Flare-On 11 Challenge 9: Serpentine

By Mustafa Nasser (@d35ha)

Overview

The file `serpentine.exe` is a 64-bit Windows executable. When executed, it prompts the user asking for a single command line parameter called **key**. The challenge is a crack-me, **key** is the unique part of the flag and the target is to analyze the file, understand it and determine the **key**.

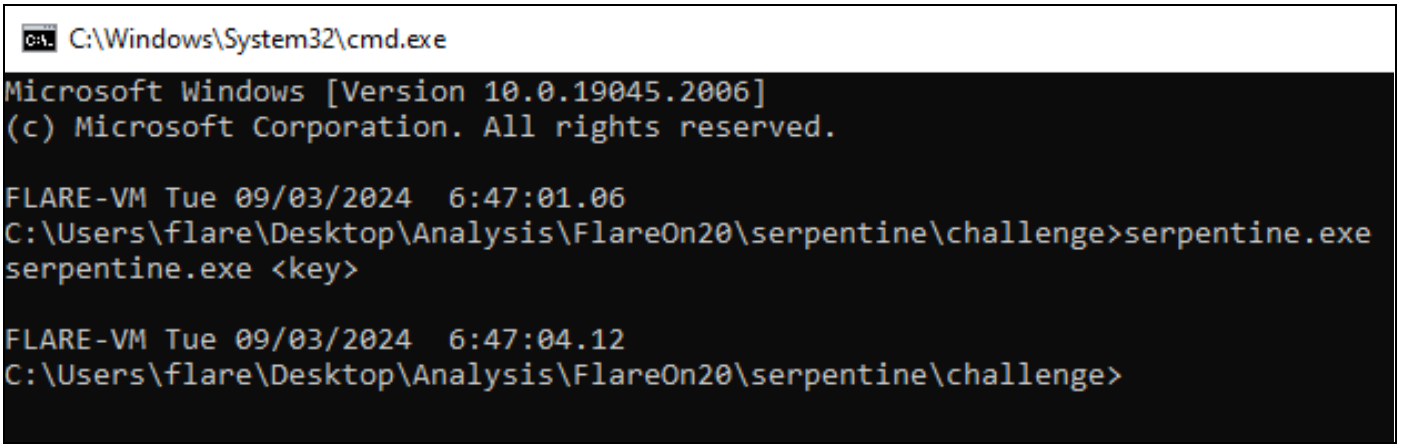

Figure 1: Program initial execution

Basic Analysis

The **key** is checked to be 32 characters long, which can be verified by inputting it with different lengths as shown in Figures 2 and 3 below.

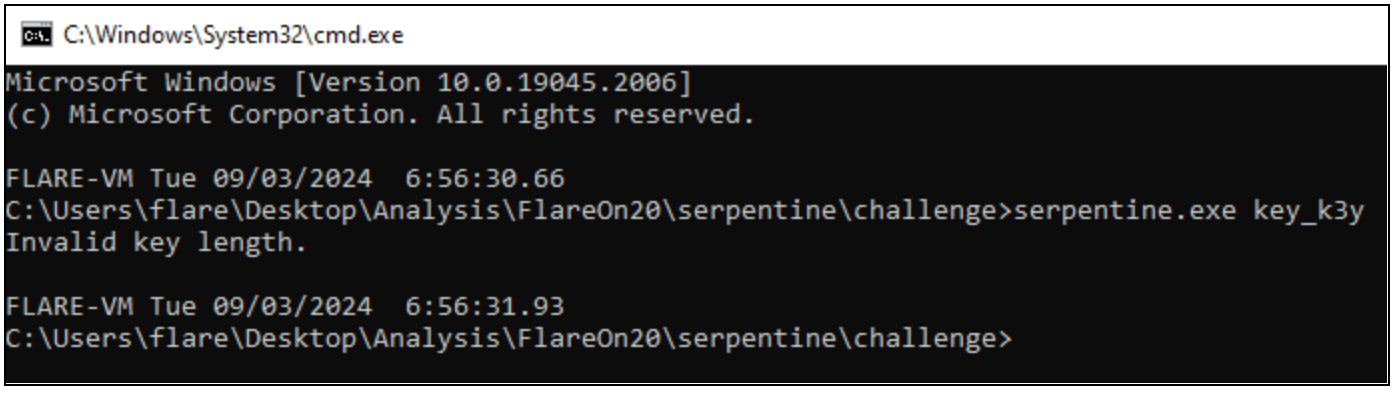

Figure 2: Trying with an invalid length

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.2006]
(c) Microsoft Corporation. All rights reserved.

FLARE-VM Tue 09/03/2024  6:59:49.22
C:\Users\flare\Desktop\Analysis\FlareOn20\serpentine\challenge>serpentine.exe key_k3y_ke7_k37_key_k3y_ke7_k37s
Wrong key

