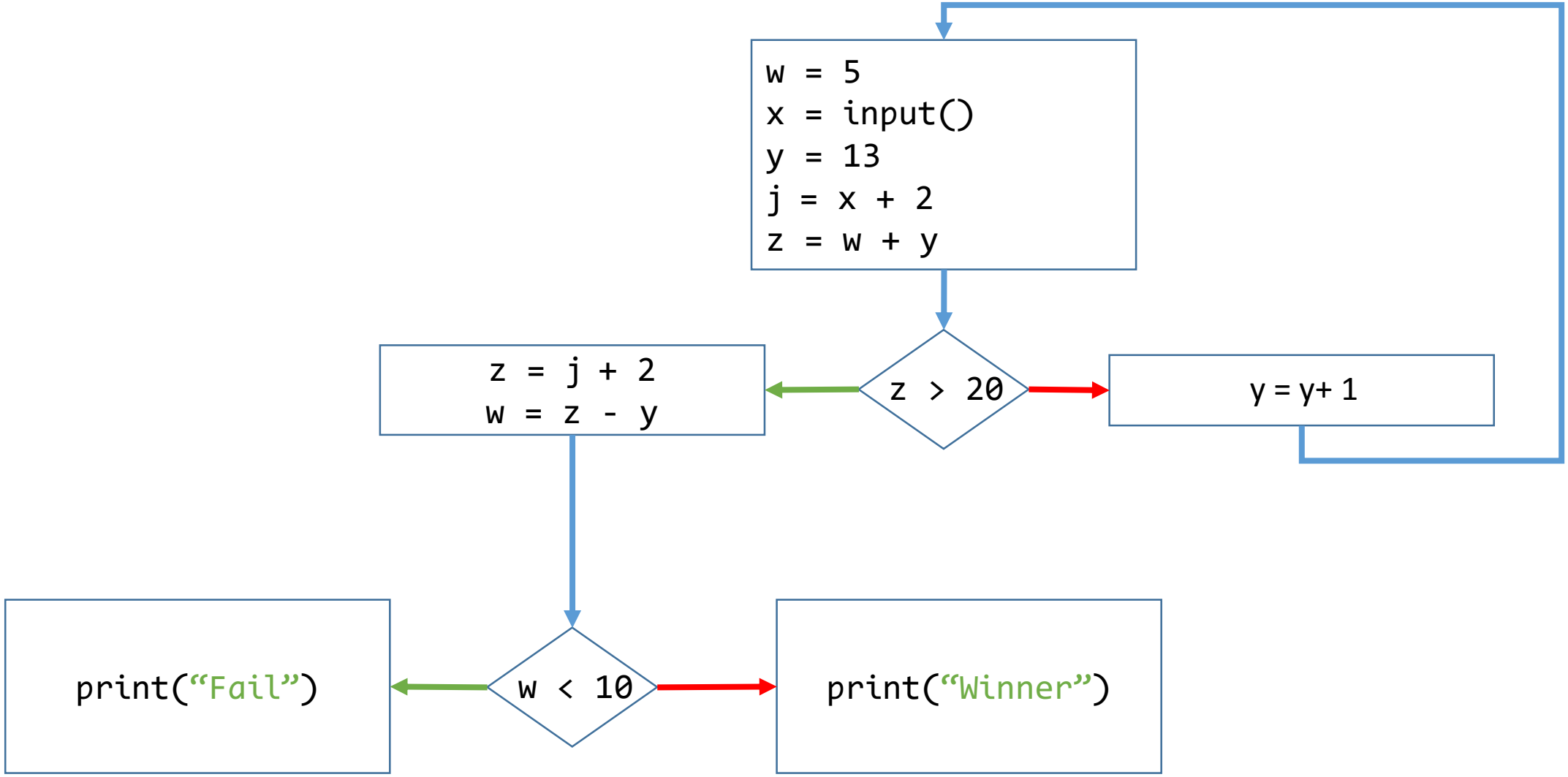
Taint Analysis

With this method it is possible to check the registers and the memory areas which can be controlled by the user when a crash occurs



https://www.slideshare.net/jimclause/penumbra-automatically-identifying-failurerelevant-inputs-issta-2007

