Toolbox for reverse engineering and binary exploitation

2024-05-17

About me

Was system administrator at McGill Genome Center

Joined ETTIC team at Desjardins in 2016

Did infrastructure testing

Then application testing

And now doing vulnerability research in IOT devices.

Challenges encountered in past engagements

- Emulation of a code block, function or even part of a function from a binary
- Search for unknowns during static analysis, for example the contents of some functions arguments
- Software emulation outside of the device, sometimes I don't have physical access to the hardware
- Emulation of an entire operating system, when several service are closely tied to each other by sockets and/or pipes

Goals of the workshop

Help others solve those challenges.

Share knowledge of the tools like:

- How to use them
- Where to look to learn more advanced usage
- When to use them
- What are the tool's limits and how to overcome them

Tools

- Angr
- Unicorn
- Qiling
- quick overview of Qemu and Renode

Setting things up

Clone the repository

```
git clone --recurse-submodules FIXME missing repo url
```

A dockerfile is provided to create a container with most software and files readily avaiable.

Jupyter will be running so you can access the challenges inside the container from a browser.

Only `Ghidra` or a similar tool needs to be run outside the container.

Build the docker container:

```
docker build --rm --tag workshop2024 .
```

This will take a while ...

To start the docker container, use docker run as follows:

```
docker run -it --rm -p 8888:8888 workshop2024
```

Angr

Angr is a tool that allows you to translate compiled code into an intermediate language and then use a form of emulation called symbolic execution.

Machine language (ARM, PowerPC, etc.) -> Intermediate language -> Symbolic execution -> constraint resolution