FLARE-VM Tue 09/03/2024  7:00:31.44
C:\Users\flare\Desktop\Analysis\FlareOn20\serpentine\challenge>
```

Figure 3: Trying with the right length gives a different output

By opening the file in IDA we can verify the length check as shown in the IDA screenshot below in Figure 4.

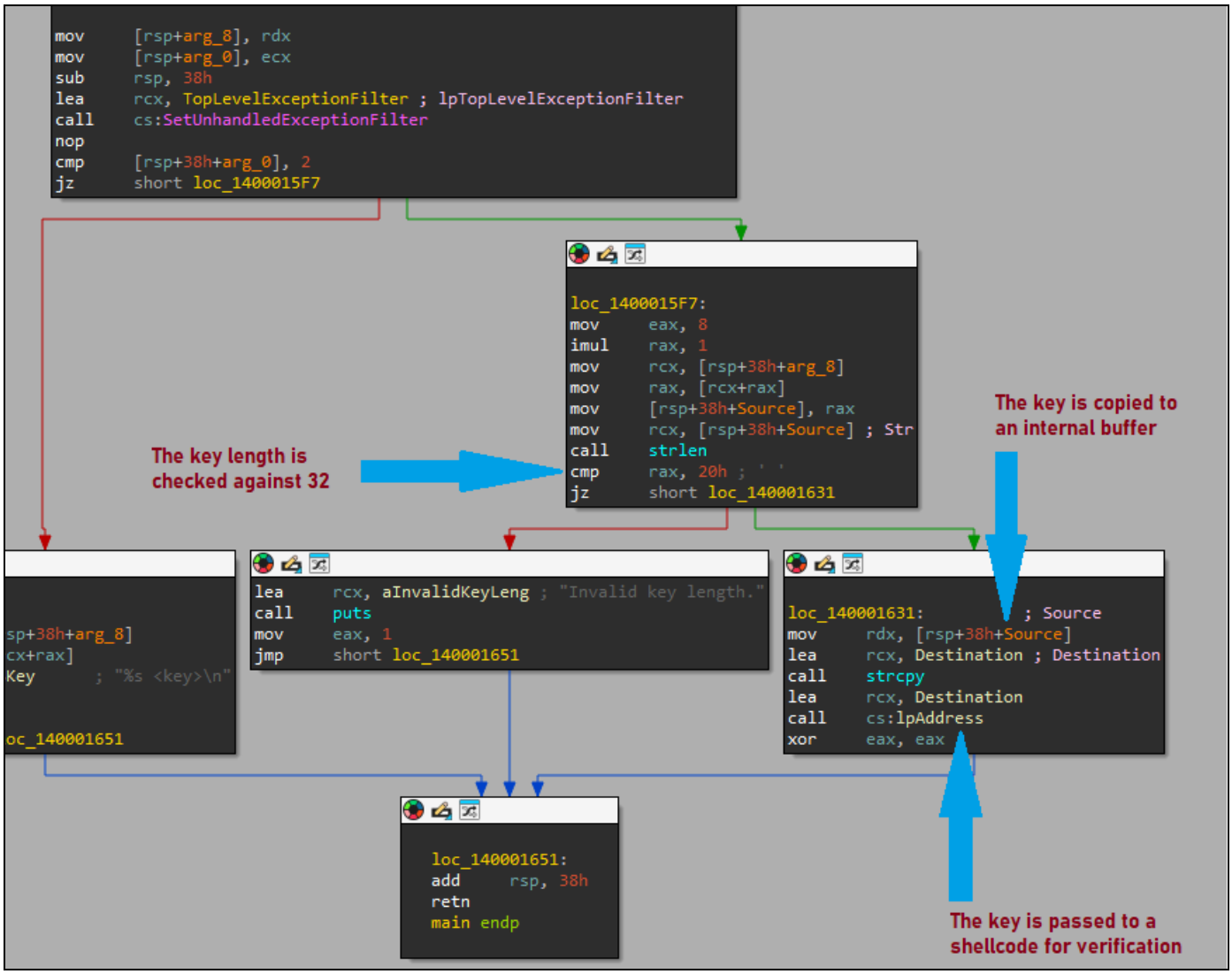

Figure 4: The main function in IDA

Also, Figure 4 shows that the **key** is checked by passing it to a shellcode and the shellcode address is saved at `0x14089B8E0` (labeled as `lpAddress` in this Figure). By checking the references to that location, apparently the file allocates the shellcode and writes it within a TLS callback function.

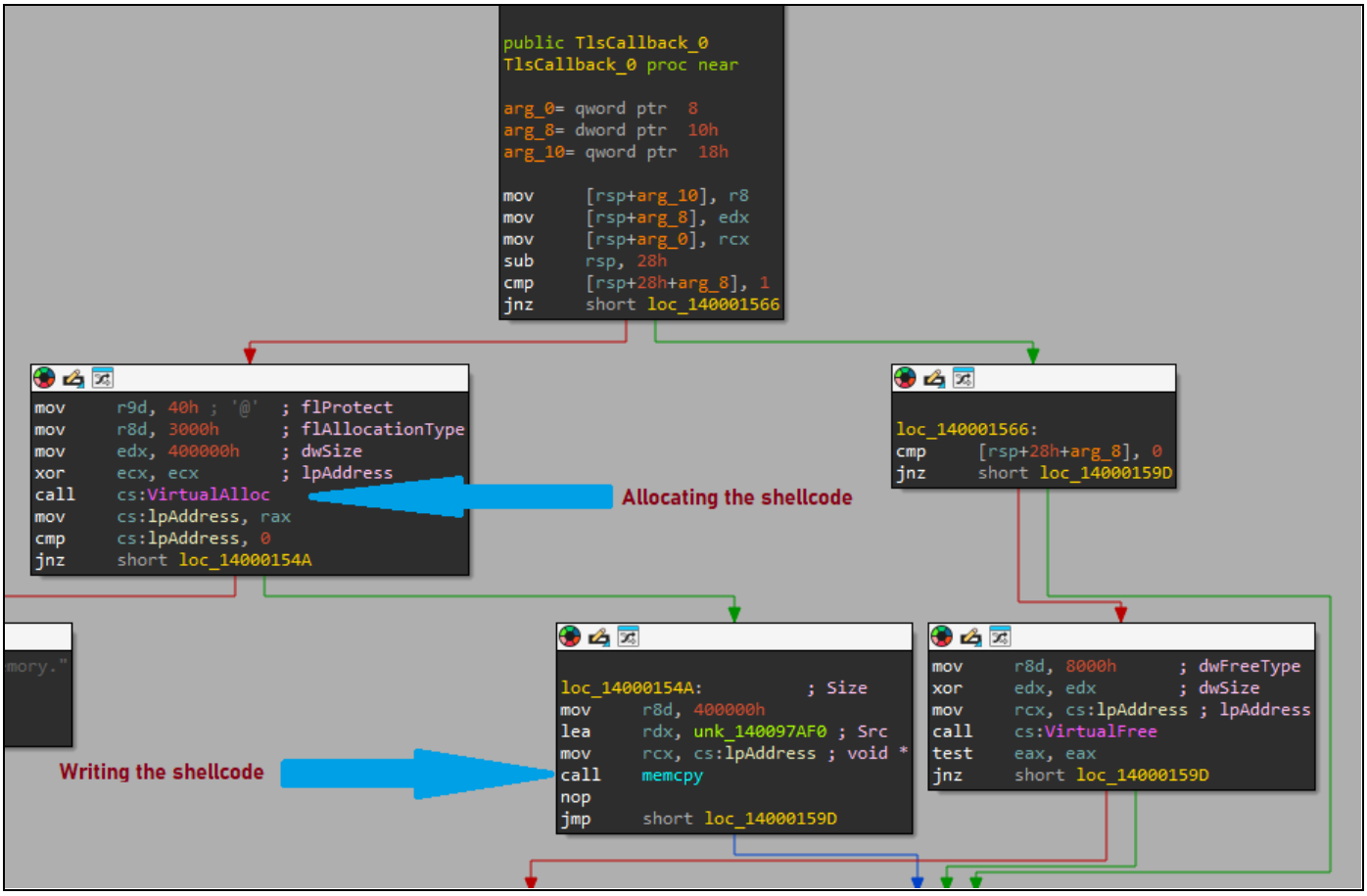

Figure 5: The TLS callback function

By analyzing the other functions, it seems that the file is trying to resolve the API `RtlInstallFunctionTableCallback` from `ntdll.dll` and use it to register the function at `0x1400010B0` as a callback for a dynamic function table partially covering the shellcode region. Figure 6 below shows the manual resolution of that function and Figure 7 shows the function being called.