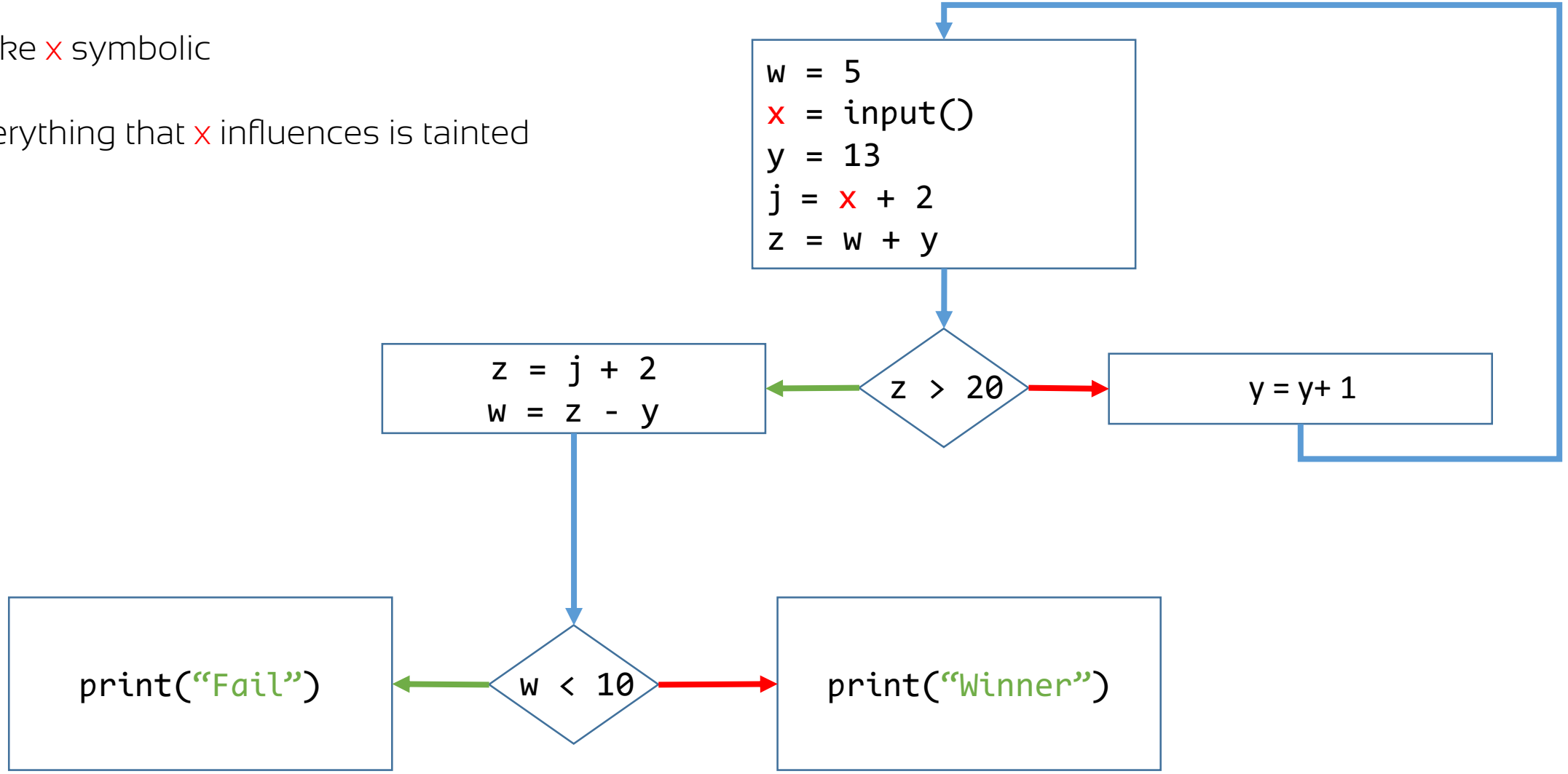
Symbolic Execution – Taint Analysis



Symbolic Execution – Taint Analysis

Make x symbolic

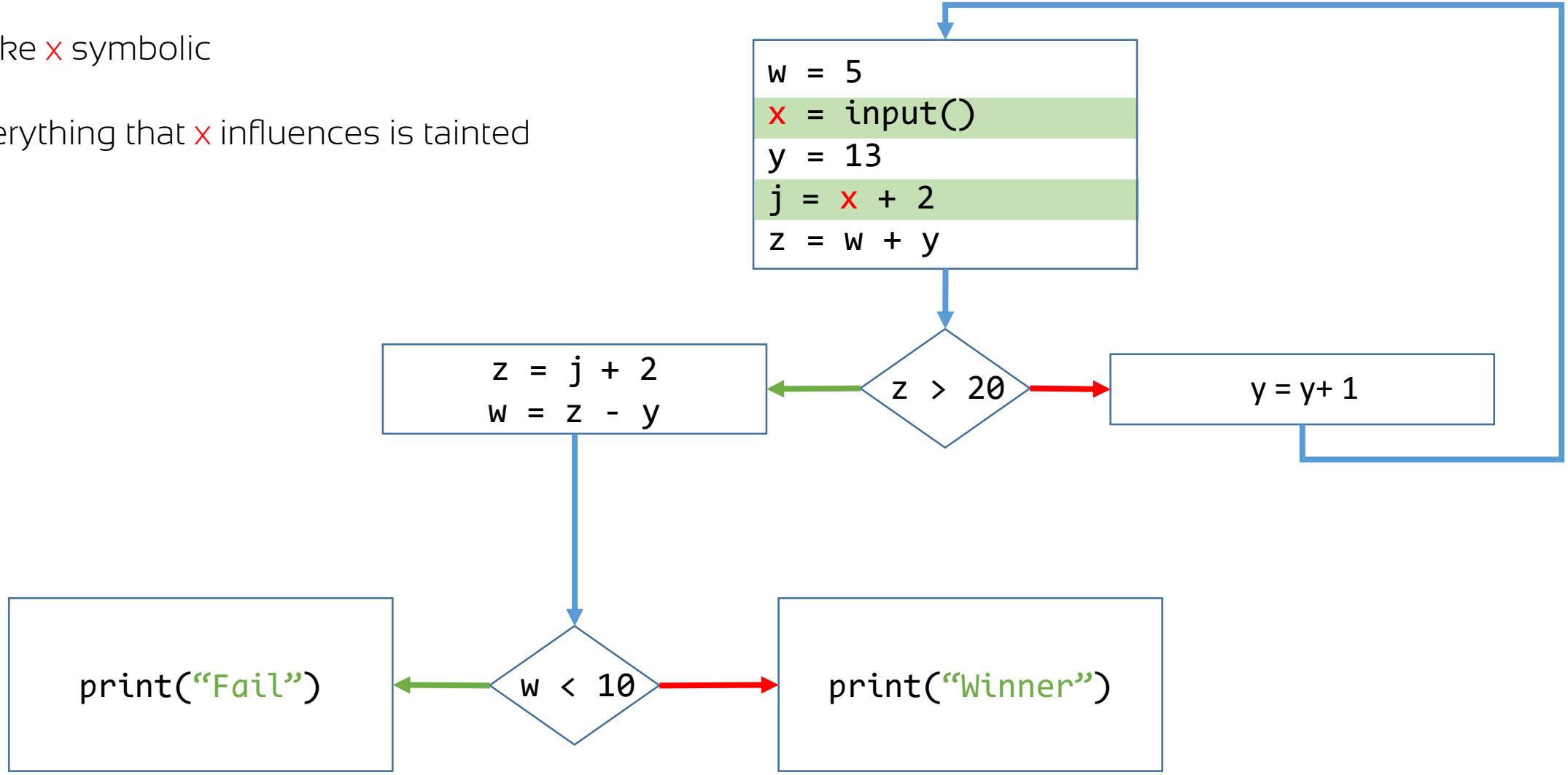
Everything that x influences is tainted



Symbolic Execution – Taint Analysis

Make x symbolic

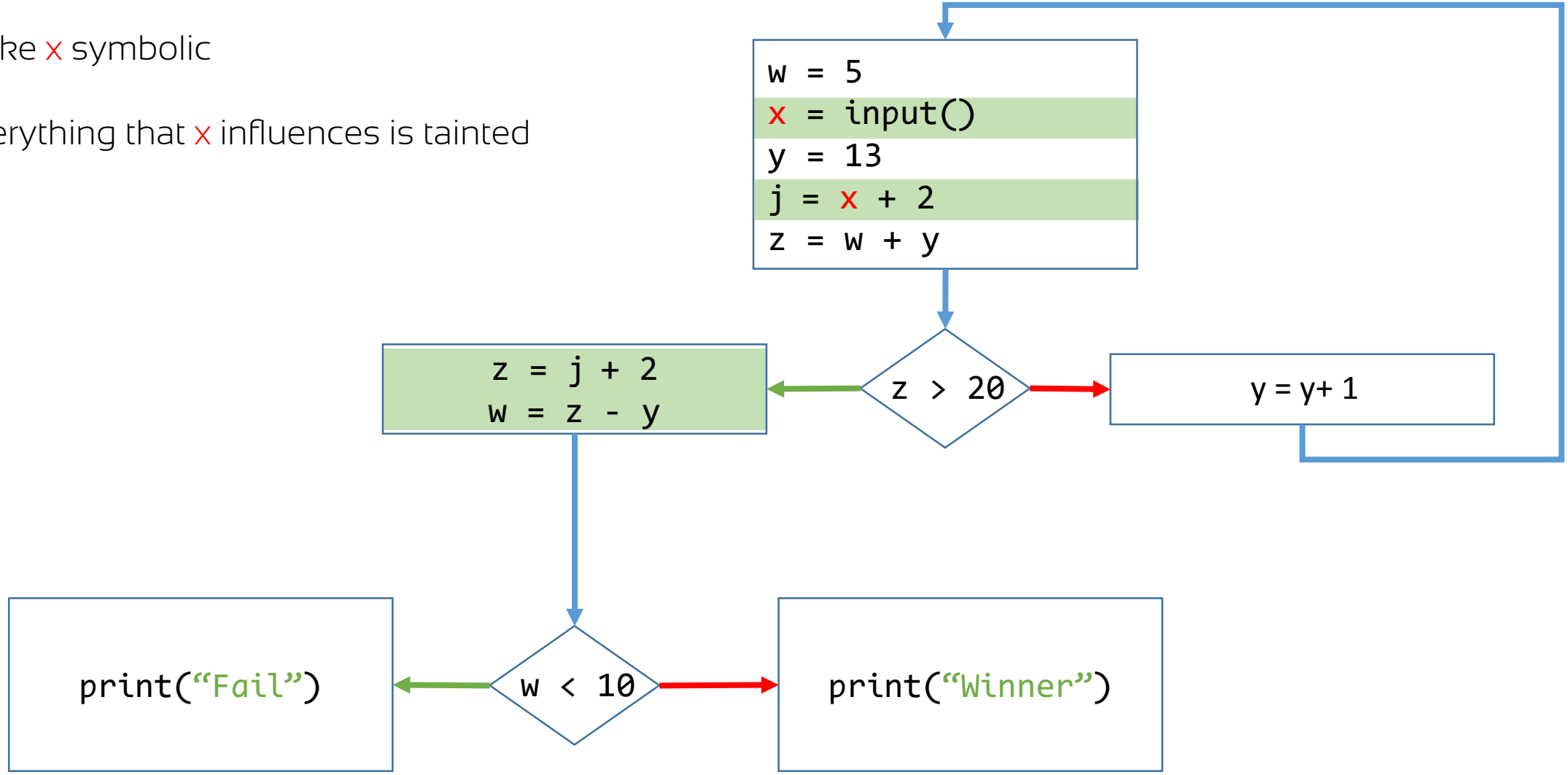
Everything that x influences is tainted



Symbolic Execution – Taint Analysis

Make x symbolic

Everything that x influences is tainted

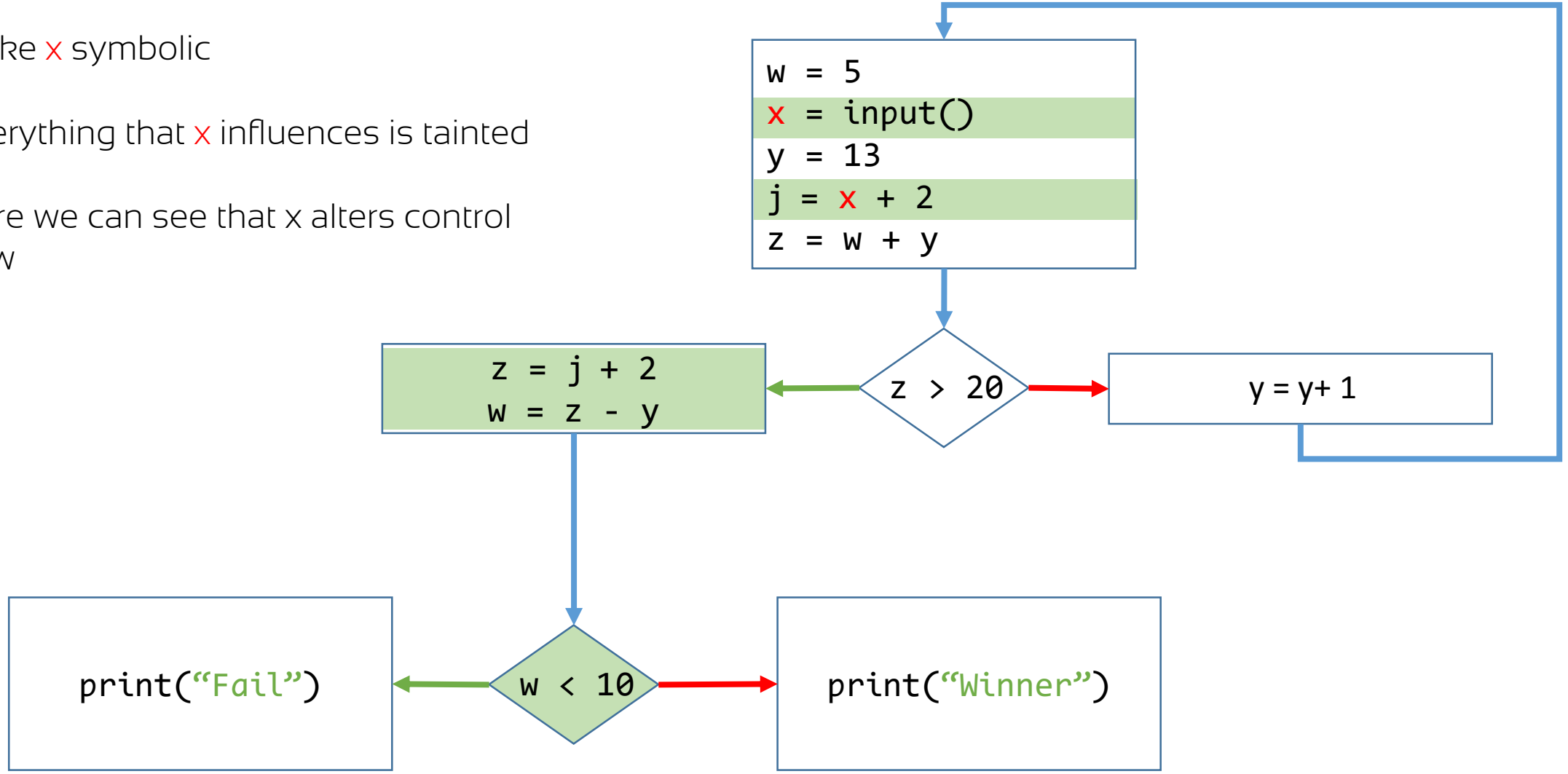


Symbolic Execution – Taint Analysis

Make x symbolic

Everything that x influences is tainted

Here we can see that x alters control flow



Symbolic Execution Limitations

Path Explosion

- Symbolically executing all feasible program paths does not scale to large programs.

Symbolic Execution Limitations

Path Explosion

- Symbolically executing all feasible program paths does not scale to large programs.
- The number of feasible paths in a program grows exponentially with an increase in program size.

Symbolic Execution Limitations

Path Explosion

- Symbolically executing all feasible program paths does not scale to large programs.
- The number of feasible paths in a program grows exponentially with an increase in program size.

Environment Interactions

- Symbolic execution that requires a file or user input will have consistency problems.

Symbolic Execution Limitations

Path Explosion

- Symbolically executing all feasible program paths does not scale to large programs.
- The number of feasible paths in a program grows exponentially with an increase in program size.

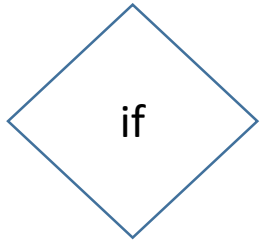
Environment Interactions

- Symbolic execution that requires a file or user input will have consistency problems.
- File operations are implemented as system calls in the kernel, and are outside the control of the symbolic execution

Concolic Execution

Concolic execution

- Performs symbolic execution along a concrete execution path



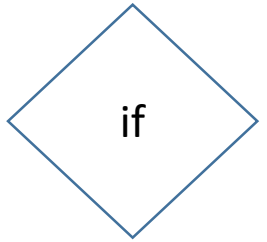
- Simplifying symbolic execution in the example

```
1  void f(int x, int y) {
2      int z = 2*y;
3      if (x == 100000) {
4          if (x < z) {
5              assert(0); /* error */
6          }
7      }
8  }
```

Concolic Execution

Concolic execution

- *Let x =1 y = 1*
- *Z=2*

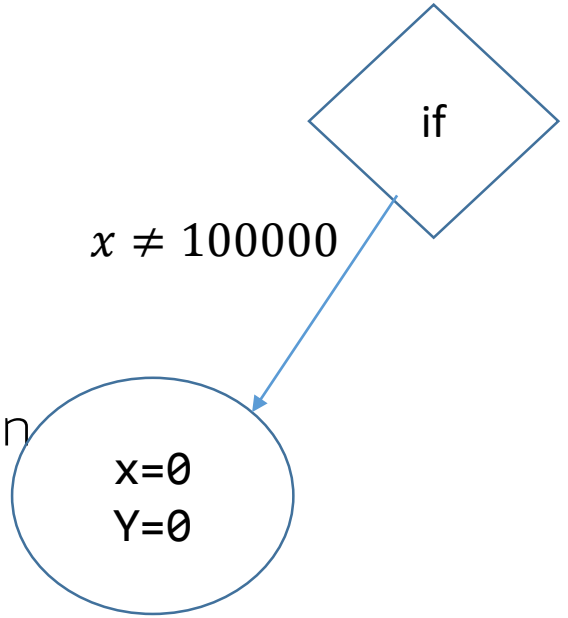



```
1  void f(int x, int y) {
2      int z = 2*y;
3      if (x == 100000) {
4          if (x < z) {
5              assert(0); /* error */
6          }
7      }
8  }
```

Concolic Execution

Concolic execution

- *Let x = y =* 1
- *Z=2*
- Line 3 fails because since 1 ≠ 100000
- From the inequality we create a path condition

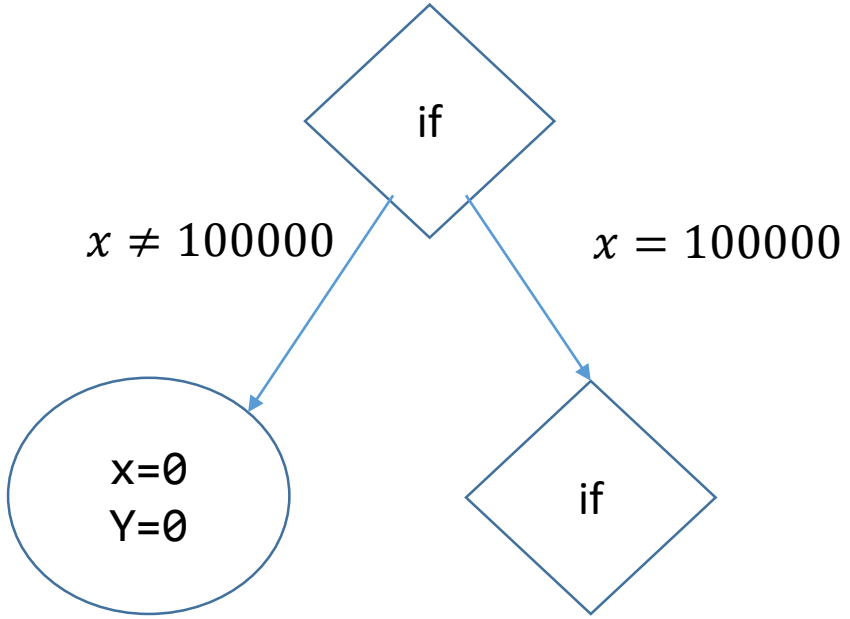



```
1  void f(int x, int y) {
2      int z = 2*y;
3      if (x == 100000) {
4          if (x < z) {
5              assert(0); /* error */
6          }
7      }
8  }
```

Concolic Execution

Concolic execution

- Try a different path, let x = 10000

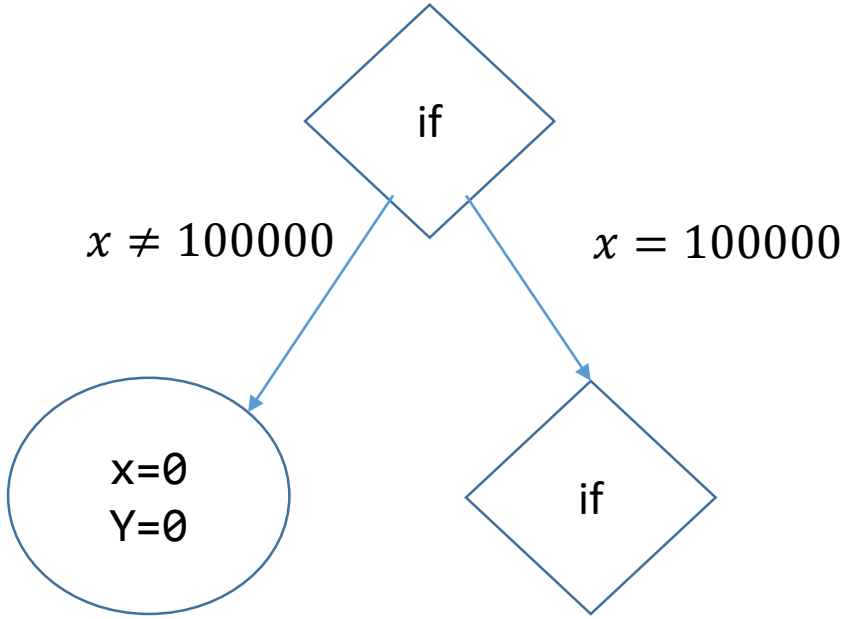



```
1  void f(int x, int y) {
2      int z = 2*y;
3      if (x == 100000) {
4          if (x < z) {
5              assert(0); /* error */
6          }
7      }
8  }
```

Concolic Execution

Concolic execution

- Try a different path, let x = 10000
- Automated theorem prover is then invoked to find values for the input variables *x* and *y*



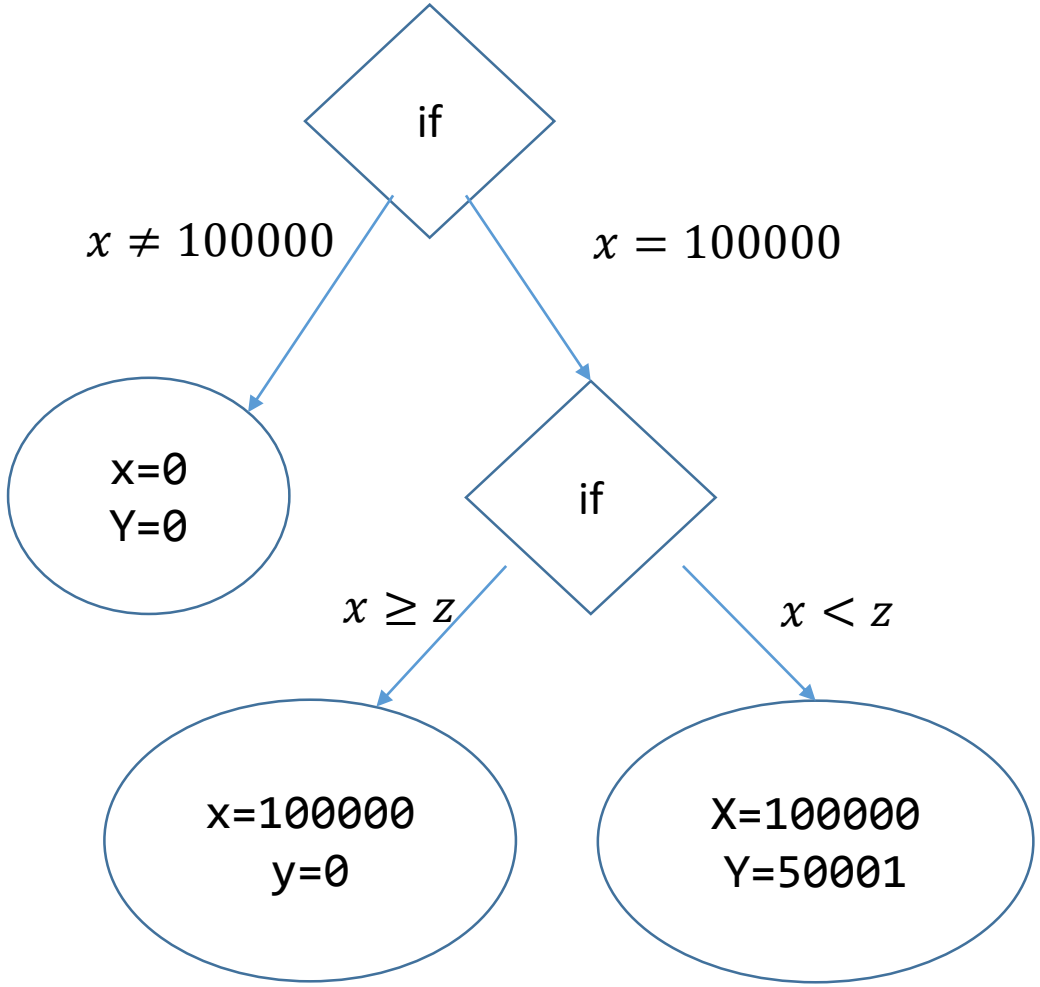

```
1  void f(int x, int y) {
2      int z = 2*y;
3      if (x == 100000) {
4          if (x < z) {
5              assert(0); /* error */
6          }
7      }
8  }
```