Angr life cycle

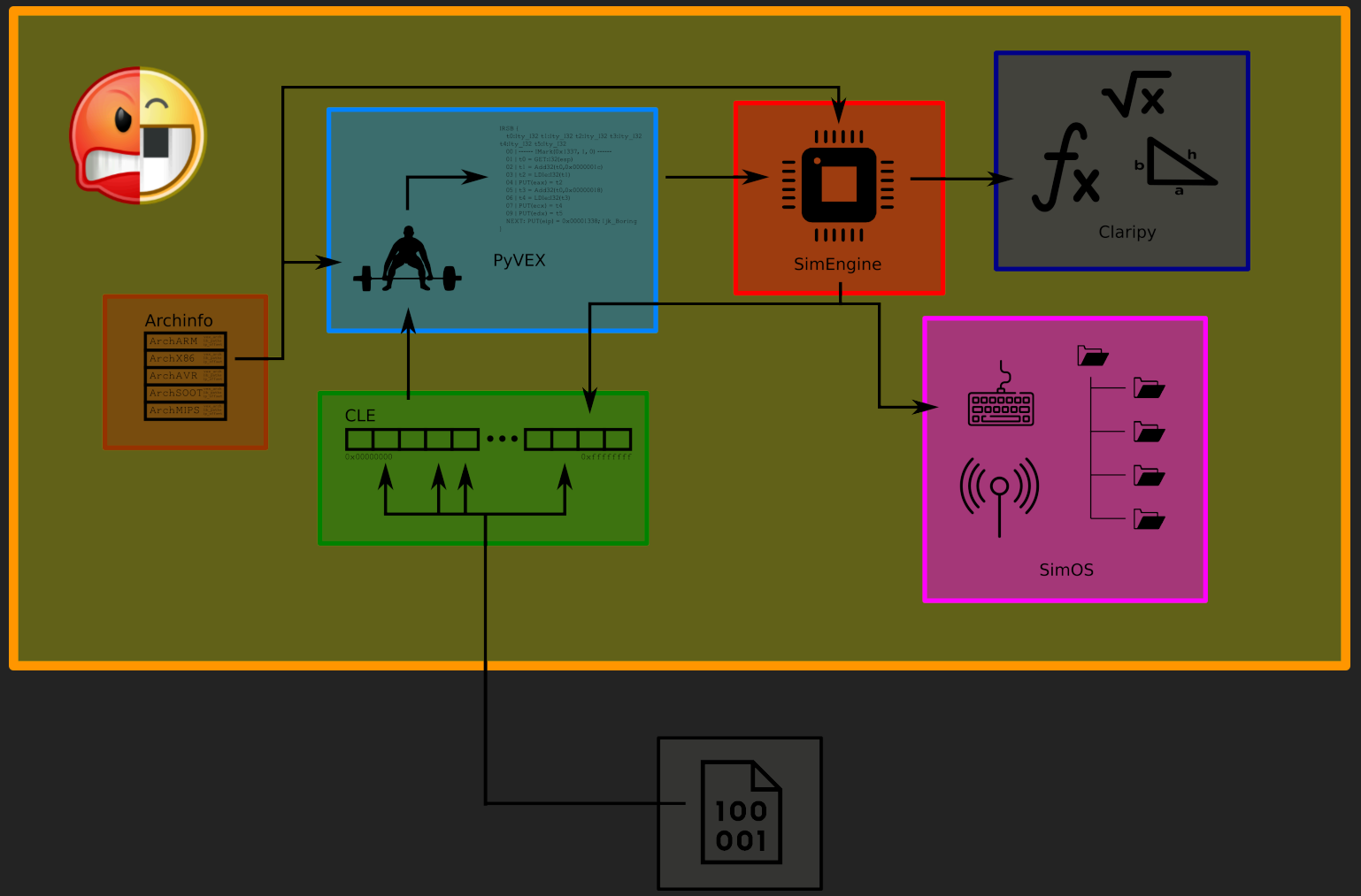


Image taken from Angr blog post here: https://angr.io/blog/throwing_a_tantrum_part_1/

Symbolic execution in my own words ...

As I understant it, symbolic execution allows code to be treated as a series of mathematical equations to be solved.

We assign certain inputs as being symbolic, i.e. variables that **Angr** can manipulate to try to solve the equations.

We define one or more constraints, for example:

- A particular address where you wish to go
- One or more addresses to avoid (eg: printf("try again"))
- A specific value that we want to see in a register at a specific time in execution

We can also define constants, i.e. concrete values. This is used to define the initial state at the start of the emulation.

For example, we may need to set the stack pointer when a fragment of an executable program is to be emulated.

Angr will try to find values to assign to the symbolic variables which allow all of the stated constraints to be satisfied.

If it is impossible to satisfy all the constraints, **Angr** raises an exception.

First Angr challenge

Consider the following example. Imagine a crackme like challenge that is doing math on some inputs and prints the flag if you have it right but would fail by default.

```
void solve(int x) {

   int y = x * 8 + 3;

   if (y == 35) {
     printf("correct answer");
   } else {
     printf("Try again");
   }
}
```

Here `x` must be defined as symbolic, we want **Angr** to find for us a value of `x` which allows to satisfy the constraint.

Two states would be created when if statement is reached, one for x when y = 35 and one other for all values of `x` which makes `y` not equal to 35.

As we want to force the program to show us "correct answer", `y == 35` will be the constraint.

The `printf` statement that says "Try again" should also be avoided. Therefore, we could set `avoid "Try again" printf` as an additional constraint. **Angr** should then converge more quickly towards a solution.

If all goes well, **Angr** should be able to find a value of `x` which allows all constraints to be satisfied.

The last this we need to do is "concretizing" symbolic values we are interested in. In this case, asking **Angr** for the value of `x` which allow the equation to be satisfied. The concrete value thus obtained should be `4`.

Now how to get Angr to do the math for us

- Open `angr/easymath` in `Ghidra`
- Also open `angr/easymath.py` in `Jupyter`

First question, where `Angr` should start symbolic execution ?

Sometimes, `Angr` can just start at the entry point like the program normally does.

```
initial_state = project.factory.entry_state()
```

But in this case, we would like to start at function solve() as like it would be a critical part of a much larger binary. Use `gridra` to get the address of function `solve`.

```
  start_address = 0x00401139
  initial_state = project.factory.blank_state(addr=start_address)
```

Careful about addresses

Modern binaries are position independent (`PIE`) so `ASLR` can work. That means that addresses you see in your favourite reverse engineering tool may not match the ones `Angr` would use in its simulation manager.