```
__int64 sub_140001270()
{
  unsigned int i; // [rsp+0h] [rbp-68h]
  __int64 v2; // [rsp+8h] [rbp-60h]
  _BYTE *v3; // [rsp+10h] [rbp-58h]
  __int64 v4; // [rsp+38h] [rbp-30h]

  v2 = qword_1408A3310[1345];
  v4 = *(unsigned int *)(*(int *)(v2 + 60) + v2 + 136);
  for ( i = 0; i < *(_DWORD *)(v4 + v2 + 20); ++i )
  {
    v3 = (_BYTE *)(*(unsigned int *)(*(unsigned int *)(v4 + v2 + 32) + v2 + 4LL * i) + v2);
    if ( *v3 == 'R' && v3[3] == 'I' && v3[10] == 'F' && v3[18] == 'T' && v3[23] == 'C' )
    {
      qword_1408A32F8 = *(unsigned int *)(*(unsigned int *)(v4 + v2 + 28)
                                        + v2
                                        + 4LL * *(unsigned __int16 *)(*(unsigned int *)(v4 + v2 + 36) + v2 + 2LL * i))
                      + v2;
      return 0LL;
    }
  }
  return 0LL;
}
```

Resolving RtlInstallFunctionTableCallback

Figure 6: Resolving RtlInstallFunctionTableCallback

```
__int64 sub_140001430()
{
  void (__fastcall *v1)(__int64, _QWORD, __int64, _QWORD, _QWORD, _QWORD); // [rsp+30h] [rbp-28h]

  v1 = (void (__fastcall *)(__int64, _QWORD, __int64, _QWORD, _QWORD, _QWORD))*((_QWORD *)&unk_14089B910 + 3901);
  if ( v1 )
    v1(
      *((_QWORD *)&unk_140898970 + 1518) | 3LL,
      *((_QWORD *)&unk_140898970 + 1518),
      3034406LL,
      qword_140022000[1206],
      0LL,
      0LL);
  return 0LL;
}
```

Evaluates to 0x1400010B0

Figure 7: Calling RtlInstallFunctionTableCallback

```
PRUNTIME_FUNCTION __fastcall sub_1400010B0(ULONG_PTR ControlPc, PVOID Context)
{
  struct _IMAGE_RUNTIME_FUNCTION_ENTRY *v3; // [rsp+28h] [rbp-20h]

