Hurdles (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? Difficulty: Medium

Hurdles was a two part reversing challenge from the CSCG Qualifications 2023. This writeup is for the first part.

Overview (Ghidra)

Simply running the program gives:

```
$ ./hurdles
Bad input
```

Opening the binary in Ghidra we can get an overview of the flag checker. The main function decompiles as follows (function names were changed):

```
undefined8 main(int argc,char **argv) {
  char cVar1;
  uint uVar2;
  undefined8 uVar3;
  ulong uVar4;

  cVar1 = check_arg_and_length();
  if ((cVar1 != '\0') && (uVar2 = stage1(argc,argv), (char)uVar2 != '\0')) {
    puts("You have completed stage 1");
    uVar3 = stage2(argc,argv);
    if ((char)uVar3 != '\0') {
      puts("You have completed stage 2");
      FUN_0048ab10(argc,argv);
      uVar4 = stage3(argc,argv);
      if ((char)uVar4 != '\0') {
        puts("You have completed stage 3");
        cVar1 = stage4(argc,argv);
        if (cVar1 != '\0') {
          puts("You have completed stage 4");
          printf("Here is your flag: cscg{%s}\n",argv[1]);
          return 0;
        }
      }
    }
  }
  puts("Bad input");
  return 0xffffffff;
}
```

`check_arg_and_length` does only very basic checks:

```
ulong check_arg_and_length(int argc,char **argv) {
  ulong uVar1;

  if (argc == 2) {
    uVar1 = strlen(argv[1]);
    return uVar1 & 0xffffffffffffff00 | (ulong)(uVar1 < 0x23);
  }
  return 0;
}
```

`strlen` is easily identifiable, because of it's decomplation.

That means, in order to pass `check_arg_and_length` we have to supply one argument to the program, which has to be shorter than `0x23` characters.

Stage 1 (Angr)

Next up is `stage1`, which can actually still be reversed using Ghidra, but let's use angr, as it will be helpful during the rest of the challenge. More precisely, I will be using angrs symbolic execution feature. You can imagine it like a simulator, which discovers restrictions on your input, that have to be fulfilled, to reach certain code paths. It can then also generate possible inputs which match these requirements (much like `z3`).

To solve `stage1` we can use the following script:

```
import angr
import claripy

proj = angr.Project("./hurdles", auto_load_libs=False)

proj.hook(0x06ef028, angr.SIM_PROCEDURES['glibc']['__libc_start_main']())
proj.hook(0x06ef038, angr.SIM_PROCEDURES['libc']['free']())
proj.hook(0x06ef020, angr.SIM_PROCEDURES['libc']['malloc']())
proj.hook(0x06ef000, angr.SIM_PROCEDURES['libc']['printf']())
proj.hook(0x06ef010, angr.SIM_PROCEDURES['libc']['puts']())

arg1 = claripy.BVS('arg1', 8*0x22)

initial_state = proj.factory.entry_state(args=["./hurdles", arg1])

sm = proj.factory.simulation_manager(initial_state)
sm.explore(find=0x0400834, avoid=0x040089f)

found = sm.found[0]

print(found.solver.eval(arg1, cast_to=bytes))
```

It first loads the `hurdles` executable, but without loading external libs like `libc`. That means we have to hook the `libc` functions with python code that emulates the behavior. This is done to avoid tracking unnecessary restrictions on the input and to keep the amount of simulated instructions low.

Then we define our input as an unrestricted vector of `0x22` bytes, which can be used to build an simulation state, that executes the binary from the entry point.

Then we simulate the code until we hit the address `0x0400834` (`puts("You have completed stage 1")`) without hitting `0x040089f` (`puts("Bad input");`).

We will find a simuation state, that matches that criterion. Evaluating the conditions on `arg1` leading to this code path can be done using the states `solver` attribute.

In the end the script prints: `b'1_kn0w_h0w_\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'`

Running `./hurdles 1_kn0w_h0w_`, we can check that we have actually passed stage 1.