Of course, simulation manager would fail to reach a solution if it ain't using the addresses you expect.

How to make sure addresses do match?

First, one could make `Angr` loader print its memory map as described in the documentation

```
print(project.loader)
```

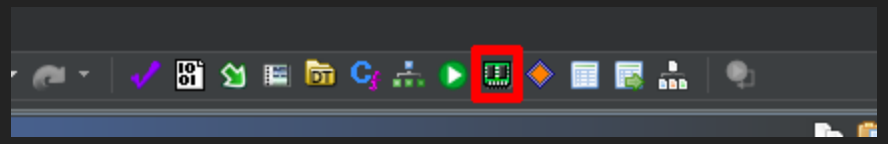
From that print statement, one should see output similar to this

```
<Loaded easymath, maps [0x400000:0xa07fff]>
```

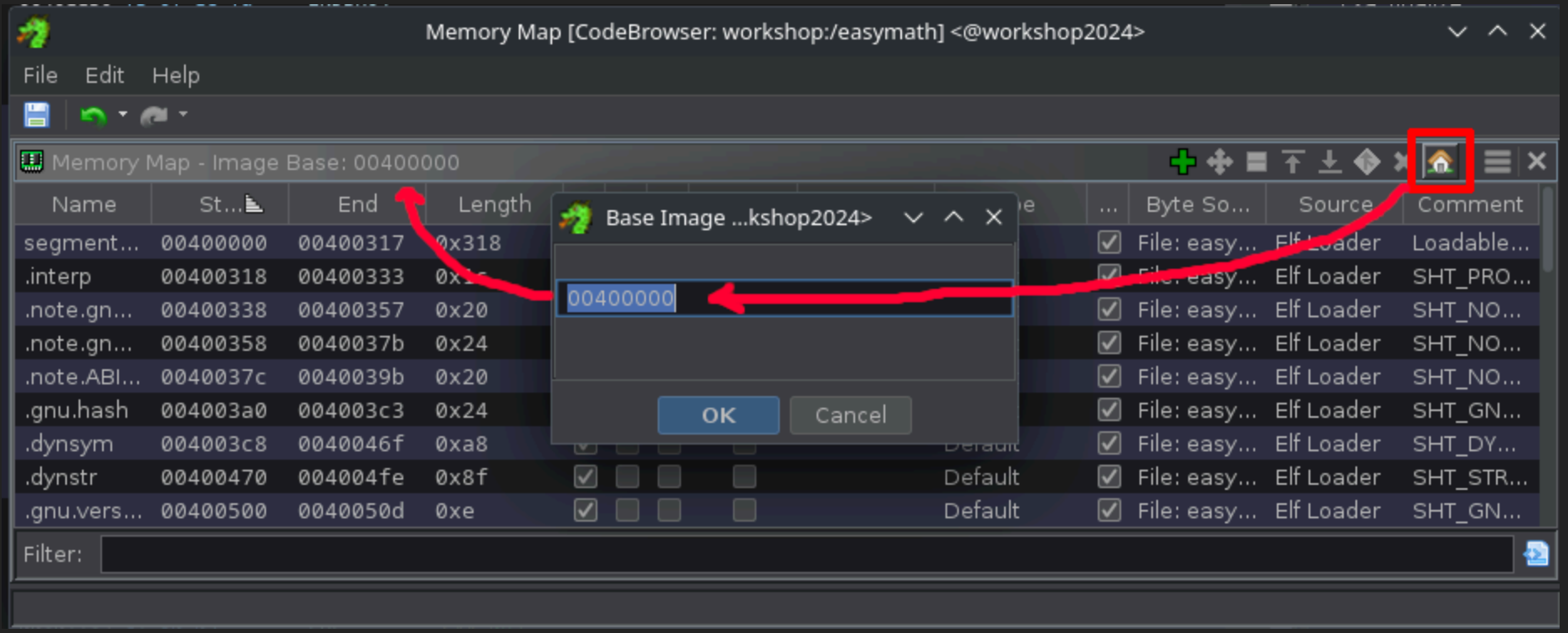
Angr's loader base address should be 0x400000. Then, set the base address in your reverse engineering tool to match.