  v3 = (struct _IMAGE_RUNTIME_FUNCTION_ENTRY *)operator new(0xCuLL);
  v3->BeginAddress = ControlPc - *((_QWORD *)&unk_140898970 + 1518);
  v3->EndAddress = v3->BeginAddress + 1;
  v3->UnwindInfoAddress = v3->EndAddress + *(unsigned __int8 *)(ControlPc + 1) + 1;
  v3->UnwindInfoAddress += (v3->UnwindInfoAddress & 1) != 0;
  return v3;
}
```

Figure 8: The callback function 0x1400010B0

The function `0x1400010B0`, shown in Figure 8 above, is called each time an exception occurs in the shellcode. It returns a pointer to the struct RUNTIME_FUNCTION that contains the offset of the stack unwinding information inside the shellcode.

The unwind information offset from the beginning of the shellcode (`UnwindInfoAddress`) is set to be equal to the first byte after the exception address added to the exception upper bound offset from the shellcode beginning (`EndAddress`) aligned.

Advanced Analysis

Using the IDA debugger to analyze the file with any 32-byte key as a parameter, it seems the shellcode is set to raise exceptions using the HLT instruction.

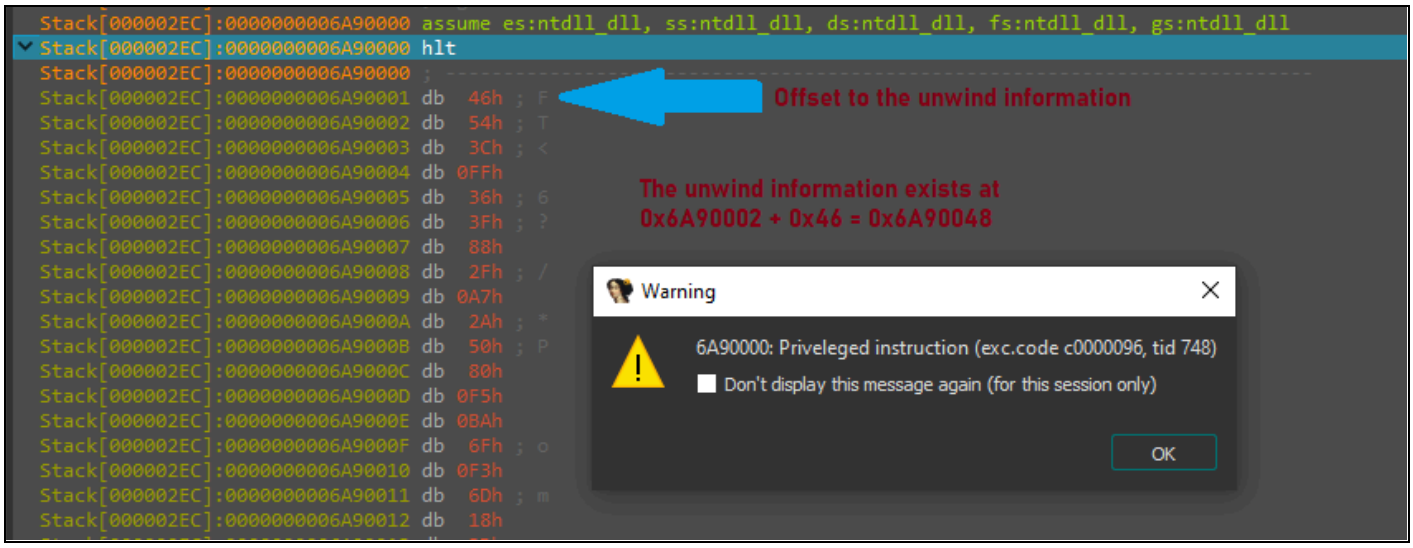

Figure 9: Calculating the unwind information offset

The unwind information structure is called `UNWIND_INFO`. It contains the unwind opcodes and optionally suffixed by the exception handler offset inside the shellcode.

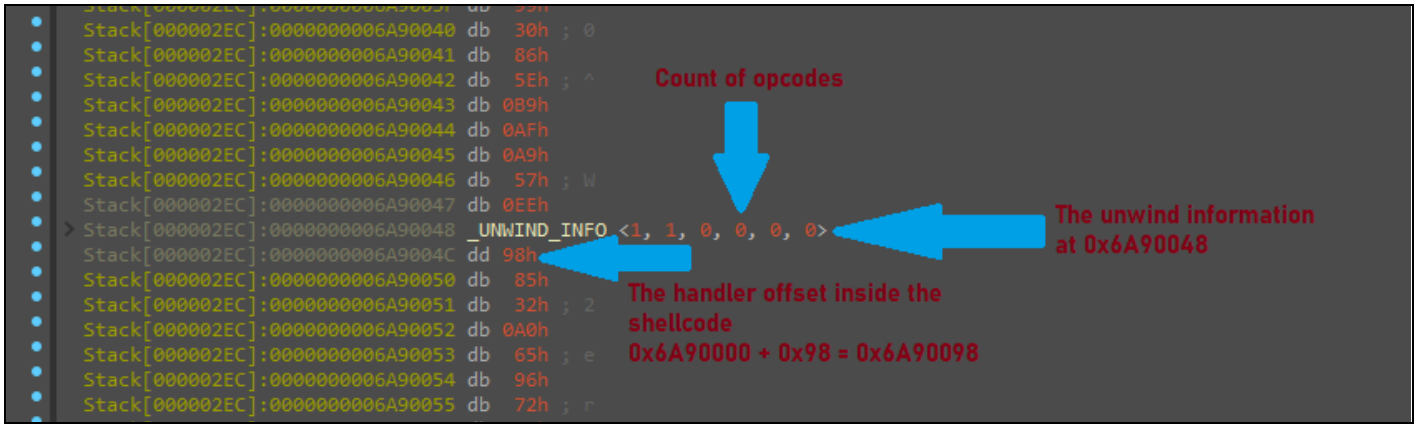

Figure 10: The unwind information

The handler offset inside the shellcode is appended to the unwind information.

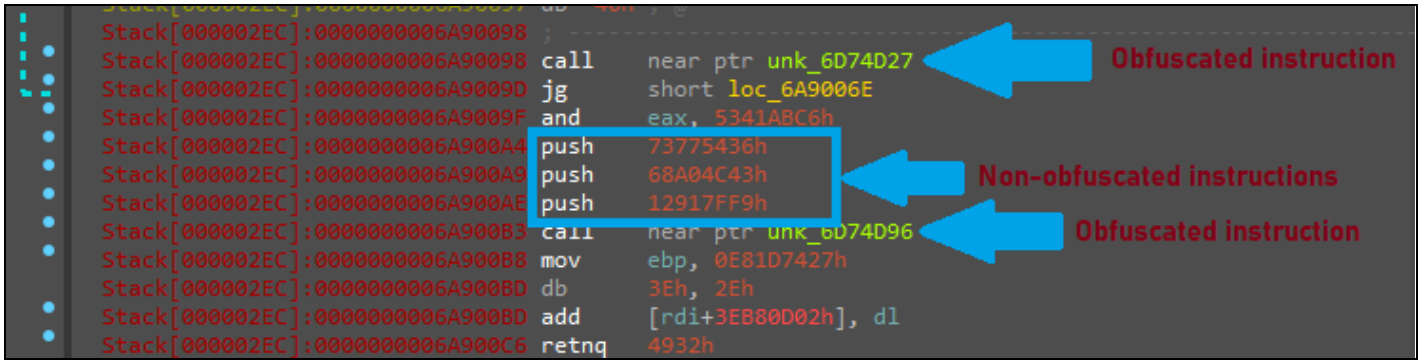

Figure 11: The exception handler

The long handler instructions (instructions longer than 5 bytes) are obfuscated by replacing it with a call instruction to a code block that is responsible for decoding the instruction, executing it and clearing it.

Figure 12: The obfuscated instruction code block

The decoding key is the lower byte of the previous instruction return offset inside the shellcode, it won't be available unless the previous instruction is executed. So, an obfuscated instruction cannot be decoded without the previous obfuscated instruction being executed. For the first obfuscated instruction, the decoding key is hardcoded in the shellcode at offset 0x2E4D26.

Exceptions happen recursively, each exception handler is configured to raise another exception. This requires a huge stack and also requires the stack limits to be patched.

```
__int64 sub_140001230()
{
  struct _TEB *v0; // rax

  v0 = NtCurrentTeb();
  v0->NtTib.StackBase = (PVOID)-1LL;
  v0->NtTib.StackLimit = 0LL;
  return 0LL;
}
```

Figure 13: Patching the stack limits

Achieving the MOV

The theory behind the challenge is to use the unwind opcodes to simulate the **MOV** instruction.
Moving a QWORD from a register plus an offset into another register can be simulated by one of the following two unwind opcode sequences (see https://learn.microsoft.com/en-us/cpp/build/exception-handling-x64?view=msvc-170):

- First sequence (requires one exception)
  - **UWOP_SET_FPREG** to set the exception context's **RSP** with the register.
  - One of **UWOP_ALLOC_*** to shift the **RSP** with the offset value.
  - **UWOP_PUSH_NONVOL** to get a QWORD from the RSP.
- Second sequence (requires two recursive exceptions, the first one's handler saves the register on the stack)
  - **UWOP_PUSH_MACHFRAME** to set the exception context's **RSP** with an entry on the stack (set with the register).
  - One of **UWOP_ALLOC_*** to shift the **RSP** with the offset value.
  - **UWOP_PUSH_NONVOL** to get a QWORD from the **RSP**.

Both sequences are used in the challenge. Memory addresses and other immediates are controlled within the handlers.

For references, see https://github.com/tongzx/nt5src/blob/daad8a087a4e75422ec96b7911f1df4669989611/Source/XPSP1/NT/base/ntos/rtl/amd64/exdsptch.c#L950-L1153.

MOVing on

Since we can simulate a **MOV** instruction from a location to a register and based on the fact that **MOV** is Turing complete, we can use it to implement the other logic operators.

The implemented operators are ADD, SUB, AND, OR and XOR. It requires a set of hardcoded 2D tables for the pre-calculated operations between single bytes.

For example, to AND a flag character with an immediate byte, the following sequence can be used:

- Moving the AND table address to a register.
- Dereference the register using the simulated MOV instruction with the immediate byte as an offset (the result is a 1D table).
- Dereference the resulting table with the flag character as an offset.
- The result is the AND operation result.

For the operators that involve a carry, the carry is added (or subtracted) within the handler (using the tables).

For references, see https://github.com/xoreaxeaxeax/movfuscator/blob/master/slides/domas_2015_the_movfuscator.pdf.

The solution

The **key** is 32-byte long, distributed over 32 first-degree polynomial equations that are checked separately.

The following is the solution script. It parses and deobfuscates the shellcode, extracts the immediates, builds the equations and solves it. It requires the z3-solver python library.

Python