Concolic Execution

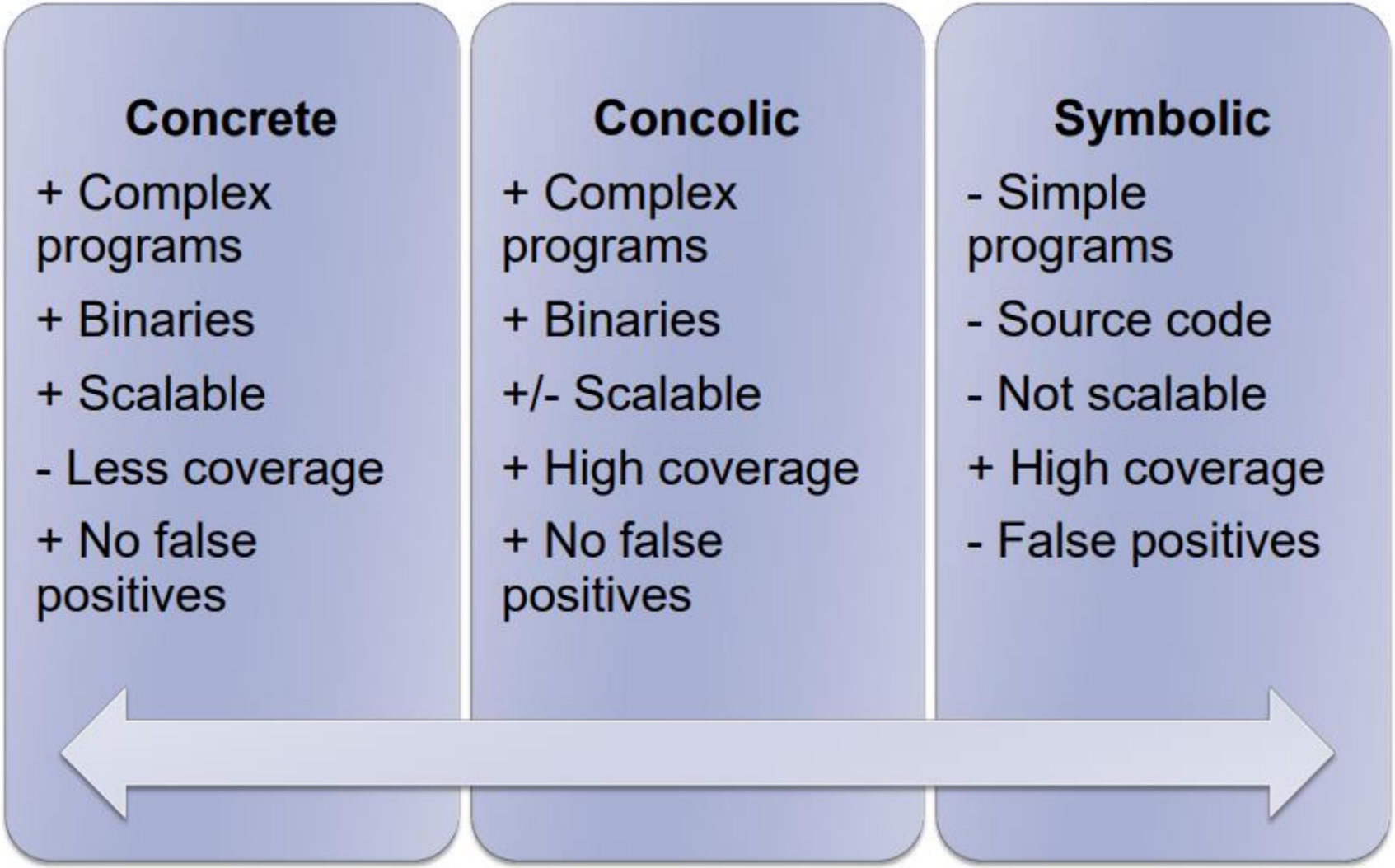
Concolic execution

- Try a different path, let x = 10000
- Automated theorem prover is then invoked to find values for the input variables *x* and *y*
- *A valid theorem might be x* = 100000, *y* = 0

```
1  void f(int x, int y) {
2      int z = 2*y;
3      if (x == 100000) {
4          if (x < z) {
5              assert(0); /* error */
6          }
7      }
8  }
```



Concolic Covering Middle Ground



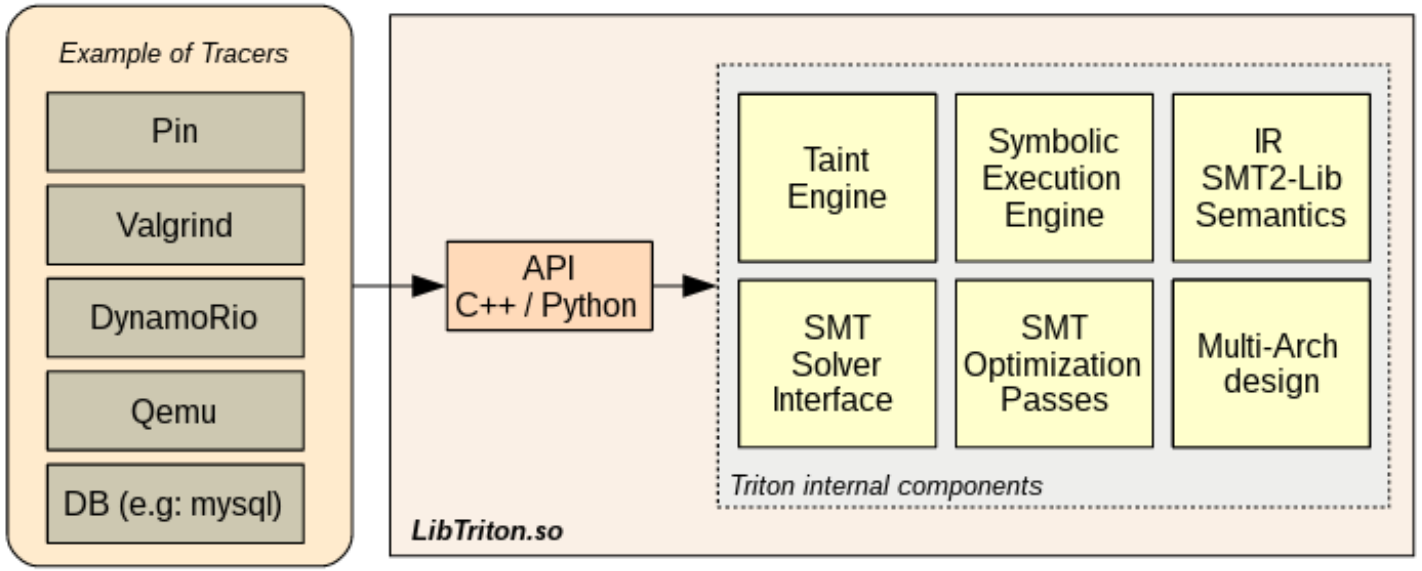
Symbolic execution engines

TRITON
Dynamic Binary Analysis

Triton

Triton is a dynamic binary analysis (DBA) framework. It provides internal components like a Dynamic Symbolic Execution (DSE) engine, a Taint Engine, AST representations of the x86 and the x86-64 instructions set semantics

https://triton.quarkslab.com/



Symbolic execution engines

Angr
Python framework for analyzing binaries. It combines both static and dynamic symbolic ("concolic") analysis, making it applicable to a variety of tasks.
http://angr.io/

If this is how you feel your in luck!

#4

Sometimes all you need is a little more ponce in your life.

Ponce

IDA's 2016 plugin contest winner

Built on the triton engine TRITON Dynamic Binary Analysis

https://github.com/illera88/Ponce

Ponce

IDA's 2016 plugin contest winner

Built on the triton engine

TRITON
Dynamic Binary Analysis

Supports both x86 and x64 binaries

https://github.com/illera88/Ponce

Cross platform, Windows, Linux and OSX natively

Ponce

Just a quick insight into ponce.

C2 Channel

Symbolize the data returned from malware. Find where the data hits the C2 switch and use the SMT solver to determine the control byte

Ponce

Just a quick insight into ponce.

C2 Channel

Symbolize the data returned from malware. Find where the data hits the C2 switch and use the SMT solver to determine the control byte

Finding exploits.

Run taint analysis on the data returned from a Recv() or a InternetReadfile() to see what blocks of data it touches. Platform to build exploits

Ponce

Just a quick insight into ponce.

C2 Channel
Symbolize the data returned from malware. Find where the data hits the C2 switch and use the SMT solver to determine the control byte

Finding exploits.
Run taint analysis on the data returned from a Recv() or a InternetReadfile() to see what blocks of data it touches. Platform to build exploits

Rapid analysis
Run and negate conditions on known buffers to find what other commands are supported.

Ponce

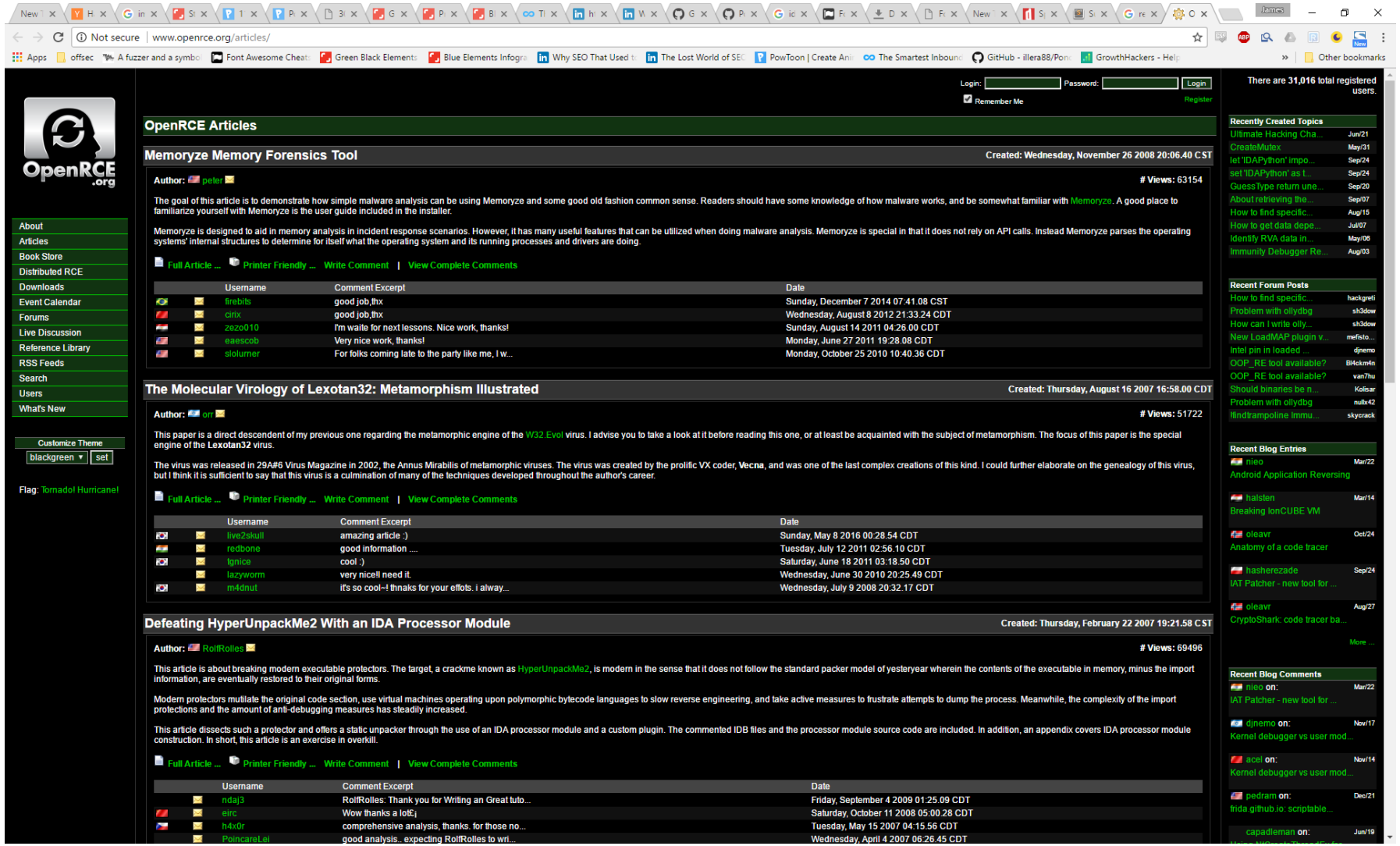
Simple to install (IDA 6.8-6.9):

Download the plugins and copy to your IDAPro plugins folder

```
$ cp Ponce_x64_IDA68_win.p64 <IDA_PRO>/plugins
```

```
$ cp Ponce_x64_IDA68_win.plw <IDA_PRO>/plugins
```