Setting the base address of a binary in Ghidra

To set the base address of a binary in `Ghidra`, click on the memory dimm icon to bring the memory map



Then click on the house icon to set the base address so it matches the one `Angr` uses



Assign inputs as symbolic so Angr can play with them

The `solve()` function takes only one argument which determine the outcome, a good candidate to tag as symbolic

If you observe the disassembly listing, the content of the `EDI` register is being loaded on the stack, that is our input.

Therefore we need to assign `EDI` as symbolic

```
00401141 89 7d ec        MOV        dword ptr [RBP + local_1c],EDI
```

First create a 32 symbolic bit vector named x

```
x = claripy.BVS('x', 32)
```

Then we load `x` into the `EDI` register

```
initial_state.regs.edi = x
```

Starting Angr

Once the initial state is set the starting address and any concrete/symbolic values you may need to set, simulation can start

This snippet would start the simulation manager with the initial state provided:

```
simulation = project.factory.simgr(initial_state)
```

Of course, you may want to tell the simulation manager where to go and where not to.

Managing the simulation manager

- Here is how we tell the manager to stop when "correct answer" is seen on `stdout` and to avoid any path that would lead to `Try again` being printed on `stdout`.

```
def is_successful(state):
    stdout_output = state.posix.dumps(sys.stdout.fileno())
    return "correct answer" in stdout_output

  def should_abort(state):
    stdout_output = state.posix.dumps(sys.stdout.fileno())
    return "Try again" in stdout_output

  simulation.explore(find=is_successful, avoid=should_abort)
```

Note that one could give an address to the `find` parameter and a list of addresses to the `avoid` parameter if the program doesn't send anything to `stdout`.

Concretize symbolic values

Let's say the simulation manager found an input which allowed to reach stated destination. Then, for a given solution (yes, Angr can find more than one), we want to know which input value would yield that solution. The process of converting symbolic to concrete values for a given solution is achived by the `eval()` method

That will allows us to recover which value of `x` would make the program print `correct answer`. Then, we would know how to patch it so it prints `correct answer` insteat of `try again` when run.

```
solution_state.solver.eval(x,cast_to=int)
```

Try it

In `Ghidra`, open `angr/easymath`

In `Jupyter`, open `angr/easymath.py`

Run the python script to see the results

Second practical example, Angr CTF challenge #3

Have a look at Jakespringer's Angr CTF:

- https://github.com/jakespringer/angr_ctf

We will do a few challenges pulled from Angr CTF here but of course you should do all of it.

Challenge #3 is about how to handle multiple local variables populated by a scanf call.

- Go into `Jupyter`, open `angr/angr_ctf/dist/scaffold03.py`
- Then outside the docker container, open angr/angr_ctf/dist/03_angr_symbolic_registers with `Ghidra`
- To run the binary, you may go to angr/angr_ctf/dist/ in a `Jupyter` shell and run it: ./03_angr_symbolic_registers

State creation

Imagine that you are at the controls of **gdb** doing **step, step, step...**

You encounter a conditional branch (if) which depends on a variable that is of interest to you (symbol).

"**What if ...**", you could duplicate yourself, create an identical clone of yourself, to explore each of the branches in parallel?

`Angr` lets you do just that. When a branch that depends on a variable marked as symbolic is encountered, **Angr** creates two states to explore each of the branches.

State explosion and limitations of Angr

As whenever such a fork in the road is encountered, states are created to explore each possibility. Unfortunately, this comes with a cost.

For instance, if the program goes through a `for` loop with an `if` statement within it, each iteration would create 2 states.

If the program need to go through the loop more than a few times, this could lead to state explosion.

Looking at the following code:

```
void obfuscate(int x, char *in, char *out) {

   for(i=0; i<4096;, i++) {
     if (x < 256) {
       out[i] = in[i] ^ x;
     } else {
       out[i] = in[i] ^ (x & 0xff)
     }
   }
}
```

We see each time the condition is encountered in the loop, two new states are created. After an exponential rise of the number of states, **Angr** risks getting your computer bogged down.