```python
import io, struct, z3
from capstone     import *
from capstone.x86 import *


SHELLCODE_OFFSET    = 0x095ef0
SHELLCODE_SIZE      = 0x800000
DECODING_KEY_OFFSET = 0x2e4d26
KEY_VIRTUAL_ADDRESS = 0x14089b8e8
XOR_VIRTUAL_ADDRESS = 0x140094ac0
AND_VIRTUAL_ADDRESS = 0x1400942c0
OR__VIRTUAL_ADDRESS = 0x1400952c0
ADD_VIRTUAL_ADDRESS = 0x140095ac0
SUB_VIRTUAL_ADDRESS = 0x140096ac0
RIGHT_KEY_FUNCTION  = 0x1400011b0
WRONG_KEY_FUNCTION  = 0x1400011f0
DISASSEMBLER        = Cs(CS_ARCH_X86, CS_MODE_64)
DISASSEMBLER.detail = True


UWOP_PUSH_NONVOL    = 0
UWOP_ALLOC_LARGE    = 1
UWOP_ALLOC_SMALL    = 2
UWOP_SET_FPREG      = 3
UWOP_PUSH_MACHFRAME = 10


REGISTERS = {
 0 : "rax",
 1 : "rcx",
 2 : "rdx",
 3 : "rbx",
 4 : "rsp",
 5 : "rbp",
 6 : "rsi",
 7 : "rdi",
```

```
 8 : "r8" ,
 9 : "r9" ,
 10: "r10",
 11: "r11",
 12: "r12",
 13: "r13",
 14: "r14",
 15: "r15"
}


class UNWIND_INFO:


 def __init__(self, data : bytes):


   unpacked_data      = struct.unpack("BBBB", data)
   self.Version       =  unpacked_data[0]       & 0x07
   self.Flags         = (unpacked_data[0] >> 3) & 0x1F
   self.SizeOfProlog  =  unpacked_data[1]
   self.CountOfCodes  =  unpacked_data[2]
   self.FrameRegister =  unpacked_data[3]       & 0x0F
   self.FrameOffset   = (unpacked_data[3] >> 4) & 0x0F


 @staticmethod
 def size() -> int:
   return 4


class UNWIND_CODE:


 def __init__(self, data : bytes):


   unpacked_data    = struct.unpack("BB", data)
   self.CodeOffset  =  unpacked_data[0]
   self.UnwindOp    =  unpacked_data[1]       & 0x0F
   self.OpInfo      = (unpacked_data[1] >> 4) & 0x0F
   self.FrameOffset =  unpacked_data[0] + (unpacked_data[1] << 8)


 def op_str(self) -> str:


   if self.UnwindOp == UWOP_PUSH_NONVOL:
```

```
      return "UWOP_PUSH_NONVOL"
    if self.UnwindOp == UWOP_ALLOC_LARGE:
      return "UWOP_ALLOC_LARGE"
    if self.UnwindOp == UWOP_ALLOC_SMALL:
      return "UWOP_ALLOC_SMALL"
    if self.UnwindOp == UWOP_SET_FPREG:
      return "UWOP_SET_FPREG"
    if self.UnwindOp == UWOP_PUSH_MACHFRAME:
      return "UWOP_PUSH_MACHFRAME"
    raise Exception(f"Unsupported unwind operation code: {self.UnwindOp}.")


  @staticmethod
  def size() -> int:
    return 2


class Parser:


  def __init__(self, shellcode: bytes, decoding_key: int) -> None:


    self.__stream     = io.BytesIO(shellcode)
    self.__dec_key    = decoding_key
    self.__exceptions = []


  def __deobfuscate_instruction(self) -> CsInsn:


    assert self.__stream.read(1)[0] == 0xe8


    call_offset  = struct.unpack("<I", self.__stream.read(4))[0]
    retn_offset  = self.__stream.tell()
    call_offset += retn_offset
    self.__stream.seek(call_offset)


    obf_header = self.__stream.read(34)
    ret_offset = struct.unpack("<I", obf_header[0x02:0x06])[0]
    obf_part   = struct.unpack("<I", obf_header[0x17:0x1b])[0] + (self.__dec_key << 8)


    instr_size = ret_offset - 0x29
    obf_offset = self.__stream.tell()
    obf_instr  = self.__stream.read(instr_size)
```

```
  obf_footer = self.__stream.read(30)
  obf_instr  = struct.pack("<I", obf_part) + obf_instr[4:]


  self.__dec_key =   retn_offset & 0xff
  retn_offset   +=   obf_footer   [0x18]
  self.__stream.seek(retn_offset)


  instr = list(DISASSEMBLER.disasm(obf_instr, obf_offset))[0]
  instr._raw.address = retn_offset - instr.size


  relocation    = obf_offset - instr.address
  reloc_operand = next((operand for operand in instr.operands if  \
                        operand.type     == X86_OP_MEM        and \
                        operand.mem.base == X86_REG_RIP), None)


  if instr.encoding.disp_size and reloc_operand:
      assert instr.disp_size == 4
      obf_instr = obf_instr[:instr.encoding.disp_offset]   +               \
                  struct.pack("<I", reloc_operand.mem.disp + relocation) + \
                  obf_instr[instr.encoding.disp_offset     + 4:]
      instr = list(DISASSEMBLER.disasm(obf_instr, instr.address))[0]


  return instr


def __get_next_info(self) -> tuple[UNWIND_INFO, list[UNWIND_CODE]]:


  assert self.__stream.read(1)[0] == 0xf4


  info_offset  = self.__stream.read(1)[0]
  info_offset += self.__stream.tell( )
  info_offset  = (info_offset + 1) if (info_offset & 1) else info_offset
  self.__stream.seek(info_offset)


  unwind_info  = UNWIND_INFO(self.__stream.read(UNWIND_INFO.size()))
  unwind_codes = []
  for _ in range(unwind_info.CountOfCodes):
    unwind_codes.append(UNWIND_CODE(self.__stream.read(UNWIND_CODE.size())))
```