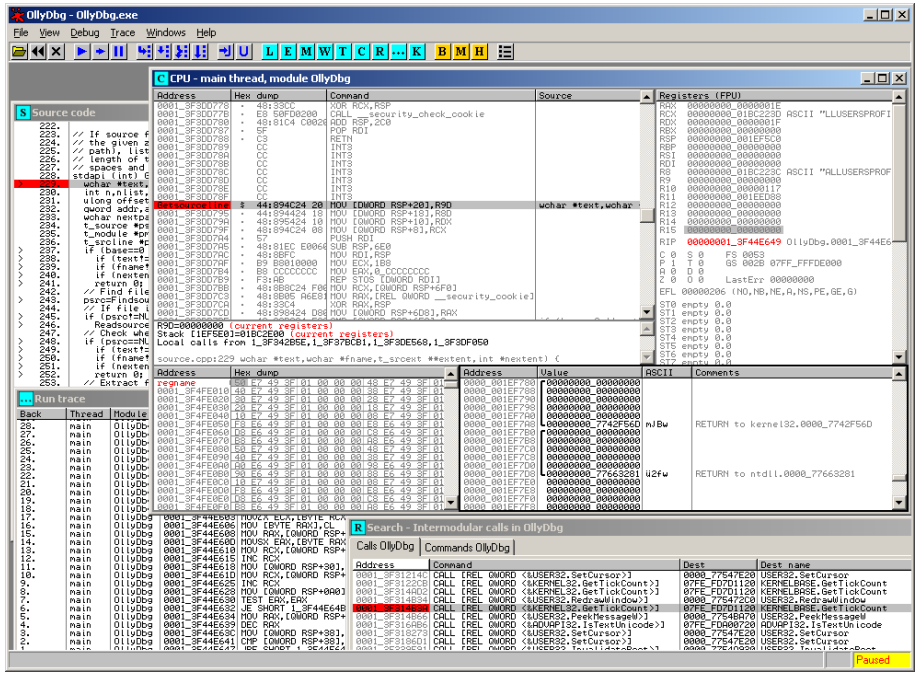
Ponce Example - Crackme



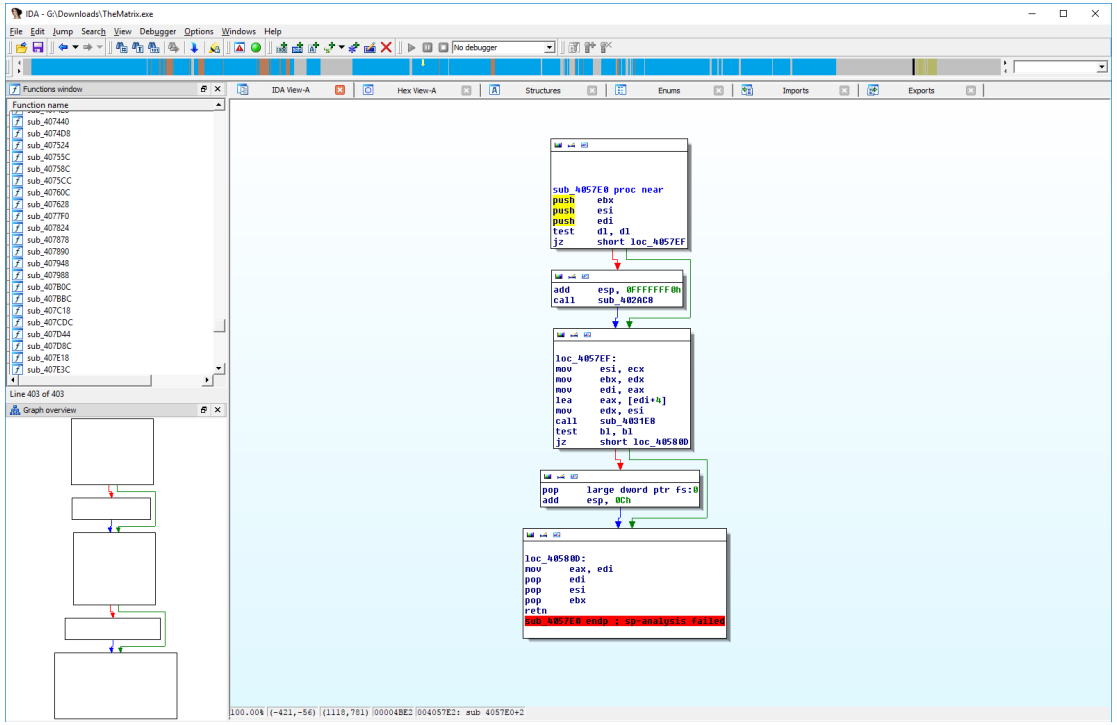
Ponce Example - Crackme

Ponce Example - Crackme

Normally load your two favorite tools



+



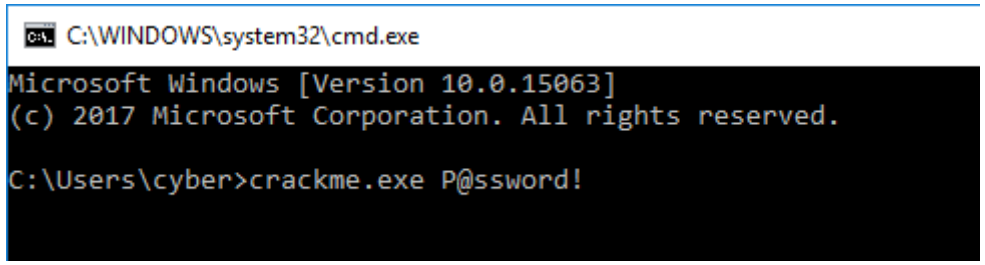
Ponce Example - Crackme

OR

Maybe use Ponce?

Ponce Example

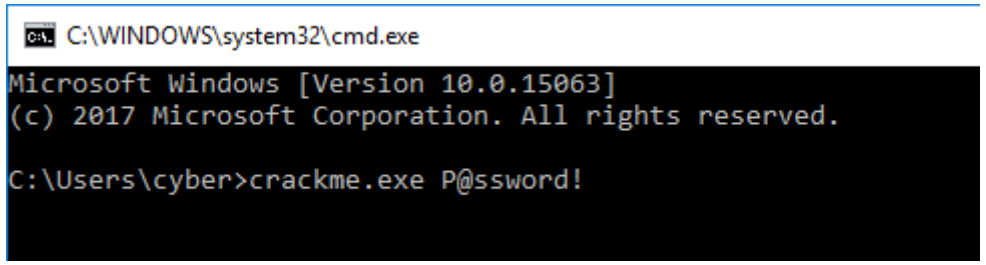
Standard crackme.exe




```c
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
        int i;
        int hash = 0xABCD;

        for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];
        return hash;
}

int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;

        ret = check(av[1]);
        if (ret == 0xad6d)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```

Ponce Example

Standard crackme.exe

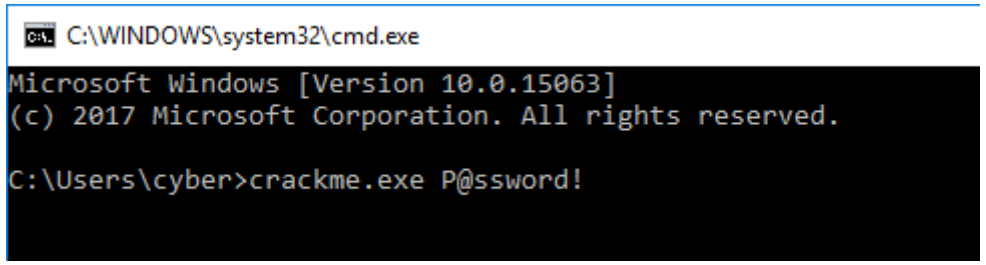


Hardcoded serial to check against

```c
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
        int i;
        int hash = 0xABCD;

        for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];
        return hash;
}

int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;

        ret = check(av[1]);
        if (ret == 0xad6d)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```

Ponce Example

Standard crackme.exe



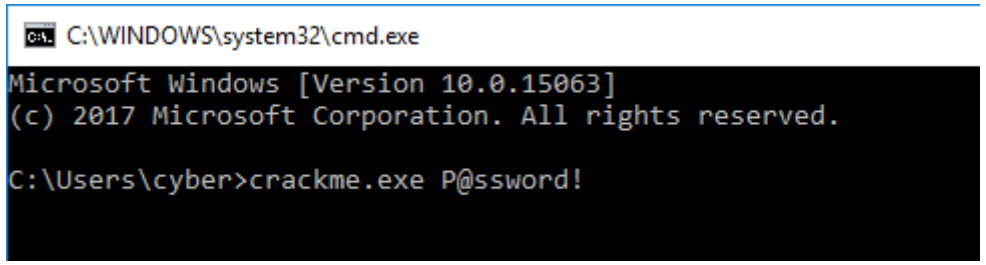
Hardcoded serial to check against

Takes the input and XOR's it against the serial.

```c
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
        int i;
        int hash = 0xABCD;

        for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];
        return hash;
}

int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;

        ret = check(av[1]);
        if (ret == 0xad6d)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```

Ponce Example

Standard crackme.exe



Hardcoded serial to check against

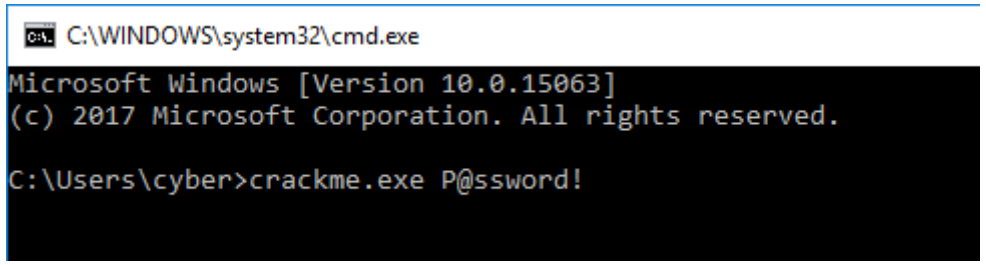
Takes the input and XOR's it against the serial.

If OxABCD + (serial XOR input) == OxAD6D Win!

```c
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
        int i;
        int hash = 0xABCD;

        for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];
        return hash;
}

int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;

        ret = check(av[1]);
        if (ret == 0xad6d)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```

Ponce Example

Standard crackme.exe



Hardcoded serial to check against

Takes the input and XOR's it against the serial.

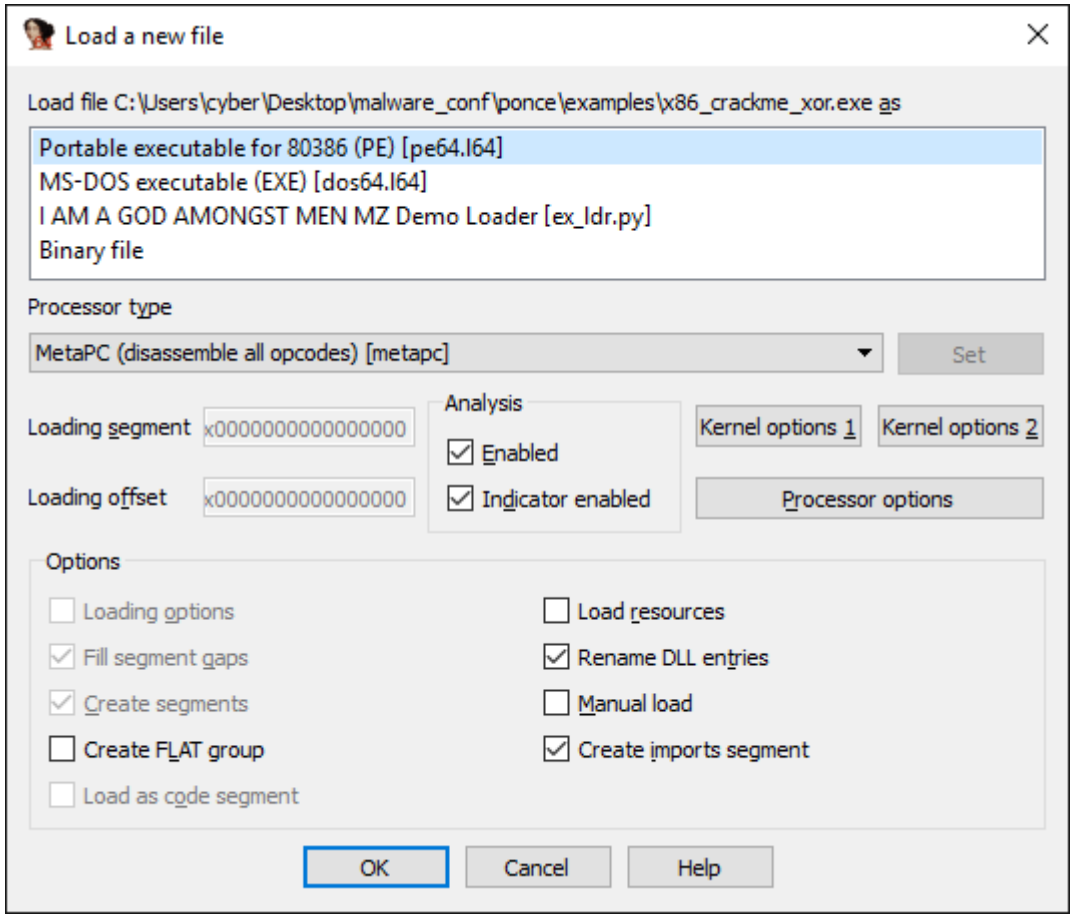
If 0xABCD + (serial XOR input) == 0xAD6D Win!