Fortunately, there is a few strategies to avoid state explosion

Third practical example, Angr CTF challenge #8

- This one will teach us one trick to avoid state explosion: checking symbolic input against reference data (if known) instead of simulating through the function which does that check.
- This challenge also show us how to add additionnal constraints on inputs on top of being able to reach a specific address.

Load `angr/angr_ctf/dist/scaffold08.py` in Jupyter

Hooks and Simprocedures

What if you need to:

- You need to jump over a piece of code?
- Or replace it?
- Replace all library functions?

Of course it can be done!

See Angr documentation about hooks and Simprocedures

Forth example, Angr CTF challenge #10

The next challenge is about Simprocedures, which allow us to reimplement functions in Python. This will be used in automatic exploit generation script so we have a quick look here

This challenge also contains multiple references to challenge #9 about hooks:

- https://github.com/jakespringer/angr_ctf/blob/master/solutions/09_angr_hooks/solve09.py

Note: Your Simprocedure will allow you to receive arguments and return like you would normally do in Python, no assembly required

Note 2*: return like Angr CTF challenge #9

```
    return claripy.If(
      user_input_string == check_against_string,
      claripy.BVV(1, 32),
      claripy.BVV(0, 32)
    )
```

Open `angr/angr_ctf/dist/scaffold10.py` in Jupyter to take the challenge

Automatic exploit generation

Yes, Angr can do the memory corruption thing for you!

So far we used the simulation manager to reach a specified address, the precise moment a specific string is sent to stdout.

But!

What if we could tell the simulation manager to keep running until:

- a memory spot get overwritten with some bytes we set
- a register is being set to a value we chose

What could possibly go wrong ?

How Angr get you a shell?

A few steps:

- First, we need to use radare2 to automatically analyze binary, feed Angr the address of target function
- Use of project.analyses.StaticHooker to load `libc` and use simprocedures to avoid using symbolic executions on libc functions.
- Step through the vulnerable function and test if instruction pointer (pc register) can by made symbolic by manipulating inputs at each step
- If instruction pointer can by made symbolic, add constraint where pc equals to some specific value
- If that constraint can be satisfied, locate that specific value in inputs to figure out how much data required to overflow buffer
- build a rop chain to take over the program
- Profit !!

AEG challenge 1: Buffer overflow with easy ROP chain.

Basic challenge i made myself just to see if I got this thing right

Source code is provided: `angr/aeg/pwnmemaybe.c` in Jupyter

Open `angr/aeg/pwnmemaybe` in `Ghidra`

Follow steps in `angr/aeg/pwnmemaybe.py`

AEG challenge 2: NSEC2023 surveillance.ctf track last challenge

Scenario: Some co-conspirator send you firmwares of cameras over a network service and you must automatically reverse the binary and return an exploit that would disable the cameras over the same network service.

The third and last challenge on that track requires automatic exploitation of a buffer overflow.

Load `angr/aeg/nsec2023.py` in Jupyter

Unicorn

Unicorn is a CPU emulator

Supports multiple architectures: ARM, X86, MIPS, PowerPC and even systemZ

Based on Qemu

Allows dynamic analysis of an executable file or even just a fragment of it.

Simple example

Hello world in Unicorn

Taken from Python tutorial: https://www.unicorn-engine.org/docs/tutorial.html

Load `unicorn/example1.py` in Jupyter

Unicorn Features

- Possibility of emulating binaries (ELF, PE, etc.), functions or code blocks.
- Hijacking (hooking) of functions
  - Can be used to set breakpoints
  - Allows you to bypass or reimplement a problematic function or piece of code (e.g. printf)
- Allows instruction tracing: **Unicorn** can do "step, step, step" for us

Using both Angr and Unicorn in reverse engineering

It is possible to combine **Angr** and **Unicorn** in order to combine the strengths and circumvent the limitations of these two tools

Either:

- Use of **Angr** for constraint resolution. For example, finding for which input(s) we obtain the desired output value for a function.
- Using **Unicorn** to emulate parts of a program that result in state exploitation in **Angr**

Deofuscation challenge

You may need to remove layers of obfuscation to assess a binary and to do so, you may more than one tool at once.

