angr Engine Demo

In this demo, I will show the basics of using angr, a binary symbolic execution framework, to solve a CTF challenge. This challenge involves an executable named `engine`:

```
(ARCUS) carter@barnum:/tmp/tmp.dw2sJ8JDOn$ file engine
engine: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=ed2cdedc3195e1000eda552554e10cbd0906075c, for GNU/Linux 3.2.0,
stripped
```

You can download a copy of `engine` here.

As shown by `file`, this is a stripped binary with no debug symbols. If I run it, I get the following prompt:

```
(ARCUS) carter@barnum:/tmp/tmp.dw2sJ8JDOn$ ./engine
Welcome! We're doing things a little differently here, you're going to tell me a
tidbit about trains. Are you ready?
```

Pressing Enter then prompts me to enter a message:

```
(ARCUS) carter@barnum:/tmp/tmp.dw2sJ8JDOn$ ./engine
Welcome! We're doing things a little differently here, you're going to tell me a
tidbit about trains. Are you ready?

Hey now, what're you trying to pull over on me, conductor?
Now, I hope you're a total trainiac. Give me your best tidbit:
```

Since I don't know what it wants me to type, I'll just type `asdf`:

```
(ARCUS) carter@barnum:/tmp/tmp.dw2sJ8JDOn$ ./engine
Welcome! We're doing things a little differently here, you're going to tell me a
tidbit about trains. Are you ready?

Hey now, what're you trying to pull over on me, conductor?
Now, I hope you're a total trainiac. Give me your best tidbit: asdf
I guess you don't know anything about trains...go do some TRAINing you
non-conductor :(
```

Apparently, that's not the input it wanted. Let's use angr to find the correct input for this program. Before getting started, you should install angr on your system by following the installation instructions on angr's website.

Setup

First, let's start a Python shell and import `angr` and `claripy`. `angr` is the core angr module and `claripy` is angr's standalone module for its underlying constraint solving engine.

```
(ARCUS) carter@barnum:/tmp/tmp.dw2sJ8JDOn$ python
Python 3.9.12 (05fbe3aa5b0845e6c37239768aa455451aa5faba, Mar 29 2022, 08:15:34)
[PyPy 7.3.9 with GCC 10.2.1 20210130 (Red Hat 10.2.1-11)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>> import angr; import claripy
>>>>
```

Next, let's create an angr project for our target binary `engine`:

```
>>>> p = angr.Project('./engine')
WARNING | 2022-09-12 13:55:52,527 | pyvex.lifting.gym.x86_spotter | The
generalized AAM instruction is not supported by VEX, and is handled specially by
pyvex. It has no flag handling at present.  See pyvex/lifting/gym/x86_spotter.py
for details
WARNING | 2022-09-12 13:55:53,489 | cle.loader | The main binary is a
position-independent executable. It is being loaded with a base address of
0x400000.
WARNING | 2022-09-12 13:55:53,583 | cle.backends.tls | The provided object has
an invalid tls_data_size. Skip TLS loading.
```

angr prints some warnings to let us know what assumptions it's making about the `engine` binary. These are safe to ignore.

Creating an Initial State

The next thing we need to do is create the starting state for our symbolic analysis. In the case of `engine`, because it reads inputs from `stdin`, it's important that we define this input correctly.

Recall from our first interaction with `engine` what it expects to recieve as `stdin` input. First, we entered a newline character by pressing Enter to proceed from the welcome message to the message that prompts us to enter our "best tidbit." We can represent this using a *bitvector value* (`claripy.BVV`), which is how angr encodes concrete data (i.e., data that has only 1 possible value):

```
>>>> first_char = claripy.BVV(b'\n')
>>>>
```

Next, `engine` wanted us to input some kind of message. Since we don't know the correct message's contents or length, we'll create one *bitvector symbol* (`claripy.BVS`) for each character and guess that there's 256 characters total:

```
>>>> msg = [claripy.BVS('flag_%d' % i, 8) for i in range(256)]
>>>>
```