Even with the source what is the key?

```c
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
        int i;
        int hash = 0xABCD;

        for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];
        return hash;
}

int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;

        ret = check(av[1]);
        if (ret == 0xad6d)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```

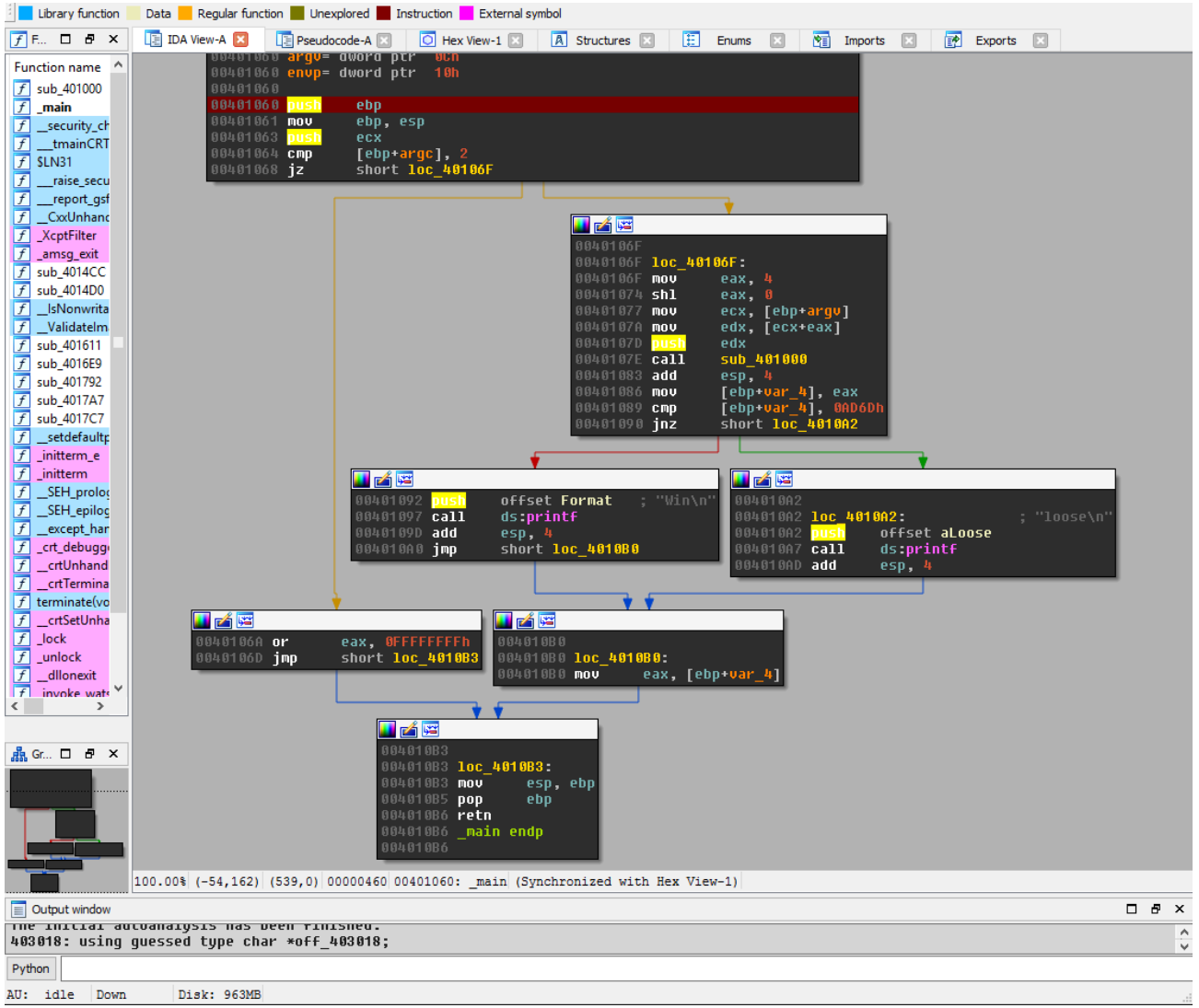
Ponce Example

Load the crackme.exe file



Ponce Example

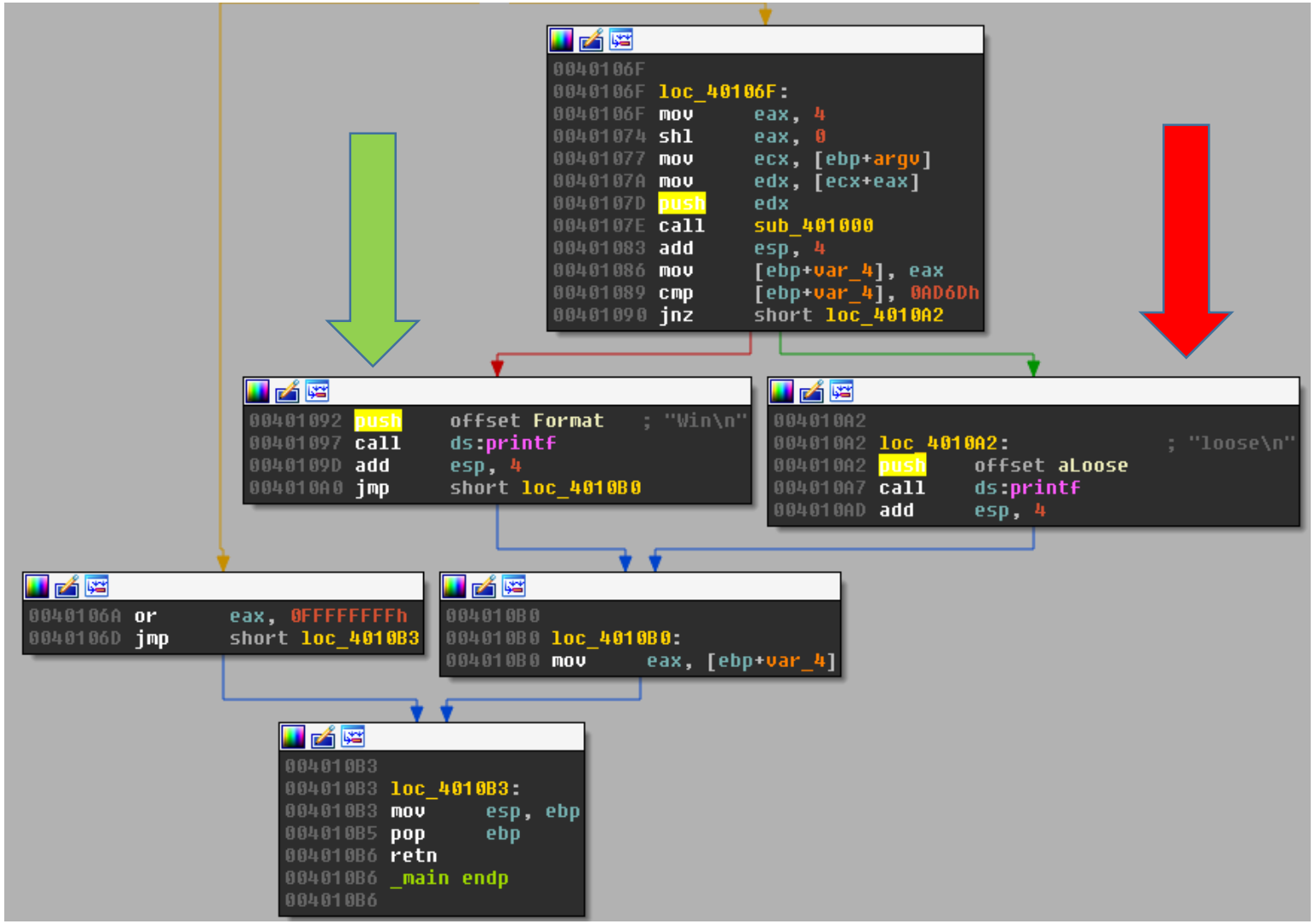
Load the crackme.exe file



Ponce Example

Win condition

Fail condition



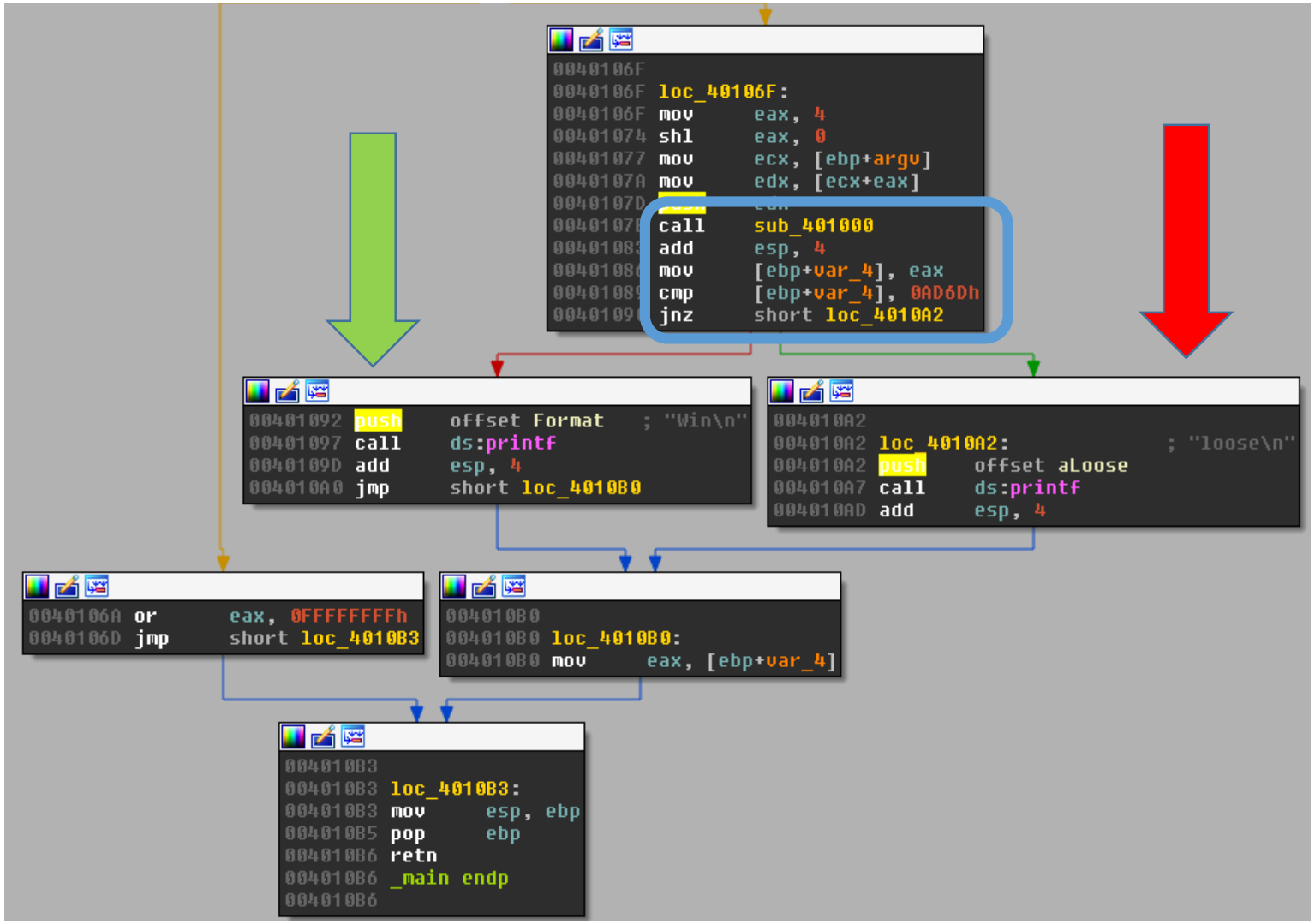
Ponce Example

Win condition

Fail condition

Call for check_function

return value into eax
jnz short loc_4010A2



Ponce Example

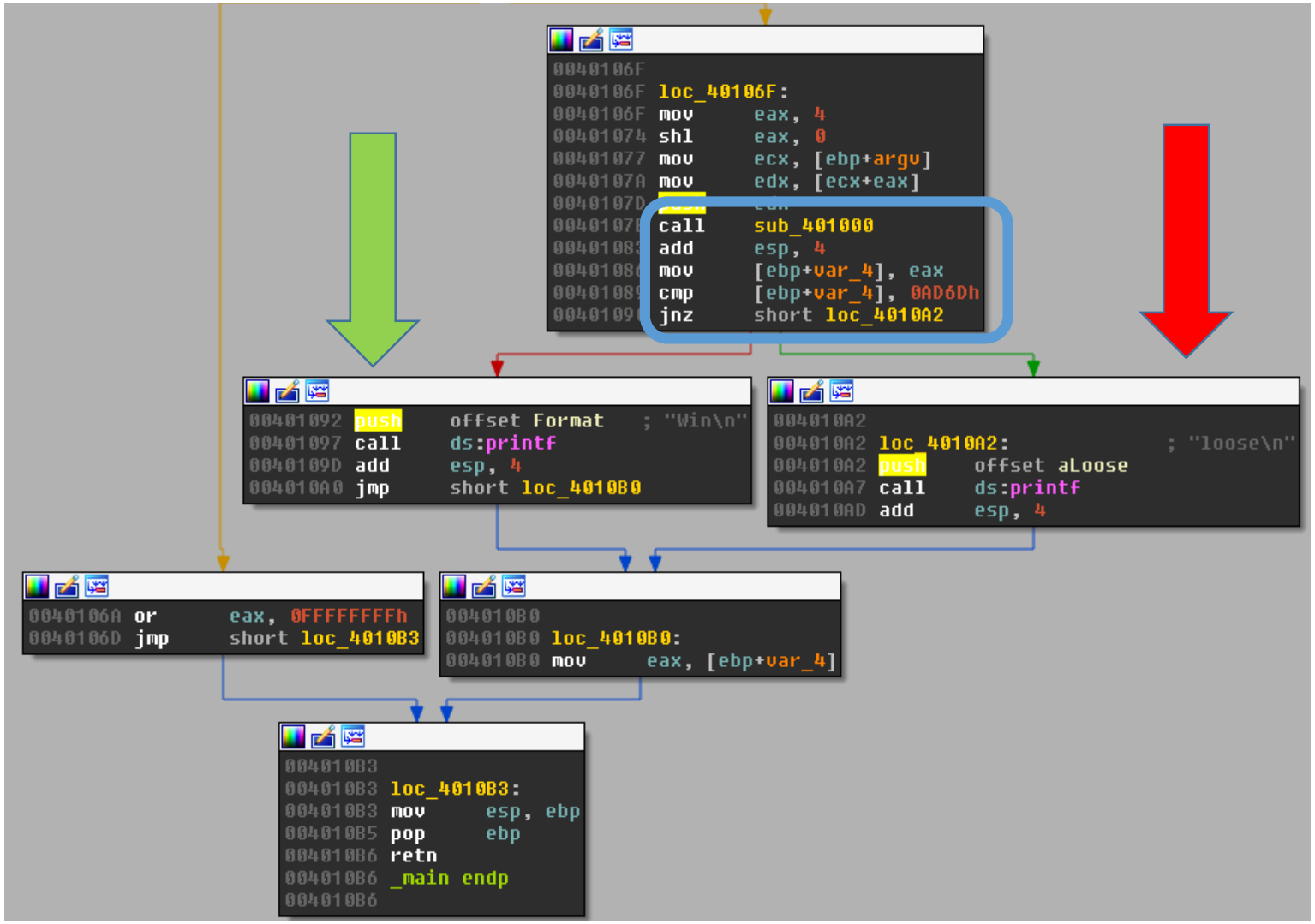
Win condition

Fail condition

Call for check_function

return value into eax
jnz short loc_4010A2

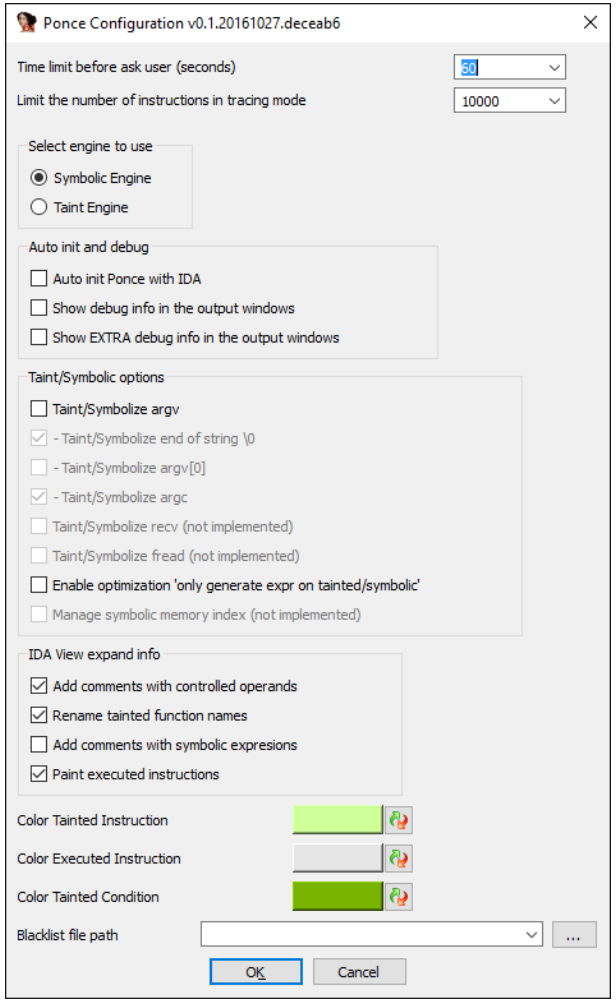
Set a breakpoint at program entry
Want to symbolize argv[1]



Ponce Example

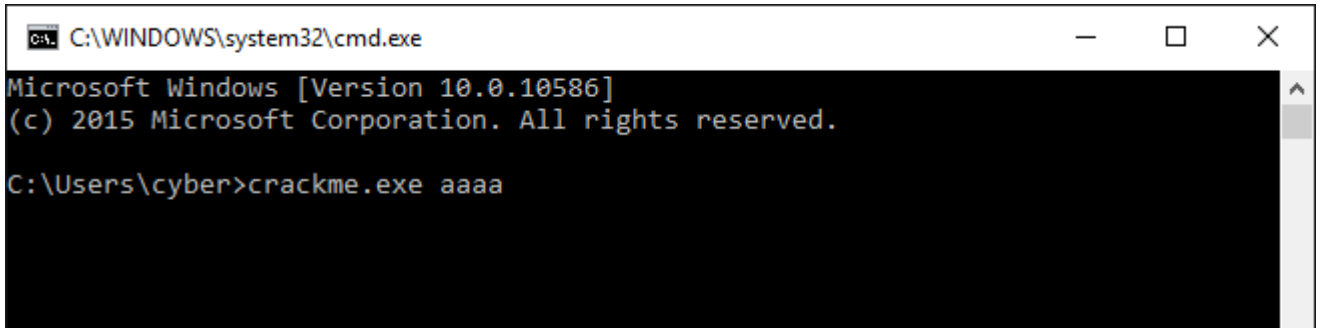
Start ponce plugin

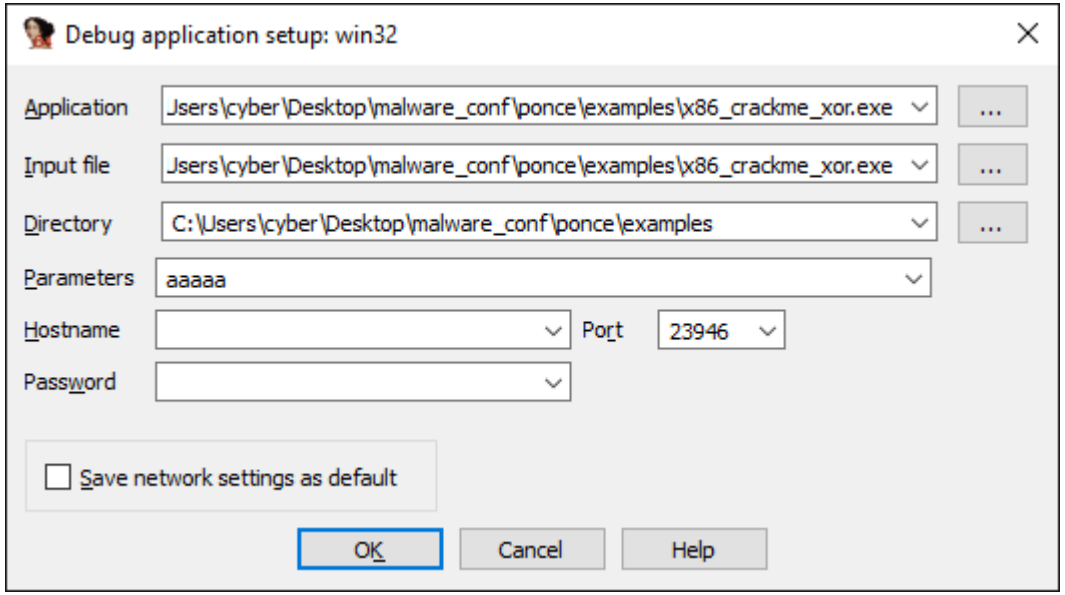
Set to symbolic engine



Ponce Example

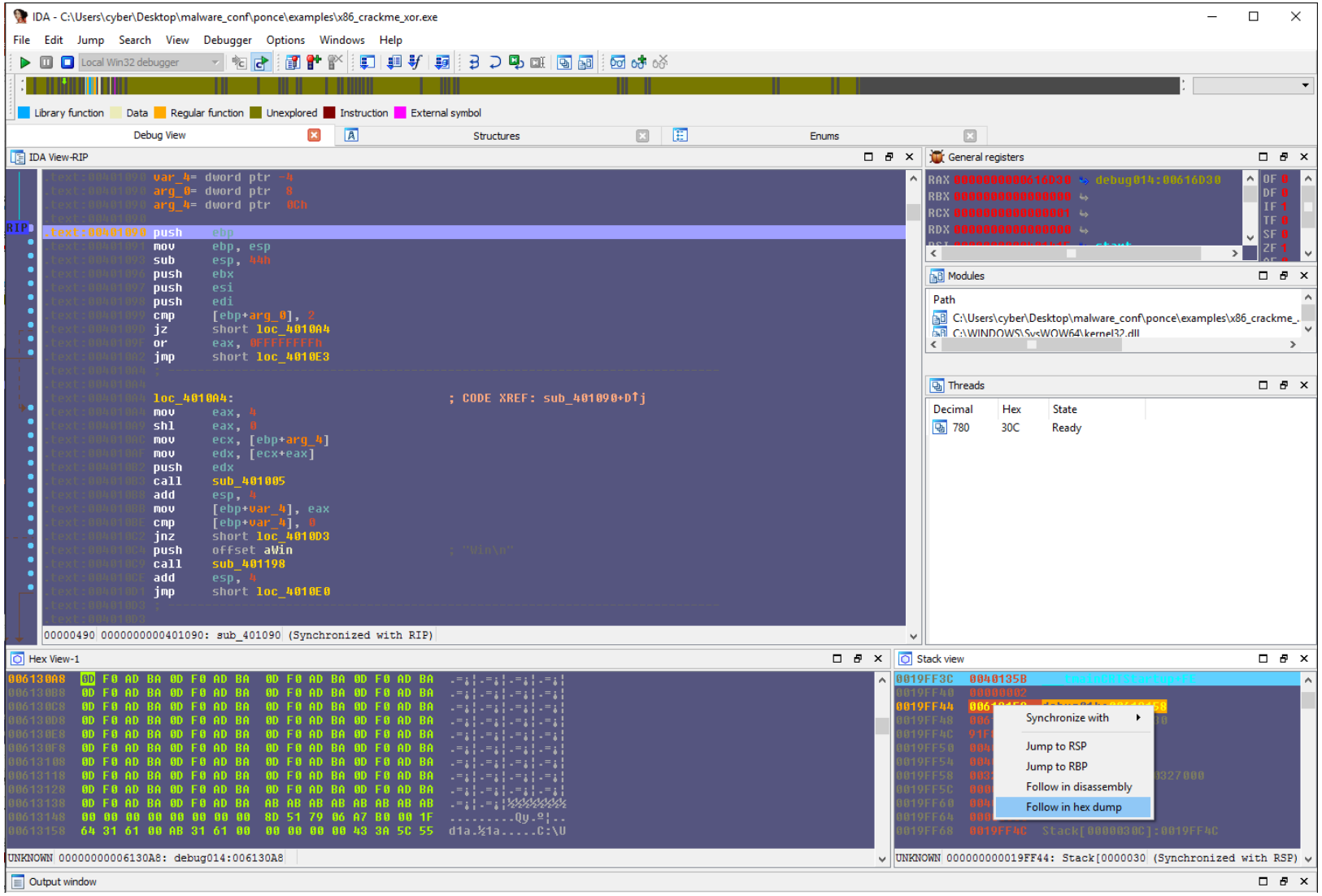
Set an input argument





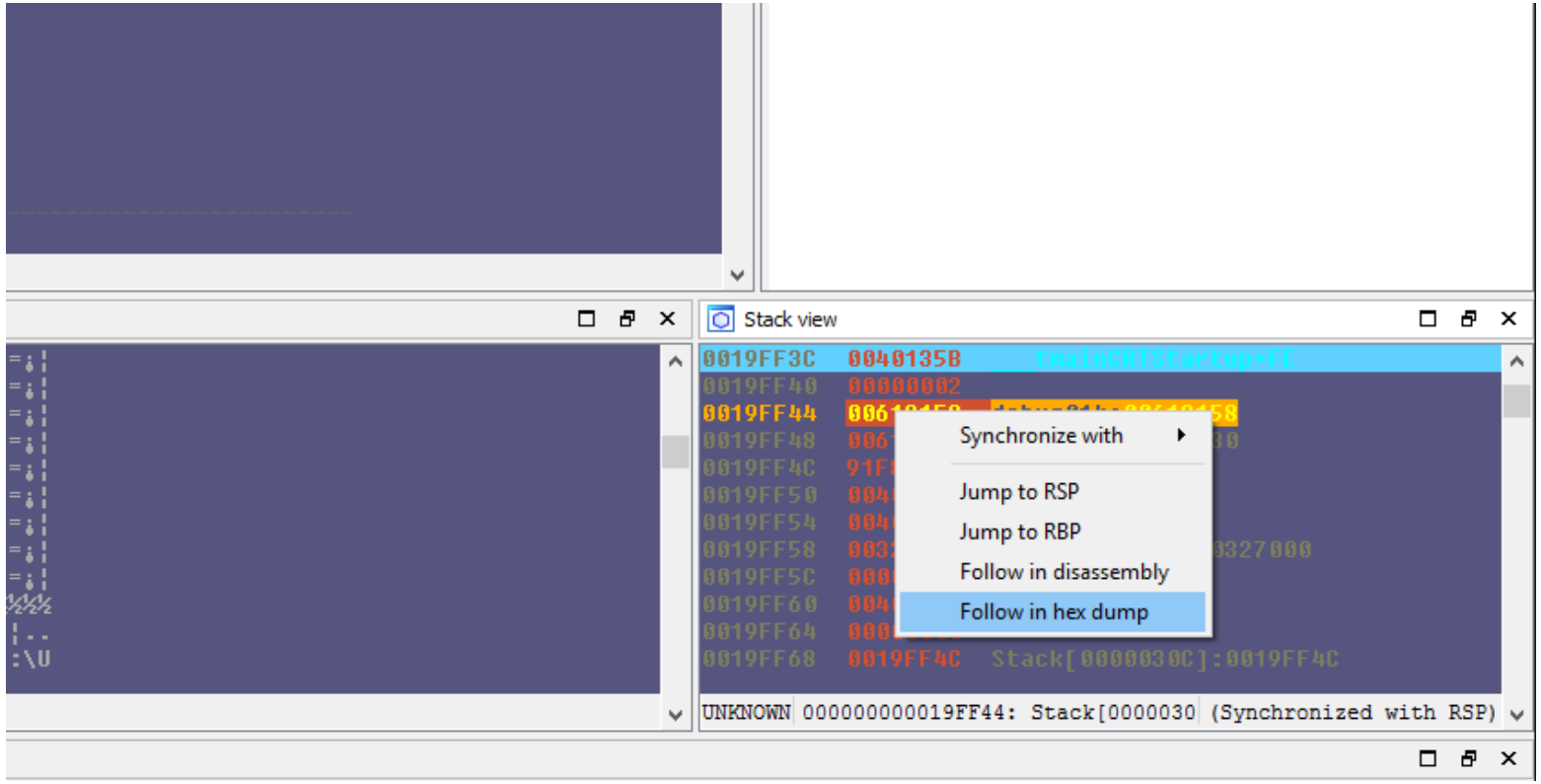
Ponce Example

Debugger started



Ponce Example

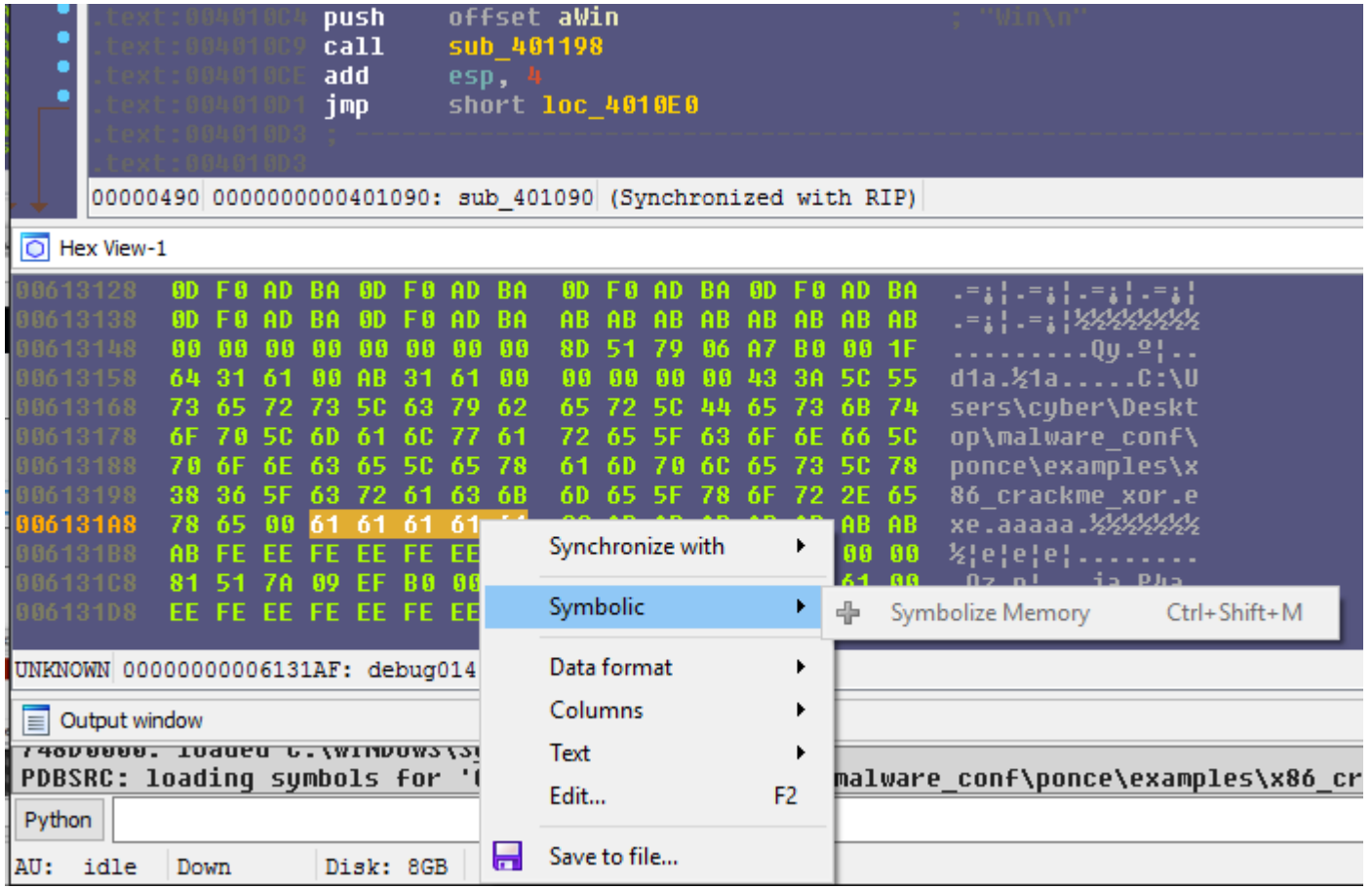
Jump in hex to the where argv[1] is passed in



Ponce Example

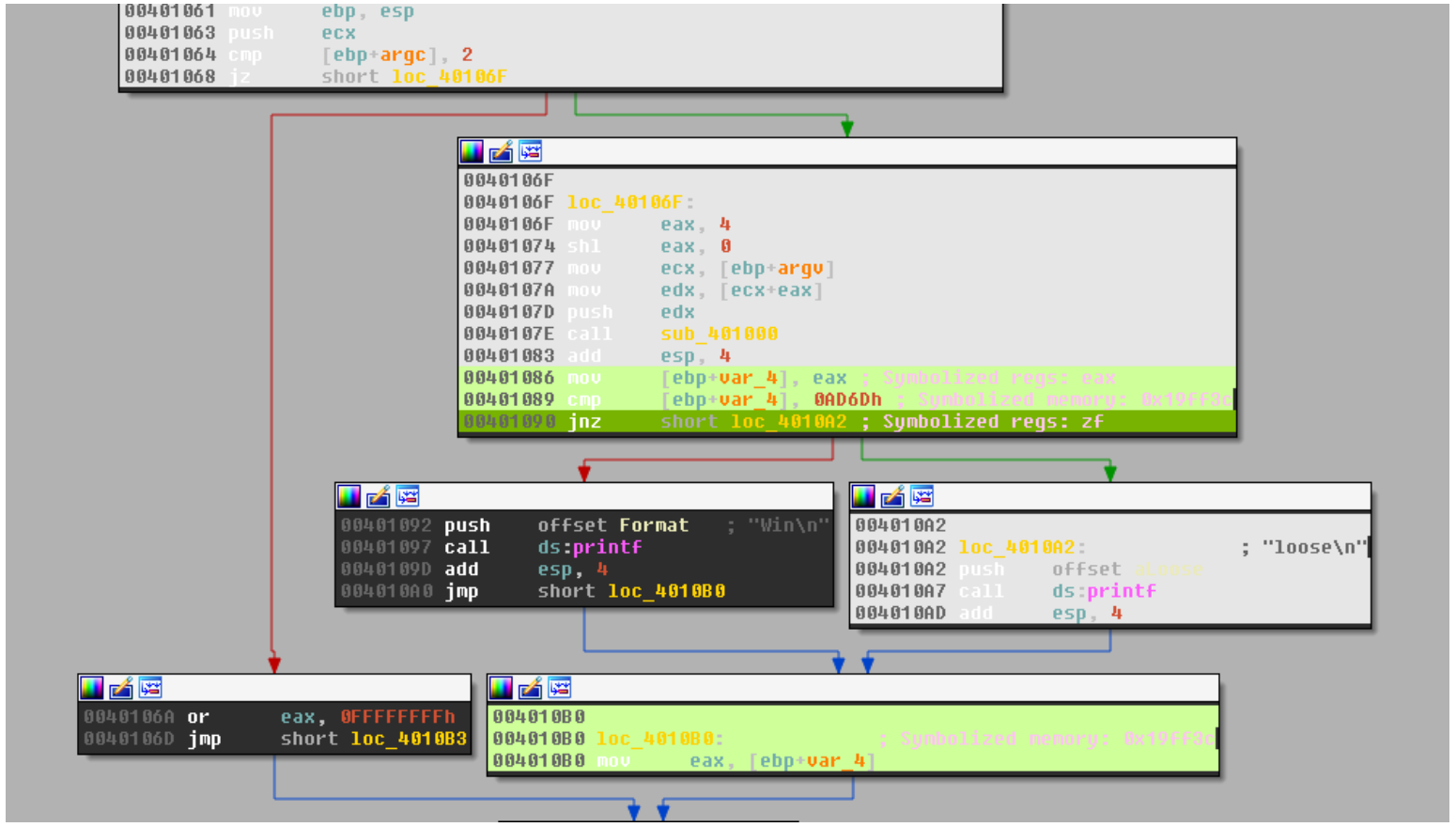
Symbolize the data that bytes that we care about

AAAAA



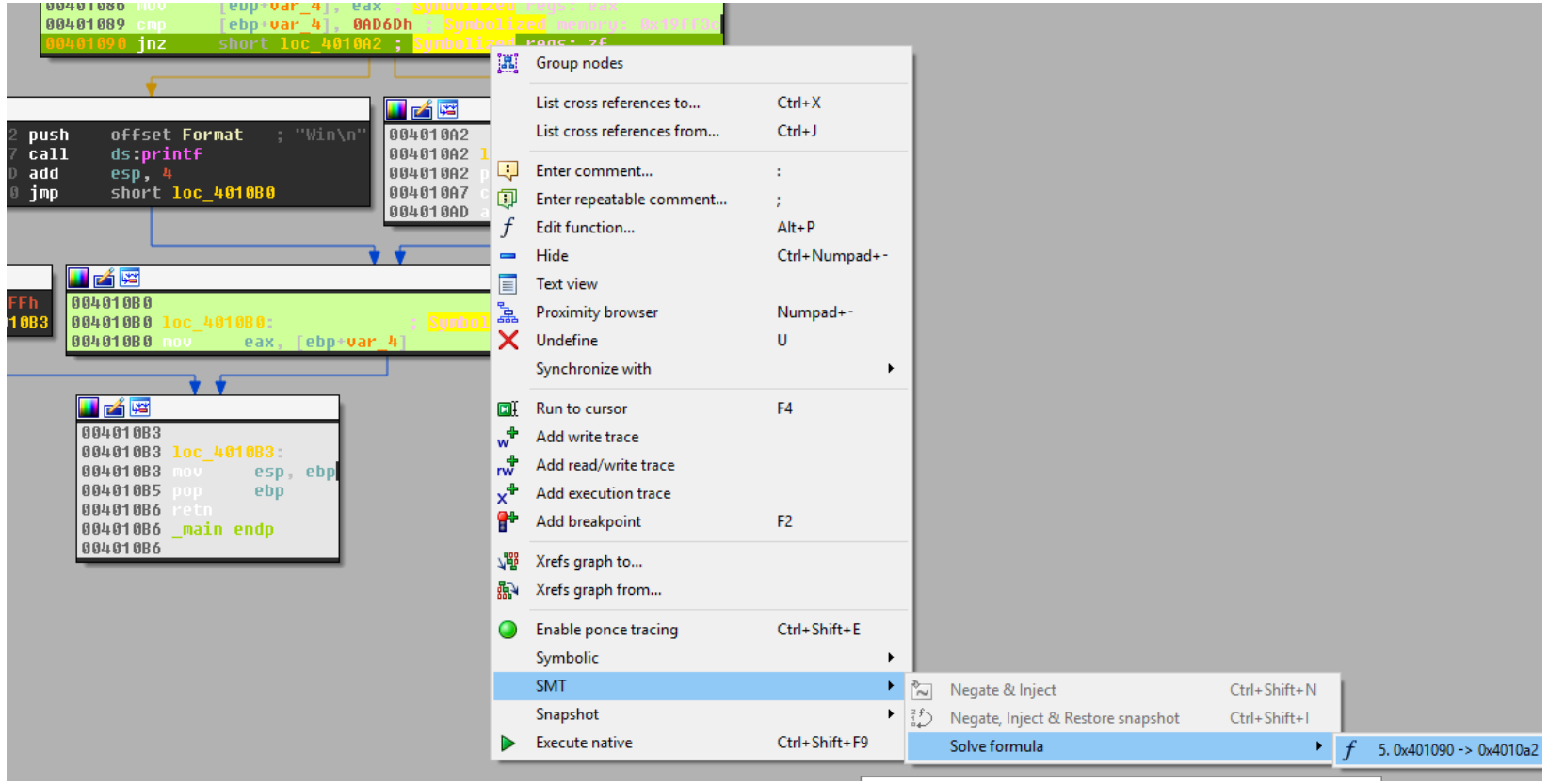
Ponce Example

Highlighted blocks of code that were executed



Ponce Example

Implement SMT solver to deduce what bytes we needed to obtain the end point



Ponce Example

We have the key!

SX7@Y