Let's start with the following obfuscated string:

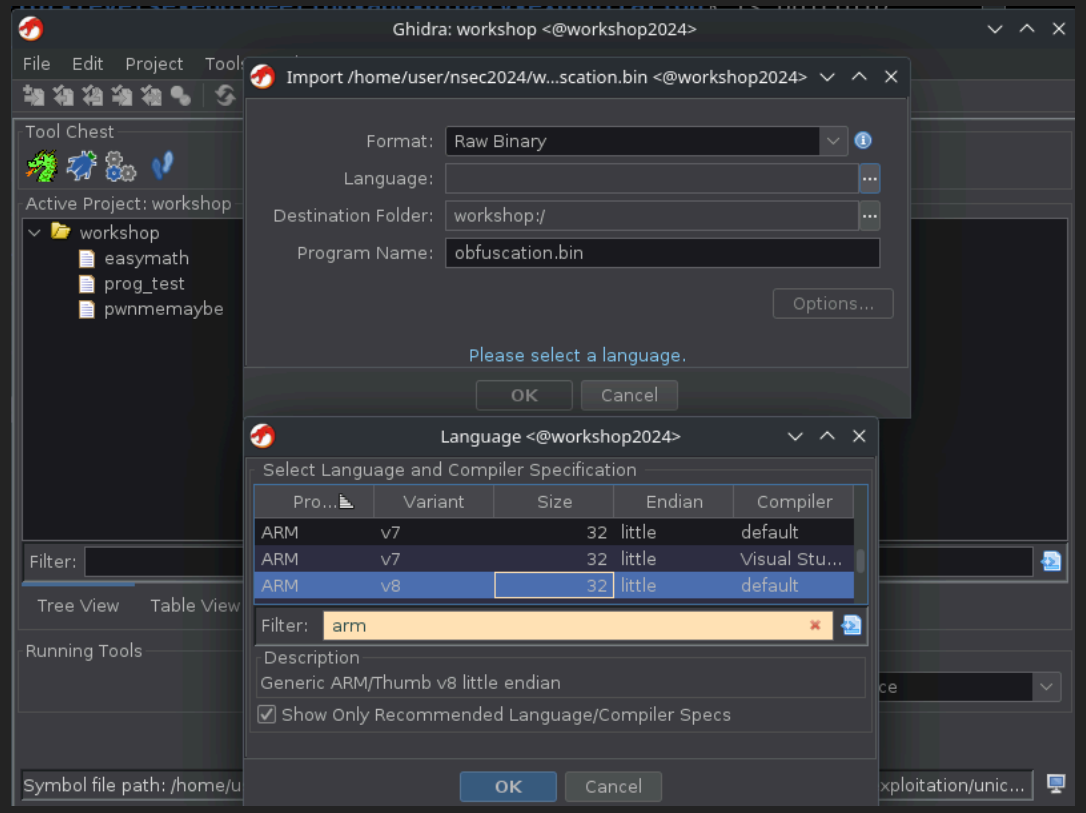
`CfX5cDTu10UkytBYP5mKB32Nghcb1tdbLdbAFynWyhc93M5SPcrdFyrX314h3N1FJtffFy7X3Bct0NZWPcCY UjfJ1FgmzdlDJtbWSg==`

The challenge is to recover the full plaintext of that obfuscated string, knowing that it start with `FLAG` in uppercase.

And let's say you only managed to partially extract the device firmware. Luckily, you have the relevant part, the decryption function in the `obfucation.bin` file.

Open `unicorn/obfuscation.bin` in `Ghidra`, load as:

- Type: Raw binary
- Language: ARMv8, 32 bits, litte endian



In `Ghidra`, set load address at 0x10000. Of course, you could pick any other address, you just have to use the same address in your deobfuscation script.

Click `yes` when prompt to do the analysis

Set cursor to the start of the listing to get proper decompilation output.

