Hurdles (Part 1+2)

Category: Reverse Engineering

Difficulty: Medium (Part 1); Hard (Part 2)

Author: zonkies

Description

Part 1

> CSCG is in may ways like hurdling. Our challenge attempts to hide the obstacles in a way you might not be used to. Can you solve the first two stages?
>
> Note: The flag(s) will start with a lowercase cscg{. Please uppercase the start characters.

Part 2

> Okay, warmup phase is over. Are you ready now for the real deal? The second half will challenge you even more!
>
> Note: The flag(s) will start with a lowercase cscg{. Please uppercase the start characters.

Summary

We're given a binary with 4 stages (4 checks, implemented in 4 different functions with different obfuscation). The first two stages are part 1 of the challenge (one will receive the first flag) and the last two stages are part 2. This writeup will discuss both parts and go through each stage individually

Solution

To solve the challenge, we employed Ghidra and angr. We initially used Ghidra to decompile the functions and used angr to solve some stages. The decompilation of the main function is as follows (functions already renamed):

```
undefined8 main(uint param_1,char **param_2)

{
  uint8_t cVar1;
  char cVar2;

  cVar1 = check_arglen_less_than_35(param_1,param_2);
  if ((cVar1 != 0) && (cVar2 = _stage1(param_1,param_2), cVar2 != 0)) {
    puts("You have completed stage 1");
    cVar2 = _stage2((ulong)param_1,param_2);
    if (cVar2 != 0) {
      puts("You have completed stage 2");
      _print_interims_flag(param_1,param_2);
      cVar2 = _stage3((ulong)param_1,param_2);
      if (cVar2 != '\0') {
        puts("You have completed stage 3");
        cVar2 = _stage4(param_1,param_2);
        if (cVar2 != '\0') {
          puts("You have completed stage 4");
          printf("Here is your flag: cscg{%s}\n",param_2[1]);
          return 0;
        }
      }
    }
  }
  puts("Bad input");
  return 0xffffffff;
}
```

Stage 0

Initially, the program calls a function that we renamed to `check_arglen_less_than_35`. Ghidra decompiled it as follows:

```
ulong check_arglen_less_than_35(int argc,char **argv)

{
  uint8_t success;

  if (argc == 2) {
    _success = __strlen(argv[1],0x2687083e,0x2687083e,0x332f7f9e);
    return _success & 0xffffffffffffff00 | (ulong)(_success < 0x23);
  }
  return 0;
```

One can view the weird arguments to the `__strlen` function. It seems this is part of the ofbsucation. The `__strlen` function ignores these parameters and just calls an implementation of `strlen` afterwards. Therefore, in total the function checks whether the first argv parameter is less than 35 long (and most likely 34). It also tells us that the input to the program will be passed via `arv[1]`

Stage 1

The decompilation is as follows:

```
uint _stage1(int argc,char **argv)

{
  uint uVar1;
  ulong uVar2;
  char *_arg;

  _arg = argv[1];
  uVar2 = __strlen(_arg,-0x2bafe328,0x4592b814,0xd135d6b);
  if (uVar2 < 0xb) {
    return 0;
  }
  do {
                    /* break if var is 1 or 0 */
    if (((some_var - 1) * some_var & 1) == 0) break;
  } while (((some_var - 1) * some_var) % 2 != 0);
  if (*_arg != '1') {
    return 0;
  }
  if (_arg[1] != '_') {
    return 0;
  }
  if (_arg[2] != 'k') {
    return 0;
  }
  do {
    if (((some_var + 0xb572a6db) * some_var & 1) != 0) goto LAB_00400a2c;
    do {
      if (((some_var - 1) * some_var & 1) == 0) {
        if (_arg[3] != 'n') {
          return 0;
        }
        if (_arg[4] != '0') {
          return 0;
        }
        if (_arg[5] != 'w') {
          return 0;
        }
        goto LAB_00400a7b;
      }
LAB_00400a2c:
    } while ((int)some_other_dat < 0x27);
  } while( true );
  while (0x55 < (int)some_other_dat) {
LAB_00400a7b:
    if (((some_var - 1) * some_var & 1) == 0) break;
  }
  uVar1 = 0;
  if ((_arg[6] == '_') && (uVar1 = 0, _arg[7] == 'h')) {
    do {
    } while (0x4d < (int)some_other_dat);
    uVar1 = 0;
    if (_arg[8] == '0') {
      do {
      } while (0x5a < (int)some_other_dat);
      uVar1 = 0;
      if (_arg[9] == 'w') {
        do {
          uVar1 = some_other_dat;
          if ((int)some_other_dat < 0x19) break;
          uVar1 = some_var;
        } while (((some_var - 1) * some_var & 1) != 0);
        uVar1 = uVar1 & 0xffffff00 | (uint)(_arg[10] == '_');
      }
    }
  }
  return uVar1;
}
```

What we can observe here is the following:

1. `some_var` and `some_other_data` are used in many loops. They are, however, never modified and their initial value is 0. Thus, those conditions and loops can be easily ignored
2. Individual characters are directly compared to another character.