```
PDB: DIA interface version 9.0
Debugger: thread 9652 has exited (code 44399)
Debugger: thread 4736 has exited (code 44399)
Debugger: process has exited (exit code 44399)
[+] Solution found! Values:
 - SymVar_0 (argv[1][0]):0x53 (S)
 - SymVar_1 (argv[1][1]):0x58 (X)
 - SymVar_2 (argv[1][2]):0x37 (7)
 - SymVar_3 (argv[1][3]):0x40 (@)
 - SymVar_4 (argv[1][4]):0x59 (Y)
```

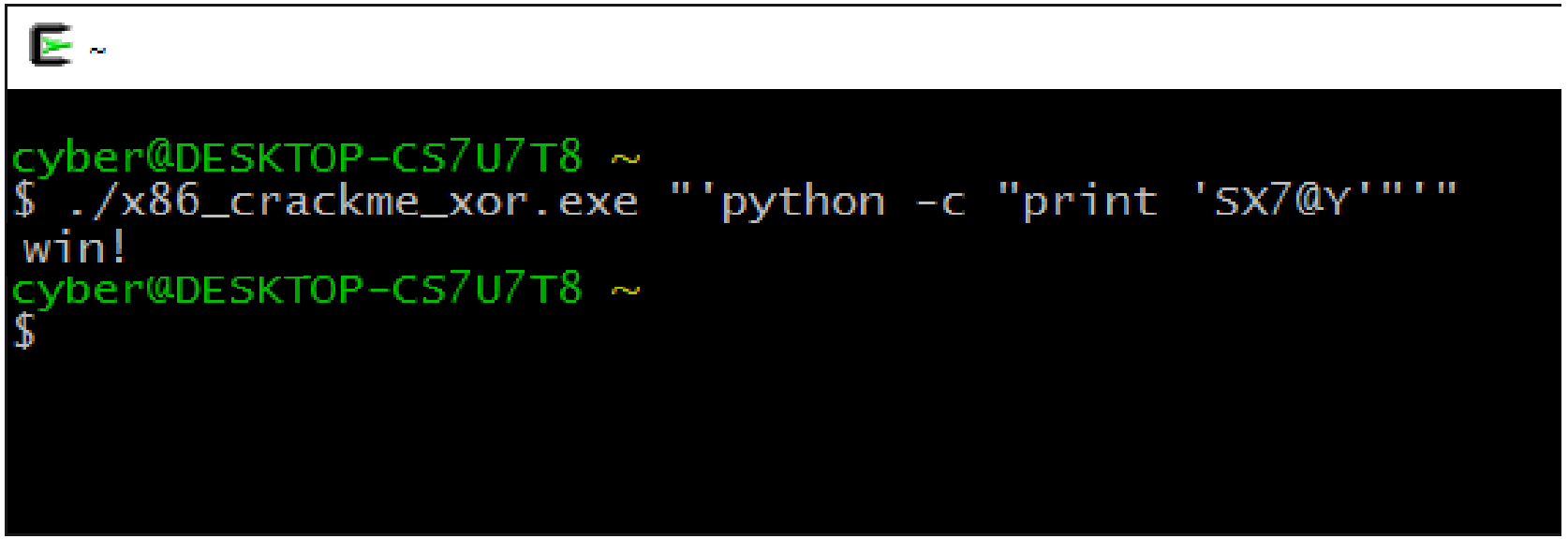
Python

AU: idle | Down | Disk: 3GB

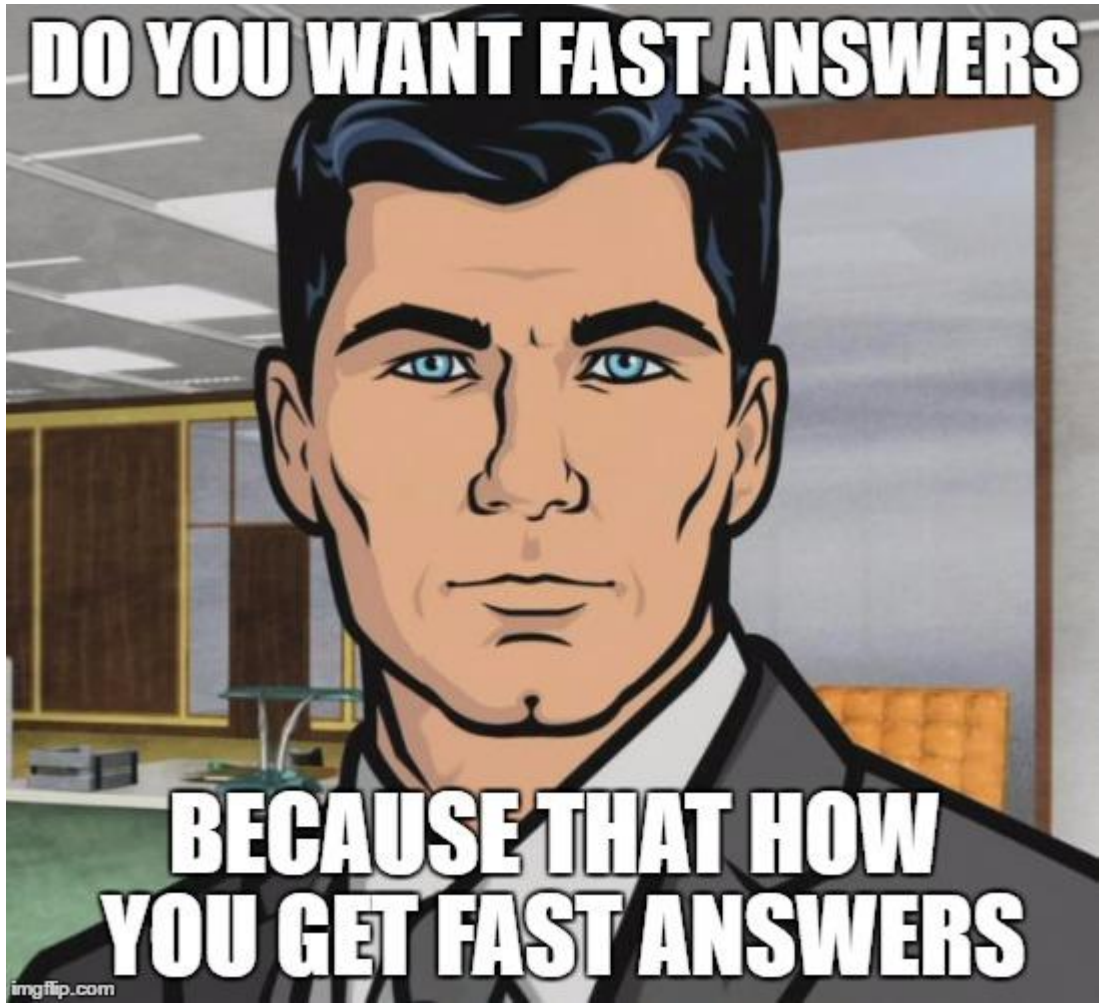
Ponce Example

Success!
We have the key

SX7@Y



Ponce


DO YOU WANT FAST ANSWERS
BECAUSE THAT HOW YOU GET FAST ANSWERS
imgflip.com

#5

Manticore is your friend...

Manticore

- A Plugin to IDA is great, but what if I wanted to do something more automated?

Manticore

A Plugin to IDA is great, but what if I wanted to do something more automated?

Maybe a code oriented approach?

Manticore

A Plugin to IDA is great, but what if I wanted to do something more automated?

Maybe a code oriented approach?

Not an auto-solve but a much better approach.

Manticore

Open source tool

Command line interface

Quickly generates use cases