As we work with a raw binary file, Ghidra has a harder time handling it.

Or click on the first function in the function list.

Static analysis

From Ghidra's decompiler, we learn the following about the decryption function:

- It receives 2 arguments
- It does some byte shifting on `param2`
- It has a while loop that does `XOR` and math operations on `param1`
- It calls 2 other functions that we don't have access to
- We would like to know what `param1` look like just before the second unaccessible function is called

Our strategy

- Since we know the start of the plaintext: `FLAG`, use `Angr` to try and recover the key
- Due to state explosion, `Angr` will use a lot of time to recover the whole string.
- So, once we can get a key that works with the first four bytes, we switch to `Unicorn` to recover the remaining plaintext

Load `deobfuscate.py` in Jupyter

Real life firmware challenge: HP iLO Generation 3, version 1.20

The goal here is to use `Unicorn` to run part of the bootloader in the `HP` `iLO3` firmware to gain access to the uncompressed "root filesystem" (mind the quotes here).

Of course, such firmware could not be distributed as part of this workshop, although a script has been included to help you download it

Getting HP iLO3 v1.20 firmware

From `Jupyter`, run the `getilo.sh` script.

The script should download the Linux installer from `HP` web site and extract the `ilo3_120.bin`, which is the file we need for this challenge

You will need to extract the `ilo3_120.bin` from the docker container to load it into `Ghidra`.

To do so, you have a few options:

- Run get `getilo.sh` script outside the container
- Share a folder from your host with the docker container using `docker run -v` option https://www.digitalocean.com/community/tutorials/how-to-share-data-between-the-docker-container-and-the-host

- Pull the file from `Jupyter` using `curl`:

```
curl -H "Authorization: Token ${yourtoken}" http://127.0.0.1:8888/files/unicorn/ilo3/ilo3_120.bin --output ilo3_120.bin
```

Static analysis of the flash image

Open `ilo3_120.bin` in `Ghidra`, set language as `ARM v8 32 bits little endian`.

You should see functions starting from address **0x1755f4**

Functions - 86 items

Name	Location	Function Signature	Function Size
FUN_001755f4	001755f4	undefined FUN_001755f4()	12
FUN_007e2cbe	007e2cbe	undefined FUN_007e2cbe()	1
FUN_007f0630	007f0630	undefined FUN_007f0630()	48
FUN_007f0660	007f0660	undefined FUN_007f0660()	104
FUN_007f06ec	007f06ec	undefined FUN_007f06ec()	16
FUN_007f06fc	007f06fc	undefined FUN_007f06fc()	28
FUN_007f0834	007f0834	undefined FUN_007f0834()	44
FUN_007f0858	007f0858	undefined FUN_007f0858()	32
FUN_007f08a4	007f08a4	undefined FUN_007f08a4()	24
FUN_007f08c4	007f08c4	undefined FUN_007f08c4()	28
FUN_007f08e0	007f08e0	undefined FUN_007f08e0()	44
FUN_007f090c	007f090c	undefined FUN_007f090c()	16
FUN_007f091c	007f091c	undefined FUN_007f091c()	16
FUN_007f092c	007f092c	undefined FUN_007f092c()	16
FUN_007f093c	007f093c	undefined FUN_007f093c()	20
FUN_007f0994	007f0994	undefined FUN_007f0994()	64
FUN_007f09d4	007f09d4	undefined FUN_007f09d4()	24
FUN_007f09ec	007f09ec	undefined FUN_007f09ec()	24
FUN_007f0a04	007f0a04	undefined FUN_007f0a04()	24
FUN_007f1adc	007f1adc	undefined FUN_007f1adc()	68
FUN_007f1f40	007f1f40	undefined FUN_007f1f40(undefined param...	188

How to find interesting functions?

In this case, we are lucky, we could search for strings then search for references to those strings.

Good examples of strings to search for:

- crypt, decrypt, encrypt
- firmware
- http, https
- uboot, boot, secure boot, sb

Another approach would be to search for peripherial access like ethernet, SPI, UART, etc.