Since each character is just directly compared, we can easily read out the first part manually: `1_kn0w_h0w_`

Stage 2

Stage 2 is a bit more complicated, here is the decompilation:

```
undefined8 _stage2(undefined8 argc,char **argv)

{
  ulong uVar1;
  undefined8 success_var;
  byte bVar2;
  char cVar3;
  byte bVar4;
  byte bVar5;
  char *_arg;

  _arg = argv[1];
  uVar1 = ___strlen(_arg,0xa8742202,0xa8742202,0x3f052cc3);
  switch(uVar1 < 0xf) {
  case false:
    bVar2 = -_arg[0xb];
    cVar3 = (bVar2 & 0x67) * '\x02' + (bVar2 ^ 0xe7);
    bVar2 = -cVar3 - 0x19;
    bVar4 = cVar3 + 0x18;
    switch((byte)((bVar2 & 0xd0) * '\x02' + (((bVar4 & bVar2 | bVar4) ^ bVar2) & 0xd0 ^ bVar
          ) {
    case false:
      success_var = 0;
      break;
    case true:
      bVar4 = -_arg[0xc];
      bVar2 = bVar4 ^ 0x30;
      switch((byte)-((~bVar2 & bVar4) * '\x02' + bVar2) < 10) {
      case false:
        success_var = 0;
        break;
      case true:
        bVar2 = _arg[0xd] - 0x39;
        bVar4 = ~bVar2;
        switch((byte)(~(~(bVar2 ^ 0x2f) & ~(bVar4 ^ bVar2 ^ 0x2f) | bVar4) * '\x02' +
                      (bVar2 & 0x2f | bVar4 & 0xd0) + 0x39) < 10) {
        case false:
          success_var = 0;
          break;
        case true:
          bVar2 = _arg[0xe];
          bVar5 = ~bVar2;
          bVar4 = ~(~(bVar2 & 0xd0 | bVar5 ^ 0x2f) | bVar5 & (bVar5 ^ 0x2f) | bVar2 & 0x2f)
          ;
          bVar5 = (bVar5 & 0xd0 ^ bVar2 & 0x2f) + 0x5f;
          switch((byte)((bVar5 & bVar4) * '\x02' + (bVar5 ^ bVar4) + 0xa1) < 10) {
          case false:
            success_var = 0;
            break;
          case true:
            success_var = CONCAT71(0xa6e3a29dbfb830,
                                   *(short *)(((ulong)(byte)_arg[0xb] * 1000 + 0x2c8e2eb1202
                                               (ulong)(byte)_arg[0xc] * 100 +
                                               (ulong)(byte)_arg[0xd] * 10 + (ulong)bVar2) *
```

```
                                                      -0x591c5d623fff17e2) == 0x3419);
          }
        }
      }
    }
    break;
  case true:
    success_var = 0;
  }
  return success_var;
}
```

This code now is more complicated and the values cannot be directly read out. The formulas could be manually inspected and simplified, but we are lazy and thought, `angr` can do the job for us.

We first initialize the angr state:

```
import angr
import claripy

base_addr = 0x400000
main_function = base_addr + 0x810
p = angr.Project("hurdles", auto_load_libs=False)

flag_chars = [claripy.BVS('flag_%d' % i, 8) for i in range(34)]

argv_addr  = 0xdead0000
argv1_addr = 0xdead0020
argv2_addr = 0xdead0040

state = p.factory.call_state(main_function, 2, argv_addr, add_options=
                             ({angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,
                               angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS}))

flag = claripy.Concat(*flag_chars + [claripy.BVV(b'\x00')])

state.mem[argv_addr].uint64_t = argv1_addr
state.mem[argv_addr+8].uint64_t = argv2_addr
state.mem[argv_addr+0x10].uint64_t = 0
state.memory.store(argv1_addr, b"./linear_code\x00")
state.memory.store(argv2_addr, flag)
```