Lastly, we have to input another newline to mark the end of our answer:

```
>>>> last_char = claripy.BVV(b'\n')
>>>>
```

We now have a bunch of bitvector values for the characters we do know the value of, and a bunch of bitvector symbols for the ones we don't. We can now concatenate them into a string:

```
>>>> stdin = [first_char] + msg + [last_char]
>>>> stdin = claripy.Concat(*stdin)
>>>>
```

Now that we've defined our input, let's create a starting state:

```
>>>> ss = p.factory.full_init_state(args=['./engine'], stdin=stdin)
WARNING | 2022-09-12 14:14:46,521 | angr.simos.simos | stdin is constrained to
258 bytes (has_end=True). If you are only providing the first 258 bytes instead
of the entire stdin, please use stdin=SimFileStream(name='stdin',
content=your_first_n_bytes, has_end=False).
>>>>
```

Here we use `args` to set the `argv` of the program. Since `engine` doesn't need any command line arguments, we just provide `argv[0]`, which is typically the name of the binary. For `stdin`, we provide our custom bitvector string.

As you can see, angr printed a warning message to tell us that it's unusual to define a max length for `stdin`. This is fine in our case because we already have a general idea of what `stdin` should contain.

At this point, we've created a starting state where the first and last bytes of `stdin` are newline characters and all the bytes between are symbols with no constraints (i.e., they can be any value). However, leaving these middle bytes unconstrained is excessive because we know they have to ASCII characters (i.e., the message we're searching for has to be something we can type on a keyboard). Thus, let's add constraints to each byte of the message so that they must be ASCII character (except newline):

```
>>>> for c in msg:
....     ss.solver.add(c < 0x7f)
....     ss.solver.add(c > 0x20)
....
```

I found the values `0x7f` and `0x20` by consulting an ASCII table.

Running Symbolic Analysis

Now that we have a starting state, it's time to unleash angr to find every feasible path through `engine`. This is done using a *Simulation Manager*. First, we initialize our manager with our starting state:

```
>>>> sm = p.factory.simulation_manager(ss)
>>>>
```

And then we run it (text truncated for brevity):

```
>>>> sm.run()
WARNING | 2022-09-12 14:25:02,150 |
angr.storage.memory_mixins.default_filler_mixin | The program is accessing
memory with an unspecified value. This could indicate unwanted behavior.
WARNING | 2022-09-12 14:25:02,150 |
angr.storage.memory_mixins.default_filler_mixin | angr will cope with this by
generating an unconstrained symbolic variable and continuing. You can resolve
this by:
WARNING | 2022-09-12 14:25:02,150 |
angr.storage.memory_mixins.default_filler_mixin | 1) setting a value to the
initial state
WARNING | 2022-09-12 14:25:02,151 |
angr.storage.memory_mixins.default_filler_mixin | 2) adding the state option
ZERO_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to make unknown regions hold null
WARNING | 2022-09-12 14:25:02,151 |
angr.storage.memory_mixins.default_filler_mixin | 3) adding the state option
SYMBOL_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to suppress these messages.
[...]
<SimulationManager with 76 deadended>
>>>>
```

Depending on your system's hardware, the analysis will take somewhere between 3 and 30 minutes to complete.

Inspecting the Results

Our Simulation Manager has gone and found every feasible path through `engine` starting from the initial state we constructed. The final states are organized into stashes, which we can list:

```
>>>> sm.stashes
defaultdict(<class 'list'>, {'active': [], 'stashed': [], 'pruned': [], 'unsat':
```

```
[], 'errored': [], 'deadended': [<SimState @ 0xd00088>, <SimState @ 0xd00088>,
<SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @
0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>,
<SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @
0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>,
<SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @
0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>,
<SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @
0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>,
<SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @
0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>,
<SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @
0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>,
<SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @
0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>,
<SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @
0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>,
<SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @
0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>,
<SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @ 0xd00088>, <SimState @
0xd00088>], 'unconstrained': []})
>>>>
```

For the purposes of this tutorial, I won't go through what each of these stashes means, but know that the states in `deadended` are the ones that reached the end of `engine`'s execution.