```
  if unwind_info.CountOfCodes & 1:
    self.__stream.read(UNWIND_CODE.size())


  handler_offset = struct.unpack("<I", self.__stream.read(4))[0]
  self.__stream.seek(handler_offset)


  return unwind_info, unwind_codes


def __read_handler(self) -> bytes:


  handler = self.__stream.read(0x100)
  self.__stream.seek(self.__stream.tell() - 0x100)
  return handler


def __parse_handler(self) -> list[CsInsn]:


  handler = self.__read_handler()


  instructions  = []
  disassembling = True
  while disassembling:
    for instr in DISASSEMBLER.disasm(handler, self.__stream.tell()):
      inside_handler = True
      if  instr.mnemonic == "call":
        self.__stream.seek(instr.address)
        instr = self.__deobfuscate_instruction()
        inside_handler = False


      if  instr.mnemonic == "jmp" and instr.imm_size:
        self.__stream.seek(instr.operands[0].imm)
        handler = self.__read_handler()
        inside_handler = False


      elif \
        instr.mnemonic == "hlt":
        self.__stream.seek(instr.address)
        disassembling = False
```

```
        else:
          handler = handler[instr.size:]


        instructions.append(instr)
        if not inside_handler or not disassembling:
          break


    return instructions


 def __decode_unwind_codes(self, address: int, info: UNWIND_INFO, codes: list[UNWIND_CODE],
target: int) -> None:


    if not info.CountOfCodes:
      print(f"{hex(address)}:\tmov\tr9, <DISPATCHER_CONTEXT>")
      print(f"{hex(address + 7)}:\tjmp\t{hex(target)}")
      return


    assert info.CountOfCodes  >= 2
    if  codes[0].UnwindOp == UWOP_SET_FPREG:
      source = REGISTERS[info.FrameRegister]
      size   = 3


    elif \
      codes[0].UnwindOp == UWOP_PUSH_MACHFRAME:
      source = f"qword ptr [rsp + {hex((3 + codes[0].OpInfo) * 0x08)}]"
      size   = 5


    else:\
      raise Exception(f"Unexpected unwind operation code: {codes[0].UnwindOp}.")


    assert codes[-1].UnwindOp == UWOP_PUSH_NONVOL
    offset      = self.get_offset_from_opcodes(codes[1:-1])
    destination = REGISTERS[codes[-1].OpInfo]


    print(f"{hex(address)}:\tmov\t{destination}, {source}")
    print(f"{hex(address + size)}:\tmov\t{destination}, qword ptr [{destination} +
{hex(offset)}]")
    if offset < 0x7f:
      size += 4
```

```
    else:
      size += 7
    print(f"{hex(address + size)}:\tmov\tr9, <DISPATCHER_CONTEXT>")
    print(f"{hex(address + size + 7)}:\tjmp\t{hex(target)}")


  def parse(self) -> None:


    address = 1
    print("0x0:\thlt\t")
    while True:
      info, codes  = self.__get_next_info()
      instructions = self.__parse_handler()


      print("; ************** UnwindInfo **************"     )
      print(f"; info.Version      : {hex(info.Version)}"     )
      print(f"; info.Flags        : {hex(info.Flags)}"       )
      print(f"; info.SizeOfProlog : {hex(info.SizeOfProlog)}")
      print(f"; info.CountOfCodes : {hex(info.CountOfCodes)}")


      if any(code.UnwindOp == UWOP_SET_FPREG for code in codes):
        print(f"; info.FrameRegister: {hex(info.FrameRegister)}
({REGISTERS[info.FrameRegister]})")
      else:
        print(f"; info.FrameRegister: {hex(info.FrameRegister)}")


      print(f"; info.FrameOffset  : {hex(info.FrameOffset)}"  )
      if codes:
        print("; ============== UnwindCode ==============")


      idx = 0
      while idx < len(codes):
        print(f"; codes[{idx}].CodeOffset : {hex(codes[idx].CodeOffset)}" )
        print(f"; codes[{idx}].UnwindOp   : {codes[idx].op_str()}"   )


        if codes[idx].UnwindOp == UWOP_PUSH_NONVOL:
          print(f"; codes[{idx}].OpInfo     : {hex(codes[idx].OpInfo)}
({REGISTERS[codes[idx].OpInfo]})")
        else:
          print(f"; codes[{idx}].OpInfo     : {hex(codes[idx].OpInfo)}" )
```

```
  if idx != len(codes) - 1:
    print("; ----------------------------------------")


  if codes[idx].UnwindOp != UWOP_ALLOC_LARGE:
    idx += 1
    continue


  op_info = codes[idx].OpInfo
  assert op_info == 0 or op_info == 1
  for _ in range(op_info + 1):
    idx += 1
    print(f"; codes[{idx}].FrameOffset: {hex(codes[idx].FrameOffset)}")
    print("; ----------------------------------------")


  idx += 1


print("; ========== UnwindCode Meaning ==========")
self.__decode_unwind_codes(address, info, codes, instructions[0].address)


if instructions:
  print("; ================ EHandler ===============")


for instr in instructions:
  print(f"{hex(instr.address)}:\t{instr.mnemonic}\t{instr.op_str}")
  address =    instr.address + instr.size


assert len(instructions) > 2
assert instructions[-1].mnemonic == "hlt" and \
       instructions[-2].mnemonic == "jmp" and \
       instructions[-2].encoding.imm_size
instructions = instructions[:-2]


self.__exceptions.append((info, codes, instructions))
if len(instructions) == 3 and \
  instructions[0].mnemonic == "movabs" and \
  instructions[1].mnemonic == "add"    and \
  instructions[2].mnemonic == "jmp":
  print("; ****************************************")
  break
```

```
 @staticmethod
 def get_offset_from_opcodes(codes: list[UNWIND_CODE]) -> int:


   if not codes:
     return 0


   assert codes[0].UnwindOp == UWOP_ALLOC_SMALL or codes[0].UnwindOp == UWOP_ALLOC_LARGE
   if codes[0].UnwindOp == UWOP_ALLOC_SMALL:
     assert len(codes) == 1
     return (codes[0].OpInfo + 1) << 3
   else:
     assert codes[0].OpInfo == 0 or codes[0].OpInfo == 1
     if codes[0].OpInfo == 0:
       assert len(codes) == 2
       return codes[1].FrameOffset << 3
     else:
       assert len(codes) == 3
       return codes[1].FrameOffset + (codes[2].FrameOffset << 16)