And then write code for exploration:

```
sm = p.factory.simulation_manager(state)

# Address after solving stage 2
```

```
sm.explore(find=lambda s: s.solver.eval(s.regs.rip) == 0x40085b)

if sm.errored:
    print(sm.errored)

for s in sm.found:
    print(s.regs.al)
    print(s.regs.eax)
    print(s.regs.rax)
    _flag_out = b""
    for flag_char in flag_chars:
        _flag_out += bytes([s.solver.eval(flag_char)])
    print(_flag_out)
    print(s.posix.dumps(1))


for s in sm.active:
    _flag_out = b""
    for flag_char in flag_chars:
        _flag_out += bytes([s.solver.eval(flag_char)])
    print(_flag_out)

for s in sm.deadended:
    print("deadended at:")
    print(s.regs.rip)
        _flag_out = b""
        for flag_char in flag_chars:
            _flag_out += bytes([s.solver.eval(flag_char)])
        print(_flag_out)
        print(hex(s.solver.eval(s.regs.rip)))
```

The code did not directly work. Most often this is the case of the program indexing an array with a symbolic index. By default, angr will concretize the index and will not consider every possible index. Thus, we write a custom concretization strategy and print relevant information about where this happens:

```
class ConstArraySimConcretizationStrategy \
    (angr.concretization_strategies.SimConcretizationStrategy):

    """
    Concretization strategy that returns all solutions
    if less than 10 solutions exist, otherwise throw error.
    """
    def _concretize(self, memory, addr, **kwargs):
        print(f"Concretizing memory addr {addr}")
        _min = self._min(memory, addr, **kwargs)
        _max = self._max(memory, addr, **kwargs)
        print(f"Min = {_min:x}, Max = {_max:x}")
        addrs = self._eval(memory, addr, 10)
        if len(addrs) == 10:
            print(addrs)
            for addr in sm.active[0].history.bbl_addrs: print(hex(addr))
            assert False
        #print(list(map(hex, addrs)))
        return addrs

def print_concretized_address(state):
    print(f"Concretizing address at {state.regs.rip}")
    if state.solver.eval(state.regs.rip) == 0x401d31:
        print(f"Concretizing address rax = {state.regs.rax}")
        print(f"Concretizing address r8 = {state.regs.r8}")

state.memory.read_strategies.insert(0, ConstArraySimConcretizationStrategy())
state.inspect.b('address_concretization', when=angr.BP_BEFORE,
                  action=print_concretized_address)
```

This code will print out the `rip` as well as min and max values of the array (to easily identify the array). By inspecting the instruction we then can also see the size of the array elements.

The instruction(s) in question is, here is the following:

```
CMP        word ptr [RAX + R8*0x2 + value_to_short_map],CX
SETZ       AL
```

`value_to_short_map` mostly looks like an array where at each index i the value i is stored as a `short`. Thus, we decided to model the array access as such.

However, we could not get a solution to work. We were also confused as the desired value seemed to be outside the array. At this point, however, we looked at the inputs generated for unsuccessful states and noticed the following: Stage 2 checks 4 characters and they were always digits. We concluded that the first 4 checks (as seen in the decompilation) were checking whether the digits were numbers. At this point, we simply decided to brute-force the digits:

```
for i in {0000..9999} ; do echo -n $i && ./hurdles 1_kn0w_h0w_$i ; done | less
```

This gave us the solution to stage 1 and 2 (`1_kn0w_h0w_2448`) and the flag for part 1.

After inspecting the solution, we found that the desired value was somewhere hidden in the array. Thus, we also decided to modify our angr script to be able to handle this case:

```
def find_correct_idx_stage2(state):
    base_addr = 0x48b7c0
    correct_val = 13337
    for i in range(0, 0x2710):
        v = state.mem[base_addr+i*2].uint16_t.resolved
        v = state.solver.eval(v)
        if v == correct_val:
            return claripy.BVV(i, 16)
    assert False


@p.hook(0x401d31, length=12)
def handle_identity_array_401d31(state):
    idx_val = state.regs.rax + state.regs.r8 * 2
    actual_val = idx_val // 2

    correct_idx = find_correct_idx_stage2(state)

    state.regs.al = claripy.If(actual_val.get_bytes(6, 2) == correct_idx,
                               claripy.BVV(b'\x01'), claripy.BVV(b'\x00'))
    state.solver.add(state.regs.al == 1)
```

Stage 3

At this point, we continued solving with angr and adjusted the explore code as follows:

Before:

```
# Address after solving stage 2
sm.explore(find=lambda s: s.solver.eval(s.regs.rip) == 0x40085b)
```

After:

```
sm.explore(find=lambda s: s.solver.eval(s.regs.rip) == 0x40085b)

state = sm.found[0]
print(state.posix.dumps(1))

sm = p.factory.simulation_manager(state)
# Address after solving stage 4
sm.explore(find=lambda s: s.solver.eval(s.regs.rip) == 0x400881)
```

We noticed that angr was immediately able to solve stage 3, so we did not even bother looking at the disassembly/decompilation.

