angr Under-Constraining Demo

In this demo, I'm going to show one of the many ways angr can be used to test individual functions for possible vulnerabilities. Specifically, I'm going to focus on how we can use a control flow graph (CFG) and a special starting state in angr called the `call_state` to identify and symbolically execute individual functions in a program, as opposed to executing full paths.

This type of analysis is what's called *under-constrained* symbolic execution because rather than building the constrains for full paths, starting from the entry point, we're going to build constraints starting from an internal program function that in a real execution would be called by other functions. Under-constrained analysis scales better to large programs, however because we aren't building the constraints for a full path, it can also yield unreachable states, which for the purposes of vulnerability detection can result in false positives. Regardless, under-constrained analysis is a useful technique when it isn't feasible to execute the entire program.

Setup

For this demo, I'm going to use a small toybox program that contains a stack overflow vulnerability:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef void (*anon_subr_void_char_ptr)(char *);

int min(int a,int b)
{
  if (a < b) {
    b = a;
  }
  return b;
}

void greet(char *name)
{
  printf("Hello %s!\n",name);
  return;
}

int do_main(int argc,char **argv)
{
  int iVar1;
  char name[8] = {0};
  anon_subr_void_char_ptr m_func = greet;

  iVar1 = atoi(argv[2]);
  iVar1 = min(iVar1,0x100);
  memcpy(name,argv[1],(long)iVar1);
  m_func(name);
  return 0;
}


int main(int argc,char **argv)
{
```

```
  if (argc != 3) {
    puts("Usage: program <name> <length>");
    return 1;
  }

  do_main(argc,argv);
  return 0;
}
```

First, let's compile this program (with debug symbols to make our lives easier). I've named it `main.c` (alternatively, you can download it here):

```
carter@barnum:/tmp/tmp.MtCyxH5MWU$ gcc -g -o hello main.c
carter@barnum:/tmp/tmp.MtCyxH5MWU$
```

If we run the program, it asks for a name and length and then greets us:

```
carter@barnum:/tmp/tmp.MtCyxH5MWU$ ./hello
Usage: program <name> <length>
carter@barnum:/tmp/tmp.MtCyxH5MWU$ ./hello Alice 5
Hello Alice!
carter@barnum:/tmp/tmp.MtCyxH5MWU$
```

Building CFG

Let's build a CFG of this program. First, let's import angr and create a project:

```
(env) carter@barnum:/tmp/tmp.MtCyxH5MWU$ 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
>>>> p = angr.Project('./hello', auto_load_libs=False)
WARNING | 2022-09-18 10:11:44,106 | cle.loader | The main binary is a
position-independent executable. It is being loaded with a base address of
0x400000.
>>>>
```

For this project, I'm setting the parameter `auto_load_libs` to `False` so angr's loader, CLE, will not load imported libraries like `libc`. This will allow us to build a CFG for just the main program, which will speed up analysis.

Let's now build our CFG:

```
>>>> cfg = p.analyses.CFGFast()
>>>>
```

angr has two types of CFGs: `CFGFast` and `CFGEmulated`. You can read about their differences here

Now, pretend that this program is massive and we don't know where to get started. One thing we could try is printing the name of every function angr knows about:

```
>>>> for func in cfg.functions:
....   print(cfg.functions[func].name)
....
_init
```

```
sub_401020
sub_40102c
puts
printf
memcpy
atoi
__cxa_finalize
_start
sub_4010aa
sub_4010ab
deregister_tm_clones
sub_4010d9
register_tm_clones
sub_401119
__do_global_dtors_aux
sub_401151
sub_401159
frame_dummy
min
greet
do_main
main
sub_401267
__libc_csu_init
sub_4012cd
__libc_csu_fini
_fini
__libc_start_main
puts
printf
memcpy
atoi
UnresolvableJumpTarget
UnresolvableCallTarget
>>>>
```

Even a small program like this has quite a few. Notice that this program uses `memcpy`, which is a `libc` function for copying data between buffers. Callers of this function are a prime candidate for overflow bugs.

So how do we test the callers of `memcpy` for bugs? First, we need to identify who they are. To start, let's extract from the CFG's *knowledge base* the information it has about `memcpy`:

```
>>>> memcpy = cfg.functions.function(name='memcpy')
>>>> memcpy
<Function memcpy (0x401050)>
>>>>
```

You can read more about the `Function` object here. The first thing to note is that the starting address for `memcpy` (0x401050) is a PLT stub. We can verify that like so:

```
>>>> hex(memcpy.addr)
'0x401050'
>>>> memcpy.is_plt
True
>>>>
```

If you aren't familiar with what a *procedure linkage table* (PLT) is, you can read about them here.

For the purposes of this demo, all we need to know is that any function in our program that calls `memcpy` must go through its PLT stub because it's a function imported from another library. Therefore, any nodes in the CFG that are `predecessors` to this PLT stub's nodes are callers. We can grab them like so:

```
>>>> callers = set()
>>>> for plt_node in cfg.get_all_nodes(addr=memcpy.addr):
....   for pred in plt_node.predecessors:
....     callers.add(pred.addr)
>>>> callers
{4198896}
>>>>
```

So what exactly did I do here? First, I created a set (`callers`) for storing the base addresses of basic blocks that call into `memcpy`. I then used `get_all_nodes` to get every node in the CFG that corresponds to the address of the PLT stub. The reason I use `get_all_nodes` instead of `get_any_node` is because in angr's CFGs, multiple nodes can correspond to the same basic block due to something called *context sensitivity*, which you can read about here.

Anyway, turns out there's only one basic block that calls `memcpy` in our program. You can verify that by reading over the original source code. Let's ask angr to describe this block:

```
>>>> caller = list(callers)[0]
>>>> p.loader.describe_addr(caller)
'do_main+0x47 in hello (0x11f0)'
>>>>
```

What this description says is that this basic block belongs to the function `do_main` in our program and it's located 0x47 bytes from `do_main`'s entry point and 0x11f0 bytes from the program binary's base address.

Preparing the Starting State

Now that we've identified all the locations in our program that call `memcpy`, let's do an under-constrained symbolic execution of each to determine if it might be vulnerable. First, we need the starting addresses of the functions our callers belong to:

```
>>>> do_main = cfg.get_any_node(addr=caller).function_address
>>>> do_main
4198825
>>>>
```

Notice how in this case I was able to use `get_any_node` because all CFG nodes for our caller's basic block will belong to the same function.

Now, we can create a starting state for the function `do_main`:

```
>>>> ss = p.factory.call_state(do_main, ret_addr=0)
WARNING | 2022-09-18 10:51:56,517 | angr.calling_conventions | Guessing call
prototype. Please specify prototype.
>>>>
```

Unlike a `full_init_state` or `init_state`, a `call_state` is a special starting state where angr creates 1 stack frame suitable for calling a single function. This is useful for symbolically executing a function that is not the program's entry point.

I like to explicitly setup my call states to return to address 0x0 because this makes it easy to detect when our starting function has returned.

Now that we have our starting state, we're ready to execute, but before we do that, we need to discuss what we're looking for as an indicator of a bug.

Detecting Control Flow Hijacks Symbolically

Since we're analyzing a function that calls `memcpy` and looking for overflows, we're going to watch for any states that become *unconstrained*. What this means is that the state's program counter has become completely symbolic and the next address that the CPU executes could be anything. This is often indicative of a bug that can allow a user to hijack the program's control flow, which is a serious vulnerability.

In angr, simulation managers automatically place unconstrained states in a special stash named `unconstrained`. Thus, while we could run our symbolic execution to completion, we really only need to run until we find an unconstrained state. Let's do that (some output truncated for brevity):

```
>>>> sm = p.factory.simulation_manager(ss)
>>>> while len(sm.active) > 0 and len(sm.unconstrained) < 1:
....   sm.step()
....
[...]
WARNING | 2022-09-18 11:04:18,282 | angr.engines.successors | Exit state has
over 256 possible solutions. Likely unconstrained; skipping.
<BV64 Reverse(mem_4032404044_62_2048{UNINITIALIZED}[1951:1888])>
<SimulationManager with 2 unconstrained>
>>>>
```

What I did here was instead of calling `sm.run()`, which would run the simulation until there's nothing left to execute, I instead created a while loop that steps all active states 1 basic block at a time and exits early if an unconstrained state is found (or there are no active states left to step).

In our case, the simulation manager found 2 unconstrained states. Just as a demonstration of what I mean by unconstrained, we can try to evaluate the value of the program counter (RIP) to multiple concrete values:

```
>>>> s1 = sm.unconstrained[0]
>>>> s1.solver.eval_upto(s1.regs.rip, 10)
[4198786, 0, 4198656, 4194304, 288231544651333636, 288231544718450708, 288231544718450700,
289357444625293324, 290483344532135948, 290483344532267020]
>>>>
```

As you can see, the next virtual address this state will execute can be anything. This is because `memcpy` has overwritten the function's return pointer.

Conclusion

And there you have it. Using under-constrained symbolic execution, we were able to identify a likely vulnerable function in our program. As a reminder, because we didn't build the full path constraints, the unconstrained states we found might not be reachable. However, in this case, our toybox program is simple enough that we can figure out a proof-of-compromise without even needing the constraint solver:

```
(env) carter@barnum:/tmp/tmp.MtCyxH5MWU$ ./hello AAAAAAAAAAAAAAAAAAAAAAAAA 20
Segmentation fault
(env) carter@barnum:/tmp/tmp.MtCyxH5MWU$
```

Happy hacking!