We can ask claripy's solver engine to evaluate `stdout` for one of these states to see what `engine` would print if we executed that path:

```
>>>> sm.deadended[0].posix.stdout.concretize()
[b"Welcome! We're doing things a little differently here, you're going to tell
me a tidbit about trains. Are you ready?", b'\n', b"Hey now, what're you trying
to pull over on me, conductor?", b'\n', b"Now, I hope you're a total trainiac.
Give me your best tidbit: ", b"I guess you don't know anything about trains...go
do some TRAINing you non-conductor :(", b'\n']
>>>>
```

In the case of this state, `engine` printed out the same failure message we got the first time we tried to run it. Let's see if there's a state that prints something different:

```
>>>> for i, s in enumerate(sm.deadended):
....     stdout = b''.join(s.posix.stdout.concretize())
....     if not b"I guess you don't know anything about trains" in stdout:
....         print("%d: %s" % (i, stdout))
....
74: b"Welcome! We're doing things a little differently here, you're going to
tell me a tidbit about trains. Are you ready?\nHey now, what're you trying to
pull over on me, conductor?\nNow, I hope you're a total trainiac. Give me your
best tidbit: Chugga chugga choo choo you're the little engine that CAN!\n"
```

Bingo! There's 1 state where `engine` prints what appears to be the success message.

Now let's evaluate `stdin` to see what needs to be entered to reach this state:

```
>>>> sm.deadended[74].posix.stdin.concretize()
b'\n?p?c?t?f?{?t?h?3?_?m?0?d?3?r?n?_?s?t?3?4?m?_?3?n?g?1?n?3?_?w?4?5?_?1?n?v?3?
n?t?3?d?_?1?n?_?1?6?9?8?_?b?u?T?_?t?h?3?_?b030s0t0_000n030_0i0n0_010904000}0000
0000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000\n'
>>>>
```

It's a little hard to read in its raw format because `stdin` uses a 2-byte character encoding, which is why every other byte is a `0` or `?`. We can clean this up by stripping off the newline characters and then decoding every other byte into an ASCII character:

```
>>>> raw = sm.deadended[74].posix.stdin.concretize()
>>>> raw_msg = raw[1:-1]
>>>> ''.join([chr(raw_msg[i]) for i in range(1, len(raw_msg), 2)])
'pctf{th3_m0d3rn_st34m_3ng1n3_w45_1nv3nt3d_1n_1698_buT_th3_b3st_0n3_in_1940}000
0000000000000000000000000000000000000000000000000'
```

We can also trim off the `0` characters at the end. They're there because we allocated 256 characters for the message and it turned out the correct message is only 75 characters:

```
>>>> ''.join([chr(raw_msg[i]) for i in range(1, len(raw_msg), 2)])[:75]
'pctf{th3_m0d3rn_st34m_3ng1n3_w45_1nv3nt3d_1n_1698_buT_th3_b3st_0n3_in_1940}'
```

Alright, that should be the correct input to `engine`. Let's exit the Python shell and give it a shot:

```
(ARCUS) carter@barnum:/tmp/tmp.dw2sJ8JDOn$ ./engine
Welcome! We're doing things a little differently here, you're going to tell me a
tidbit about trains. Are you ready?

Hey now, what're you trying to pull over on me, conductor?
Now, I hope you're a total trainiac. Give me your best tidbit:
pctf{th3_m0d3rn_st34m_3ng1n3_w45_1nv3nt3d_1n_1698_buT_th3_b3st_0n3_in_1940}
Chugga chugga choo choo you're the little engine that CAN!
```

Success!

We just solved a CTF challenge without having to look at a single line of assembly code.

Happy hacking!