Stage 4

Our explore code was already adapted to Stage 4, but our index concretization check fails. Furthermore, decompilation doesn't give us good results, Ghidra fails properly resolving some indirect jumps which seem to be jumptables.

Therefore, our approach was to analyze and model the symbolic indexing. From the min/max values we can infer the used array and can learn that it is a function ptr array which maps a byte value to a function ptr. Furthermore, the array seems to have a different function table for each byte in the input. The dereference of the symbolic index then happens in the following function and instruction:

```
void _copy_val(undefined8 *param_1,undefined8 *param_2)

{
  *param_2 = *param_1;
  return;
}
```

```
MOV         RAX,qword ptr [RDI]
```

As we can see, this is just a copy function.We initially assume this function is only used to dereference the symbolic index (and it actually was!).

Next, we try to model this dereferencing. First, we try to model it by assuming that the dereferenced function ptr is called before the next dereferencing happens. We model this the following way: We remember the base idx of the function table (i.e. which index in the input is used) and set the dereferenced value to a non-mapped value, here `0xaaaa000000`. Then, we write an angr hook where we model the call.

The code for the first part is then:

```
last_funcptr_off = None

@p.hook(0x488ed0, length=3)
def handle_488ed0(state):
    print(f"handling 488ed0, rdi = '{state.regs.rdi}'")

    global last_funcptr_off
    array_off = ((state.regs.rdi - 0x4905f0) // 8)
    last_funcptr_off = ((array_off >> 8) << 16) + (array_off & 0xff)
    state.regs.rax = 0xaaaa000000
```

To model the actual function call, we need to analyze the behavior of the functions in the function table and the differences between individual functions.

For this, let us look at two example functions:

```
undefined FUN_00455220(void)
{
  _DAT_006eef04 = 0x1100ee;
  return DAT_006eef00;
}
```

```
bool FUN_00444800(void)

{
  _DAT_006eef04 = 0xc007a;
  return DAT_006eef00 == '\0';
}
```

We can observe the following:

1. The function writes a value into `_DAT_006eef04`. By further inspecting it, the function writes the following value: `byte_idx * 0x10000 + byte_val`.
2. The function uses `DAT_006eef00` to return either 0 or 1 in the register `AL`. After inspecting the references to `DAT_006eef00`, this value is never modified and therefore will always equal 0.
3. By further inspecting the function tables, we can see that there are two different types of tables:

   3.1. Every function in the table returns 1.

   3.2. Only one function in the table returns 1, every other returns 0.

Therefore, our model works as follows:

1. We compute for each byte index, which byte value will return 1 and store it. If all indices return 1, we store `None`. To achieve this, we run the function with angr and inspect the return value. To prevent re-evaluation of this for each run, we cache the results.
2. We modify `_DAT_006eef04` according to the formula above.
3. We set `AL` to an If-Then-Else Expression depending on the value.