```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ manticore
usage: manticore [-h] [--workspace WORKSPACE] [-v] [--profile]
                 [--buffer BUFFER] [--size SIZE] [--offset OFFSET]
                 [--maxsymb MAXSYMB] [--data DATA] [--env ENV]
                 [--policy POLICY] [--dumpafter DUMPAFTER]
                 [--maxstorage MAXSTORAGE] [--maxstates MAXSTATES]
                 [--procs PROCS] [--timeout TIMEOUT] [--replay REPLAY]
                 [--coverage COVERAGE] [--errorfile ERRORFILE]
                 [--context CONTEXT] [--assertions ASSERTIONS] [--names NAMES]
                 PROGRAM [PROGRAM ...]
manticore: error: too few arguments
```

Manticore

Open source tool

Command line interface

Quickly generates use cases

```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ manticore
usage: manticore [-h] [--workspace WORKSPACE] [-v] [--profile]
                 [--buffer BUFFER] [--size SIZE] [--offset OFFSET]
                 [--maxsymb MAXSYMB] [--data DATA] [--env ENV]
                 [--policy POLICY] [--dumpafter DUMPAFTER]
                 [--maxstorage MAXSTORAGE] [--maxstates MAXSTATES]
                 [--procs PROCS] [--timeout TIMEOUT] [--replay REPLAY]
                 [--coverage COVERAGE] [--errorfile ERRORFILE]
                 [--context CONTEXT] [--assertions ASSERTIONS] [--names NAMES]
                 PROGRAM [PROGRAM ...]
manticore: error: too few arguments
```

Python API

To answer more in depth questions

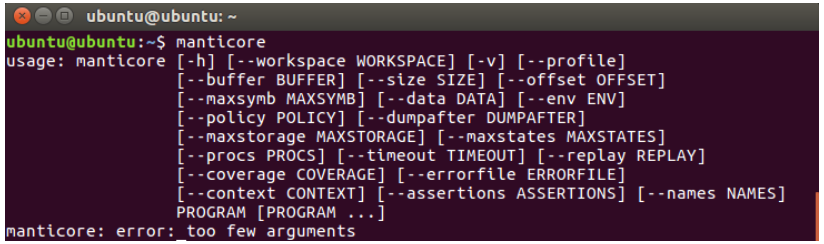
- How many times does a program execute this function?

Manticore

Open source tool

Command line interface

Quickly generates use cases



Python API

To answer more in depth questions

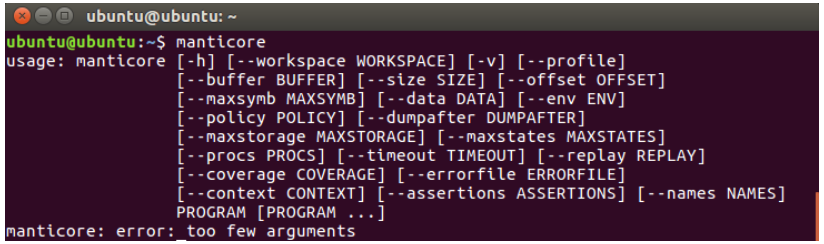
- How many times does a program execute this function?
- What input causes execution of this block of code?

Manticore

Open source tool

Command line interface

Quickly generates use cases



Python API

To answer more in depth questions

- How many times does a program execute this function?
- What input causes execution of this block of code?
- At point X in execution, is it possible for variable Y to be a specified value?

Manticore

Pythonic symbolic programming

Hook addresses and add constraints

```
from manticore import Manticore

hook_pc = 0x400ca0

m = Manticore('./path/to/binary')

@m.hook(hook_pc)
def hook(state):
    cpu = state.cpu
    print 'eax', cpu.EAX
    print cpu.read_int(cpu.SP)

    m.terminate()  # tell Manticore to stop

m.run()
```

Manticore

Pythonic symbolic programming

Hook addresses and add constraints

Read/write into registers/memory at point of execution

```
from manticore import Manticore

hook_pc = 0x400ca0

m = Manticore('./path/to/binary')

@m.hook(hook_pc)
def hook(state):
    cpu = state.cpu
    print 'eax', cpu.EAX
    print cpu.read_int(cpu.SP)

    m.terminate()  # tell Manticore to stop

m.run()
```

Manticore

Pythonic symbolic programming

Hook addresses and add constraints

Read/write into registers/memory at point of execution

Use SMT solver in Z3 to do symbolically solve deep assembly structures.