If you search for the string `decrypt`, you should find the complete string `Starting decrypt` at address **0x7f3a38**

Searching for references should lead you to function `FUN_007f37a0`

That is the one we need to emulate to extract the firmware from the flash image we are currently working with.

Static analysis of FUN_007f37a0

We note that the function takes a single unsigned integer as an argument.

We may assume that would be a pointer to some offset within our flash image, we may try 0 first.

`Angr` could even be useful here, to find which integer gets us where we want to go.

A test is being done at **0x7f37fc** against the first 4 bytes of the offset pointed at by first argument. They must be equal to **0x33**, **0x4f**, **0x4c** and **0x69** which decodes to `iLO3` in `ASCII`, mind the endianness.

If we look at address **0x7f3a08**, toward the end of that function, we see message about decompression being done.

Our firmware should be accessible at this point so that is where we want to go

Static analysis of FUN_007f37a0 (cont'd)

Magic number at offset 0x440 (0x110 * 4): 0x334F4C69 (iLO3)

Length of compressed `ELF` file at offset 0x880 (0x110*4 + 0x440)

Offset of compressed `ELF` file at offset 0x884, immediately after length.

Load `unicorn/ilo3/ilo-extract.py` in `Jupyter`

Limitations of Unicorn

- Unicorn emulates a CPU but not an operating system.
- Instructions used for communications between user mode and the kernel are not supported
  - Under 32-bit ARM, the SWI instruction is used for communications with the kernel (SYSCALLS)

It is therefore not possible to use **Unicorn** to emulate software that invokes syscalls to communicate with the kernel, e.g. socket().

It is still possible to manage syscalls, see the example in the repository:

https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/sample_network_auditing.py

Tcpvuln challenge

Inspired by a real life challenge

Load `unicorn/tcpvuln` binary in `Ghidra`

Load `unicorn/tcpvuln.py` in Jupyter

Can you get to the `http_auth` function? If not, why?

Qiling

From the github repository: https://github.com/qilingframework/qiling

While Unicorn emulates a CPU, Qiling aims to emulate a whole operating system

Some operating systems supported: `Linux`, `Windows`, `BSD`, `UEFI`, `Ethereum virtual machine`

Supports executable file formats like `PE` and `ELF`

Can handle system calls

Can load shared libraries

Provide hooks like Angr and Unicorn

A few good demos of Qiling

Qiling for malware analysis

https://n1ght-w0lf.github.io/tutorials/qiling-for-malware-analysis-part-1/

Hooking memfd_create to gain access to a dropped payload
https://codemuch.tech/2021/04/28/unpacking-in-memory-malware/

Using Qiling for UEFI emulation

https://www.sentinelone.com/labs/moving-from-manual-reverse-engineering-of-uefi-modules-to-dynamic-emulation-of-uefi-firmware/

Emulate Android native libraries
https://www.appknox.com/blog/how-to-emulate-android-native-libraries-using-qiling

Qiling CTF

https://joansivion.github.io/qilinglabs/

Emulating tcpvuln using Qiling

We'll use `Qiling` to emulate the `tcpvuln` binary we tried to emulate with `Unicorn`

Could we reach `http_auth` this time?

Limitations of Qiling

From https://qiling.io/comparison/:

Qemu is a full system emulator, but not an analysis tool. Qemu comes with build-in GDB and we cannot analyze a process via qemu. In contrast, Qiling Framework does not perform full system emulation but it still understands OS, syscalls & IO handlers

Sometimes we really do need a full system emulator, like when a service your are interested in like the `HTTP` daemon has close interactions (sockets, pipes, etc) with other serivces running on the device.

Conclusion

Here is other tools I needed to use that shall be covered in a future workshop

Qemu

Allow for full system emulation

May be used to study one process while it interacts closely with many other processes

Was successful building a kernel for a Qemu supported board to emulate binaries from a root filesystem extracted from a device.

Renode

Create your own machines for emulation. You start from a bus, the you add CPU(s) and peripherial

Recreate peripherial from a flattened device tree you managed to extract from a device

Unfortunately won't fit in this workshop