 @property
 def exceptions(self) -> list[tuple[UNWIND_INFO], list[UNWIND_CODE], list[CsInsn]]:
   return self.__exceptions


class Solver:


 def __init__(self, shellcode: bytes, decoding_key: int) -> None:


   self.__parser     = Parser(shellcode, shellcode[DECODING_KEY_OFFSET])
   self.__parser.parse()
   self.__exceptions = self.__get_exceptions()
   self.__imm_index  = 0
   self.__immediate  = [0x0] * 8
   self.__cur_table  = 0
   self.__equation   = ""
   self.__operation  = ""
   self.__key_mul    = ""
   self.__solver     =  z3.Solver()
   self.__key        = [z3.BitVec('key[%d]' % i, 64) for i in range(32)]


   for idx in range(32):
```

```
    self.__solver.add(self.__key[idx] > 31, self.__key[idx] < 127)


def __get_exceptions(self) -> ...:


  for exception in self.__parser.exceptions:
    yield exception


def __commit_immediate(self) -> None:


  if self.__cur_table:
     immediate_value  = struct.unpack("<Q", bytes(self.__immediate))[0]
     self.__equation  = f"({self.__equation} {self.__operation} {hex(immediate_value)})"
     self.__immediate = [0x0] * 8
     self.__cur_table = 0


def __move_mem_ptr_to_reg(self, address: int, offset: int) -> None:


  if address != KEY_VIRTUAL_ADDRESS:
    assert offset % 8 == 0
    offset >>= 3
    assert offset < 0x100


  if address == KEY_VIRTUAL_ADDRESS:
     self.__key_mul = f"key[0x{'{:02x}'.format(offset)}]"
     self.__commit_immediate()
     return


  assert self.__immediate[self.__imm_index] == 0x00 or \
         self.__immediate[self.__imm_index] == 0xff and self.__cur_table ==
AND_VIRTUAL_ADDRESS
  self.__immediate       [self.__imm_index] = offset


  if address == self.__cur_table:
    return
  else:
    self.__commit_immediate()
    self.__cur_table = address
```

```
    if address == XOR_VIRTUAL_ADDRESS:
        self.__operation = "^"


    elif \
        address == ADD_VIRTUAL_ADDRESS:
        self.__operation = "+"


    elif \
        address == SUB_VIRTUAL_ADDRESS:
        self.__operation = "-"


    elif \
        address == AND_VIRTUAL_ADDRESS:
        self.__immediate = [0xff] * 8
        self.__operation = "&"


    elif \
        address == OR__VIRTUAL_ADDRESS:
        self.__operation = "|"

    else:
        raise Exception(f"Unknown table address: {hex(address)}.")


 def __move_mem_ptr_to_reg_direct(self, info: UNWIND_INFO, codes: list[UNWIND_CODE],
instructions: list[CsInsn]) -> None:


    address_index = len(instructions) - 2
    address_abs   = instructions[address_index + 0].operands[1].imm
    address_abs  += instructions[address_index + 1].operands[1].imm


    info, codes, instructions = next(self.__exceptions)
    offset = Parser.get_offset_from_opcodes(codes[1:-1])


    self.__move_mem_ptr_to_reg(address_abs, offset)


 def __move_mem_ptr_to_reg_stack(self, info: UNWIND_INFO, codes: list[UNWIND_CODE],
instructions: list[CsInsn]) -> None:
```

```
    address_abs = instructions[0].operands[1].imm
    if instructions[5].mnemonic == "push":
      address_abs += instructions[6].operands[1].imm
    else:
      address_abs += instructions[5].operands[1].imm


    info, codes, instructions = next(self.__exceptions)
    offset = Parser.get_offset_from_opcodes(codes[1:-1])


    self.__move_mem_ptr_to_reg(address_abs, offset)


  def __move_reg_ptr_to_reg(self, offset) -> None:


    assert offset < 0x8
    if offset <= self.__imm_index:
       self.__commit_immediate()
    self.__imm_index = offset


  def __move_reg_ptr_to_reg_direct(self, info: UNWIND_INFO, codes: list[UNWIND_CODE],
instructions: list[CsInsn]) -> None:


    offset = Parser.get_offset_from_opcodes(codes[1:-1])
    self.__move_reg_ptr_to_reg(offset)


  def __move_reg_ptr_to_reg_stack(self, info: UNWIND_INFO, codes: list[UNWIND_CODE],
instructions: list[CsInsn]) -> None:


    info, codes, instructions = next(self.__exceptions)
    offset = Parser.get_offset_from_opcodes(codes[1:-1])
    self.__move_reg_ptr_to_reg(offset)


  def __set_reg_nth_from_table(self, info: UNWIND_INFO, codes: list[UNWIND_CODE],
instructions: list[CsInsn]) -> None:


    if len(instructions) == 6:
      reg_idx = 0
    else:
      reg_idx = instructions[6].operands[1].imm
```

```
   assert reg_idx % 8 == 0
   reg_idx >>= 3
   assert reg_idx < 0x8
   assert reg_idx == self.__imm_index


 def __get_reg_ptr(self, info: UNWIND_INFO, codes: list[UNWIND_CODE], instructions:
list[CsInsn]) -> None:


   if self.__key_mul:
      self.__equation = f"({self.__key_mul})"
      self.__key_mul  = ""


 def __add_reg_to_reg(self, info: UNWIND_INFO, codes: list[UNWIND_CODE], instructions:
list[CsInsn]) -> None:


   self.__equation = f"({self.__equation} + {self.__key_mul})"
   self.__key_mul  =  ""


 def __sub_reg_to_reg(self, info: UNWIND_INFO, codes: list[UNWIND_CODE], instructions:
list[CsInsn]) -> None:


   self.__equation = f"({self.__equation} - {self.__key_mul})"
   self.__key_mul  =  ""


 def __xor_reg_to_reg(self, info: UNWIND_INFO, codes: list[UNWIND_CODE], instructions:
list[CsInsn]) -> None:


   self.__equation = f"({self.__equation} ^ {self.__key_mul})"
   self.__key_mul  =  ""


 def __get_reg_multiplier(self, info: UNWIND_INFO, codes: list[UNWIND_CODE], instructions:
list[CsInsn]) -> None:


   immediate  = instructions[2].operands[1].imm
   immediate += instructions[3].operands[1].imm
```

```
    self.__key_mul += f" * {hex(immediate)}"


 def __jump_to_handler_if_reg(self, info: UNWIND_INFO, codes: list[UNWIND_CODE],
instructions: list[CsInsn]) -> None:


    jmp_address  = instructions[2].operands[1].imm
    jmp_address += instructions[3].operands[1].imm
    assert jmp_address == WRONG_KEY_FUNCTION


    self.__commit_immediate()
    exec(f"solver.add({self.__equation} == 0x0)", {"solver": self.__solver, "key":
self.__key})


 def __jump_to_handler(self, info: UNWIND_INFO, codes: list[UNWIND_CODE], instructions:
list[CsInsn]) -> None:


    jmp_address  = instructions[0].operands[1].imm
    jmp_address += instructions[1].operands[1].imm
    assert jmp_address == RIGHT_KEY_FUNCTION


    print("Calculating the key ...")