```
from manticore import Manticore

hook_pc = 0x400ca0

m = Manticore('./path/to/binary')

@m.hook(hook_pc)
def hook(state):
    cpu = state.cpu
    print 'eax', cpu.EAX
    print cpu.read_int(cpu.SP)

    m.terminate()  # tell Manticore to stop

m.run()
```

Manticore

Simple to install (Ubuntu 16.04):

Of course working in a virtual environment is recommended

```
$ pip install manticore
```

```
$ apt install z3
```

Manticore

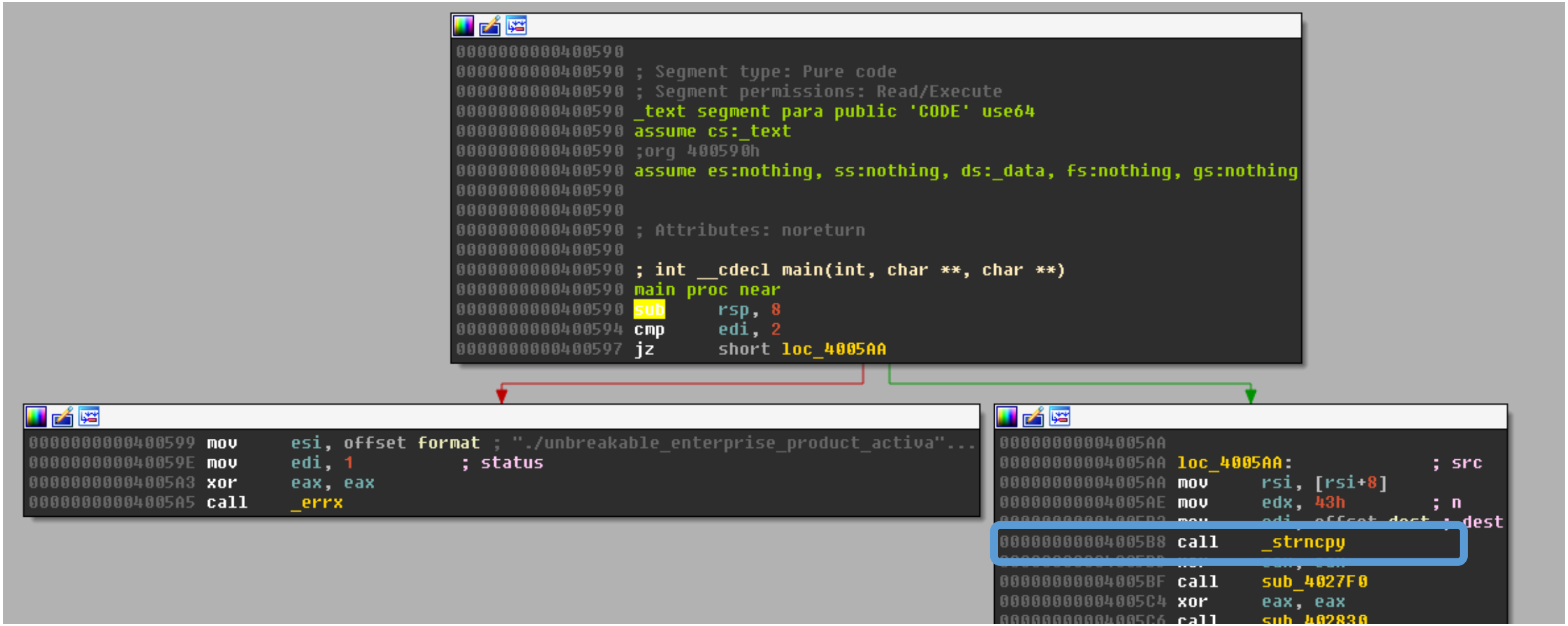
Google CTF 2016

- **Cory Duplantis**

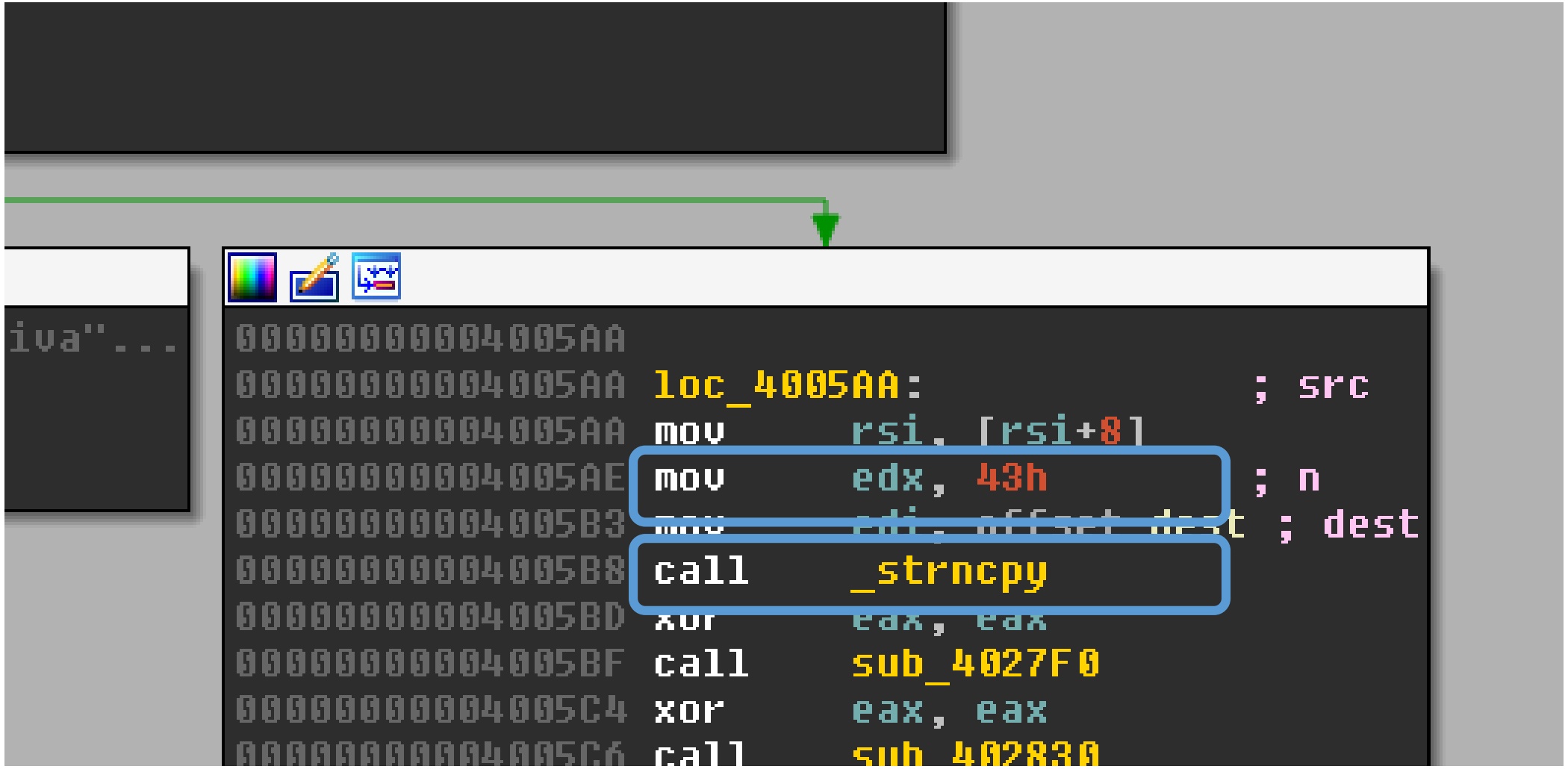
Reverse engineering challenge

Bust out the rush tape!

Manticore – Dive in



Manticore – Dive in



Manticore – Dive in

```python
input_addr = 0x6042c0
num_bytes = 0x43

# Entry point
@m.hook(0x400729)
def hook(state):
    """ CAUTION: HACK """
    """ From entry point, jump directly to code performing check """

    # Create a symbolic buffer for our input
    buffer = state.new_symbolic_buffer(0x43)

    # We are given in the challenge that the flag begins with CTF{
    # So we can apply constraints to ensure that is true
    state.constrain(buffer[0] == ord('C'))
    state.constrain(buffer[1] == ord('T'))
    state.constrain(buffer[2] == ord('F'))
    state.constrain(buffer[3] == ord('{'))

    # Store our symbolic input in the global buffer read by the check
    state.cpu.write_bytes(input_addr, buffer)

    # By default, `hook` doesn't patch the instruction, so we hop over
    # the hooked strncpy and execute the next instruction
    state.cpu.EIP = 0x4005bd
```

Manticore – Dive in

Set up a symbolic buffer 0x43 in size

```
input_addr = 0x6042c0
num_bytes = 0x43

# Entry point
@m.hook(0x400729)
def hook(state):
    """ CAUTION: HACK """
    """ From entry point, jump directly to code performing check """

    # Create a symbolic buffer for our input
    buffer = state.new_symbolic_buffer(0x43)

    # We are given in the challenge that the flag begins with CTF{
    # So we can apply constraints to ensure that is true
    state.constrain(buffer[0] == ord('C'))
    state.constrain(buffer[1] == ord('T'))
    state.constrain(buffer[2] == ord('F'))
    state.constrain(buffer[3] == ord('{'))

    # Store our symbolic input in the global buffer read by the check
    state.cpu.write_bytes(input_addr, buffer)

    # By default, `hook` doesn't patch the instruction, so we hop over
    # the hooked strncpy and execute the next instruction
    state.cpu.EIP = 0x4005bd
```

Manticore – Dive in

Set up a symbolic buffer 0x43 in size

In this challenge the string starts with: **CTF{**

```
input_addr = 0x6042c0
num_bytes = 0x43

# Entry point
@m.hook(0x400729)
def hook(state):
    """ CAUTION: HACK """
    """ From entry point, jump directly to code performing check """

    # Create a symbolic buffer for our input
    buffer = state.new_symbolic_buffer(0x43)

    # We are given in the challenge that the flag begins with CTF{
    # So we can apply constraints to ensure that is true
    state.constrain(buffer[0] == ord('C'))
    state.constrain(buffer[1] == ord('T'))
    state.constrain(buffer[2] == ord('F'))
    state.constrain(buffer[3] == ord('{'))

    # Store our symbolic input in the global buffer read by the check
    state.cpu.write_bytes(input_addr, buffer)

    # By default, `hook` doesn't patch the instruction, so we hop over
    # the hooked strncpy and execute the next instruction
    state.cpu.EIP = 0x4005bd
```

Manticore – Dive in

Set up a symbolic buffer 0x43 in size

In this challenge the string starts with: **CTF{**

Write data to the buffer

```
input_addr = 0x6042c0
num_bytes = 0x43

# Entry point
@m.hook(0x400729)
def hook(state):
    """ CAUTION: HACK """
    """ From entry point, jump directly to code performing check """

    # Create a symbolic buffer for our input
    buffer = state.new_symbolic_buffer(0x43)

    # We are given in the challenge that the flag begins with CTF{
    # So we can apply constraints to ensure that is true
    state.constrain(buffer[0] == ord('C'))
    state.constrain(buffer[1] == ord('T'))
    state.constrain(buffer[2] == ord('F'))
    state.constrain(buffer[3] == ord('{'))

    # Store our symbolic input in the global buffer read by the check
    state.cpu.write_bytes(input_addr, buffer)

    # By default, `hook` doesn't patch the instruction, so we hop over
    # the hooked strncpy and execute the next instruction
    state.cpu.EIP = 0x4005bd
```

Manticore – Dive in

Set up a symbolic buffer 0x43 in size

In this challenge the string starts with: **CTF{**

Write data to the buffer

Hook does not patch instruction

Set EIP to after the call

```
input_addr = 0x6042c0
num_bytes = 0x43

# Entry point
@m.hook(0x400729)
def hook(state):
    """ CAUTION: HACK """
    """ From entry point, jump directly to code performing check """

    # Create a symbolic buffer for our input
    buffer = state.new_symbolic_buffer(0x43)

    # We are given in the challenge that the flag begins with CTF{
    # So we can apply constraints to ensure that is true
    state.constrain(buffer[0] == ord('C'))
    state.constrain(buffer[1] == ord('T'))
    state.constrain(buffer[2] == ord('F'))
    state.constrain(buffer[3] == ord('{'))

    # Store our symbolic input in the global buffer read by the check
    state.cpu.write_bytes(input_addr, buffer)

    # By default, `hook` doesn't patch the instruction, so we hop over
    # the hooked strncpy and execute the next instruction
    state.cpu.EIP = 0x4005bd
```

Manticore – Dive in

Function to remove failed paths – auto negate and inject

```
# Failure case
@m.hook(0x400850)
def hook(state):
    """ Stop executing paths that reach the `failure` function"""
    print("Invalid path.. abandoning")
    state.abandon()

# Success case
@m.hook(0x400724)
def hook(state):
    print("Hit the final state.. solving")

    state._solver._command = 'z3 -t:240000 -smt2 -in' # Hack around tr
    buffer = state.cpu.read_bytes(input_addr, num_bytes)
    res = ''.join(chr(state.solve_one(x)) for x in buffer)
    print(res) # CTF{0The1Quick2Brown3Fox4Jumped5Over6The7Lazy8Fox9}

    # We found the flag, no need to continue execution
    m.terminate()

m.should_profile = True
m.run(procs=10)
```

Manticore – Dive in

Function to remove failed paths – auto negate and inject

Define end point in execution

- Use z3 to solve against the input address we defined

```
# Failure case
@m.hook(0x400850)
def hook(state):
    """ Stop executing paths that reach the `failure` function"""
    print("Invalid path.. abandoning")
    state.abandon()

# Success case
@m.hook(0x400724)
def hook(state):
    print("Hit the final state.. solving")

    state._solver._command = 'z3 -t:240000 -smt2 -in' # Hack around tr
    buffer = state.cpu.read_bytes(input_addr, num_bytes)
    res = ''.join(chr(state.solve_one(x)) for x in buffer)
    print(res) # CTF{0The1Quick2Brown3Fox4Jumped5Over6The7Lazy8Fox9

    # We found the flag, no need to continue execution
    m.terminate()

m.should_profile = True
m.run(procs=10)
```

Manticore – Dive in

```
Invalid path.. abandoning
Invalid path.. abandoning
Invalid path.. abandoning
Invalid path.. abandoning
Invalid path.. abandoning
Invalid path.. abandoning
Hit the final state.. solving
CTF{0The1Quick2Brown3Fox4Jumped5Over6The7Lazy8Fox9}eeeeeeeeeeeeeeeee
ubuntu@ubuntu:~/Downloads$
```

Manticore – Dive in

#6

Okay, so maybe it won't solve everything but it will help.

Where are we going?

Symbolic of the future?

Traditional tools have served us really well and will still be the go to solve all of the in depth questions we have

The future of reverse engineering?

Increasing pace of malware, new outbreaks and the need to get answers fast. We can’t afford the time that attackers provide.

SYMBOLIC OF THE FUTURE

Thanks for your time!

Questions?

(Also it's lunch time and I'm hungry)