Our angr code is then:

```
def returns_1(function):
    state = p.factory.call_state(function, add_options=
                                  ({angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,
                                    angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS}))
    sm2 = p.factory.simulation_manager(state)
    sm2.run()
    #print(sm2)
    #print(sm2.deadended[0].regs.al)

    assert len(sm2.deadended) == 1
    s = sm2.deadended[0]

    b = s.solver.eval(s.regs.al) == 1
    if b:
        print(s.regs.al)
    return b
```

```python
al_map = {}

def _compute_al(state, idx):
    al = None

    for i in range(0x20, 0x7f):
        func_address  = state.mem[0x4905f0 + (idx * 256 + i) * 8].uint64_t.resolved
        if returns_1(func_address):
            if al:
                print(f"Multiple return 1s for idx = {idx}")
                return None
            al = i
            continue

        # Shortcut if > 1 solution is correct - HEURISTIC!
        if al:
            return al


def pre_compute_als():
    global al_map
    if os.path.exists('als.json'):
        # load from cache
        with open('als.json', 'r') as f:
            _json = json.load(f)
            al_map = dict(map(lambda x: (int(x[0]), x[1]), _json.items()))
        return
    state = p.factory.blank_state(add_options=
                                  ({angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,
                                    angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS}))
    for idx in range(len(flag_chars)):
        al_map[idx] = _compute_al(state, idx)
    with open('als.json', 'w') as f:
        json.dump(al_map, f)

pre_compute_als()

def compute_al(state, idx):
    return al_map[state.solver.eval(idx)]

@p.hook(0xaaaa000000)
def handle_funcptr_call(state):
    global last_funcptr_off
    print(f"funcptr call with last_funcptr_off = {last_funcptr_off}")
    # set val
    assert last_funcptr_off is not None
    state.mem[0x6eef04].uint32_t = last_funcptr_off.get_bytes(4, 4)
    print(state.mem[0x6eef00].uint32_t)

    al_val = compute_al(state, last_funcptr_off >> 16)

    print(al_val)

    if al_val is None:
        state.regs.al = 1
    else:
        state.regs.al = claripy.If(last_funcptr_off.get_bytes(7, 1) == al_val,
                                   claripy.BVV(b'\x01'), claripy.BVV(b'\x00'))

    print(state.regs.al)

    last_funcptr_off = None

    # return
    ret_addr = state.mem[state.regs.rsp].uint64_t.resolved
    print(f"returning to: {ret_addr}")
    state.regs.rip = ret_addr
    state.regs.rsp += 8
```

Using this model, angr is able to successfully retrieve the correct input (and flag): `CSCG{1_kn0w_h0w_2448_0bfu5c4710n_w0rk5!}`

Mitigations

As this is about reverse-engineering, there is no real mitigation. Reversing can always be made harder, but it wouldn't completely prevent reversing. To make symbolic execution harder and possibly fail on path explosion, the program should make use of constraints where checking requires non-trivial loops that depend on the input values. Then, symbolic execution will most likely fail due to path-explosion and would require more elaborate modeling.