    self.__solver.check()
    model = self.__solver.model()


    key = bytes(model[self.__key[idx]].as_long() for idx in range(32)).decode()
    print(f"Flag: {key}@flare-on.com")


 def get_flag(self) -> None:


    while True:
      try:
        info, codes, instructions = next(self.__exceptions)
      except StopIteration:
        break
```

```
if  len(instructions) == 2 and instructions[0].mnemonic == "movabs" or \
    len(instructions) == 4 and instructions[2].mnemonic == "movabs":
    self.__move_mem_ptr_to_reg_direct(info, codes, instructions)


elif \
  len(instructions) > 5 and \
  instructions[0].mnemonic == "movabs" and \
  instructions[1].mnemonic == "push"   and \
  instructions[2].mnemonic == "push"   and \
  instructions[3].mnemonic == "push"   and \
  instructions[4].mnemonic == "push":
  self.__move_mem_ptr_to_reg_stack(info, codes, instructions)


elif \
  (len(instructions) == 4 and \
    instructions[0].mnemonic == "mov"     and \
    instructions[1].mnemonic == "ldmxcsr" and \
    instructions[2].mnemonic == "movabs") or  \
  (len(instructions) == 6 and \
    instructions[0].mnemonic == "mov"     and \
    instructions[1].mnemonic == "ldmxcsr" and \
    instructions[4].mnemonic == "movabs"):
  self.__move_mem_ptr_to_reg_direct(info, codes, instructions[2:])


elif \
  len(instructions) > 7 and \
  instructions[0].mnemonic == "mov"     and \
  instructions[1].mnemonic == "ldmxcsr" and \
  instructions[2].mnemonic == "movabs"  and \
  instructions[3].mnemonic == "push"    and \
  instructions[4].mnemonic == "push"    and \
  instructions[5].mnemonic == "push"    and \
  instructions[6].mnemonic == "push":
  self.__move_mem_ptr_to_reg_stack(info, codes, instructions[2:])


elif \
  (len(instructions) == 12 and \
    instructions[0 ].mnemonic == "mov"     and \
    instructions[1 ].mnemonic == "ldmxcsr" and \
    instructions[10].mnemonic == "movabs") or  \
  (len(instructions) == 14 and \
    instructions[0 ].mnemonic == "mov"     and \
    instructions[1 ].mnemonic == "ldmxcsr" and \
    instructions[12].mnemonic == "movabs"):
```

```
    self.__move_mem_ptr_to_reg_direct(info, codes, instructions[10:])


  elif \
    len(instructions) > 15 and \
    instructions[0 ].mnemonic == "mov"     and \
    instructions[1 ].mnemonic == "ldmxcsr" and \
    instructions[10].mnemonic == "movabs"  and \
    instructions[11].mnemonic == "push"    and \
    instructions[12].mnemonic == "push"    and \
    instructions[13].mnemonic == "push"    and \
    instructions[14].mnemonic == "push":
    self.__move_mem_ptr_to_reg_stack(info, codes, instructions[10:])


  elif \
    all(instruction.mnemonic.startswith("mov") for instruction in instructions):
    self.__move_reg_ptr_to_reg_direct(info, codes, instructions)


  elif \
    len(instructions) >= 6 and \
    instructions[0].mnemonic == "mov"  and \
    instructions[1].mnemonic == "mov"  and \
    instructions[3].mnemonic == "push" and \
    instructions[4].mnemonic == "push" and \
    instructions[5].mnemonic == "push":
    self.__move_reg_ptr_to_reg_stack(info, codes, instructions)


  elif \
    len(instructions) >= 6 and \
    instructions[0].mnemonic == "mov" and \
    instructions[1].mnemonic == "mov" and \
    instructions[2].mnemonic == "mov" and \
    instructions[3].mnemonic == "add" and \
    instructions[4].mnemonic == "mov" and \
    instructions[5].mnemonic == "mov":
    self.__set_reg_nth_from_table(info, codes, instructions)


  elif \
    len(instructions) >= 4 and \
    instructions[0].mnemonic == "mov"  and \
    instructions[1].mnemonic == "mov"  and \
    instructions[2].mnemonic == "push" and \
    instructions[3].mnemonic == "mov":
    self.__get_reg_ptr(info, codes, instructions)
```

```
elif \
  len(instructions) == 3 and \
  instructions[0].mnemonic == "mov" and \
  instructions[1].mnemonic == "mov" and \
  instructions[2].mnemonic == "add":
  self.__add_reg_to_reg(info, codes, instructions)


elif \
  len(instructions) == 3 and \
  instructions[0].mnemonic == "mov" and \
  instructions[1].mnemonic == "mov" and \
  instructions[2].mnemonic == "sub":
  self.__sub_reg_to_reg(info, codes, instructions)


elif \
  len(instructions) == 3 and \
  instructions[0].mnemonic == "mov" and \
  instructions[1].mnemonic == "mov" and \
  instructions[2].mnemonic == "xor":
  self.__xor_reg_to_reg(info, codes, instructions)


elif \
  len(instructions) == 7 and instructions[5].mnemonic == "mul" or \
  len(instructions) == 8 and instructions[6].mnemonic == "mul":
  self.__get_reg_multiplier(info, codes, instructions)


elif \
  len(instructions) == 8 and \
  instructions[6].mnemonic == "cmovne" and \
  instructions[7].mnemonic == "jmp":
  self.__jump_to_handler_if_reg(info, codes, instructions)


elif \
  len(instructions) == 3 and \
  instructions[0].mnemonic == "movabs" and \
  instructions[1].mnemonic == "add"    and \
  instructions[2].mnemonic == "jmp":
  self.__jump_to_handler(info, codes, instructions)


else:
```

```
        raise Exception("Unable to identify a code block.")


if __name__ == '__main__':


 import sys
 if len(sys.argv) != 2:
   print(f"{sys.argv[0]} <file>")
   exit(1)


 content   = open(sys.argv[1], "rb").read()
 shellcode = content[SHELLCODE_OFFSET: SHELLCODE_OFFSET + SHELLCODE_SIZE]


 solver = Solver(shellcode, shellcode[DECODING_KEY_OFFSET])
 solver.get_flag()
```

Figure 14: Python Solver Script

Executing the script in Figure 14 above will print to the console the final flag.

Final Flag

```
Unset
$$_4lway5_k3ep_mov1ng_and_m0ving@flare-on.com
```