Stage 2 (Angr + GDB)

Sadly we cannot simply copy the strategy from `stage1` to `stage2`.

```
import angr
import claripy

proj = angr.Project("./hurdles", auto_load_libs=False)

proj.hook(0x06ef028, angr.SIM_PROCEDURES['glibc']['__libc_start_main']())
proj.hook(0x06ef038, angr.SIM_PROCEDURES['libc']['free']())
proj.hook(0x06ef020, angr.SIM_PROCEDURES['libc']['malloc']())
proj.hook(0x06ef000, angr.SIM_PROCEDURES['libc']['printf']())
proj.hook(0x06ef010, angr.SIM_PROCEDURES['libc']['puts']())

# Further optimization: make strlen simply return 0x22 and bypass `check_arg_and_length`
proj.hook(0x04007e0, angr.SIM_PROCEDURES['stubs']['ReturnUnconstrained'](return_val=0x22))
proj.hook(0x04008c0, angr.SIM_PROCEDURES['stubs']['ReturnUnconstrained'](return_val=0x1))


arg1 = claripy.BVS('arg1', 8*0x22)

initial_state = proj.factory.entry_state(args=["./hurdles", arg1])

sm = proj.factory.simulation_manager(initial_state)
sm.explore(find=0x0400834, avoid=0x040089f).unstash(from_stash='found', to_stash='active')
sm.explore(find=0x040084c, avoid=0x040089f)

print(sm)
```

Executing this script yields `<SimulationManager with 16 avoid>`, so no code paths, that reach `0x040084c` (`puts("You have completed stage 2")`), have been found.

Decompiling the `stage2` function in Ghidra shows, that the following check has to pass in order for the function to return true.

```
// [...]
uVar2 = CONCAT71(0xa6e3a29dbfb830,
                 *(short *)(((ulong)(byte)flag[0xb] * 1000 + 0x2c8e2eb120231781 +
                             (ulong)(byte)flag[0xc] * 100 + (ulong)(byte)flag[0xd] * 10
                            + (ulong)bVar3) * 2 + -0x591c5d623fff17e2) == 0x3419);
// [...]
return uVar2;
```

Assembly:

```
00401d31  CMP        word ptr [RAX + R8*0x2 + 0x048b7c0],CX
00401d3a  SETZ       AL
//  [...]
00402103  RET
```

So let's try to find a code path to the address of the instruction, that is responsible for this compare (`0x0401d31`) with angr. Indeed a path is found and if we evaluate the restrictions on the input, we get:

```
>>> sm
<SimulationManager with 1 found, 15 avoid>
>>> found = sm.found[0]
>>> found.solver.eval(arg1, cast_to=bytes)
b'1_kn0w_h0w_8012\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

Generating further examples is also possible:

```
>>> found.solver.eval_upto(arg1[-15 * 8:], 10, cast_to=bytes)
[b'1_kn0w_h0w_6090', b'1_kn0w_h0w_8810', b'1_kn0w_h0w_8490', b'1_kn0w_h0w_6010', b'1_kn0w_h0w_8850', b'1_kn0w_h0w_0011', b'1_kn0w_h0w_8090', b'1_kn0w_h0w_8830', b'1_k
```

Overall it looks like the next four bytes have to be a decimal number. The ghidra decomplation also hints at this, as flag bytes are being multiplied by powers of 10.

To fully understand what's going on, I created a breakpoint at `0x0401d31` in gdb and ran the program with one of the example inputs (`1_kn0w_h0w_6090`).

In this case `rax + 2 * r8` evaluates to `12180` and `cx` contains `13337`. So the 16-bit value in memory at `0x048b7c0 + 12180` should contain `13337` to pass the check. (`CMP word ptr [RAX + R8*0x2 + 0x048b7c0],CX`)

One might notice, that `12180 = 2 * 6090` is twice the value passed in using the argument. That means, that the `stage2` function is mainly about parsing the decimal number in the argument.

Searching the memory for two bytes after `0x048b7c0` that contain `13337` one can find `0x48cae0`. An input of `(0x48cae0 - 0x048b7c0) / 2 = 2448` should therefor pass the check.

And indeed:

```
$ ./hurdles 1_kn0w_h0w_2448
```

```
You have completed stage 1
You have completed stage 2
You have made it to the interim flag: cscg{y4y_1_50lv3d_7h3_f1r57_h4lf}
Bad input
```