Complete Code

```
#!/usr/bin/python3

import angr
import claripy
import multiprocessing
import os
```

```python
import json

sm = None

base_addr = 0x400000
main_function = base_addr + 0x810
p = angr.Project("hurdles", auto_load_libs=False)

flag_chars = [claripy.BVS('flag_%d' % i, 8) for i in range(34)]

class ConstArraySimConcretizationStrategy \
    (angr.concretization_strategies.SimConcretizationStrategy):
    """
    Concretization strategy that returns all solutions
    if less than 10 solutions exist, otherwise throw error.
    """
    def _concretize(self, memory, addr, **kwargs):
        #print(f"Concretizing memory addr {addr}")
        _min = self._min(memory, addr, **kwargs)
        _max = self._max(memory, addr, **kwargs)
        #print(f"Min = {_min:x}, Max = {_max:x}")
        addrs = self._eval(memory, addr, 10)
        if len(addrs) == 10:
            print(addrs)
            for addr in sm.active[0].history.bbl_addrs: print(hex(addr))
            assert False
        #print(list(map(hex, addrs)))
        return addrs

def print_concretized_address(state):
    print(f"Concretizing address at {state.regs.rip}")
    if state.solver.eval(state.regs.rip) == 0x401d31:
        print(f"Concretizing address rax = {state.regs.rax}")
        print(f"Concretizing address r8 = {state.regs.r8}")


argv_addr  = 0xdead0000
argv1_addr = 0xdead0020
argv2_addr = 0xdead0040

state = p.factory.call_state(main_function, 2, argv_addr, add_options=
                              ({angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,
                                angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS}))

state.memory.read_strategies.insert(0, ConstArraySimConcretizationStrategy())
state.inspect.b('address_concretization', when=angr.BP_BEFORE,
                  action=print_concretized_address)

flag = claripy.Concat(*flag_chars + [claripy.BVV(b'\x00')])

state.mem[argv_addr].uint64_t = argv1_addr
state.mem[argv_addr+8].uint64_t = argv2_addr
state.mem[argv_addr+0x10].uint64_t = 0
state.memory.store(argv1_addr, b"./linear_code\x00")
state.memory.store(argv2_addr, flag)




def find_correct_idx_stage2(state):
    base_addr = 0x48b7c0
    correct_val = 13337

    for i in range(0, 0x2710):
        v = state.mem[base_addr+i*2].uint16_t.resolved
        v = state.solver.eval(v)
        if v == correct_val:
            return claripy.BVV(i, 16)
    assert False


@p.hook(0x401d31, length=12)
def handle_identity_array_401d31(state):
    idx_val = state.regs.rax + state.regs.r8 * 2
    actual_val = idx_val // 2

    correct_idx = find_correct_idx_stage2(state)

    state.regs.al = claripy.If(actual_val.get_bytes(6, 2) == correct_idx,
                               claripy.BVV(b'\x01'), claripy.BVV(b'\x00'))
    state.solver.add(state.regs.al == 1)


def returns_1(function):
    state = p.factory.call_state(function, add_options=
                                 ({angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,
                                   angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS}))
    sm2 = p.factory.simulation_manager(state)
    sm2.run()

    assert len(sm2.deadended) == 1
    s = sm2.deadended[0]
    b = s.solver.eval(s.regs.al) == 1
    if b:
        print(s.regs.al)
    return b


al_map = {}

def _compute_al(state, idx):
    al = None

    for i in range(0x20, 0x7f):
        func_address  = state.mem[0x4905f0 + (idx * 256 + i) * 8].uint64_t.resolved
        if returns_1(func_address):
            if al:
                print(f"Multiple return 1s for idx = {idx}")
                return None
            al = i
            continue

        # Shortcut if > 1 solution is correct - HEURISTIC!
        if al:
            return al


def pre_compute_als():
    global al_map
    if os.path.exists('als.json'):
        # load from cache
        with open('als.json', 'r') as f:
            _json = json.load(f)
            al_map = dict(map(lambda x: (int(x[0]), x[1]), _json.items()))
        return
    state = p.factory.blank_state(add_options=
                                  ({angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,
                                    angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS}))
    for idx in range(len(flag_chars)):
        al_map[idx] = _compute_al(state, idx)
    with open('als.json', 'w') as f:
        json.dump(al_map, f)

pre_compute_als()

def compute_al(state, idx):
    return al_map[state.solver.eval(idx)]
```

```python
last_funcptr_off = None
@p.hook(0xaaaa000000)
def handle_funcptr_call(state):
    global last_funcptr_off

    print(f"funcptr call with last_funcptr_off = {last_funcptr_off}")
    # set val
    assert last_funcptr_off is not None
    state.mem[0x6eef04].uint32_t = last_funcptr_off.get_bytes(4, 4)
    print(state.mem[0x6eef00].uint32_t)

    al_val = compute_al(state, last_funcptr_off >> 16)

    print(al_val)

    if al_val is None:
        state.regs.al = 1
    else:
        state.regs.al = claripy.If(last_funcptr_off.get_bytes(7, 1) == al_val,
                                   claripy.BVV(b'\x01'), claripy.BVV(b'\x00'))

    print(state.regs.al)

    last_funcptr_off = None

    # return
    ret_addr = state.mem[state.regs.rsp].uint64_t.resolved
    print(f"returning to: {ret_addr}")
    state.regs.rip = ret_addr
    state.regs.rsp += 8



@p.hook(0x488ed0, length=3)
def handle_488ed0(state):
    print(f"handling e88ed0, rdi = '{state.regs.rdi}'")

    global last_funcptr_off
    array_off = ((state.regs.rdi - 0x4905f0) // 8)
    last_funcptr_off = ((array_off >> 8) << 16) + (array_off & 0xff)
    state.regs.rax = 0xaaaa000000


for flag_char in flag_chars:
    if isinstance(flag_char, int):
        continue
    state.solver.add(flag_char >= 0x20)
    state.solver.add(flag_char <= 0x7f)

sm = p.factory.simulation_manager(state)


sm.explore(find=lambda s: s.solver.eval(s.regs.rip) == 0x40085b)

state = sm.found[0]
print(state.posix.dumps(1))


sm = p.factory.simulation_manager(state)

sm.explore(find=lambda s: s.solver.eval(s.regs.rip) == 0x400881)


if sm.errored:
    print(sm.errored)

for s in sm.found:
    print(s.regs.al)
    print(s.regs.eax)
    print(s.regs.rax)
    _flag_out = b""
    for flag_char in flag_chars:
        _flag_out += bytes([s.solver.eval(flag_char)])
    print(_flag_out)
    print(s.posix.dumps(1))

for s in sm.deadended:
    print("deadended at:")
    print(s.regs.rip)

print(sm)
```