Fuzzing and Symbolic Execution

Algorithm for Find and Avoid

- Load the binary
- Specify a starting point and create a simulation manager
- While we have not found what we want…
  - Step all active states
  - Run our '`should_accept_state`' function on each active state
    - If one matches, we found what we wanted! Exit the loop
  - Run our '`should_avoid_state`' function on each active state
    - For each state that is matches, mark it for termination
  - Remove all states that are marked for termination from the set of active states

Shortcut: The 'Explore' Method

The previous algorithm is so common that `angr` wrote a single function to do it for you, called the '**`explore`**' function:

```
simulation.explore(find=should_accept, avoid=should_avoid)
```

… will add any path that is accepted to the list '**`simulation.found`**'

Additionally, searching or avoiding a specific instruction address is common enough that the find and avoid parameters also accept addresses:

```
simulation.explore(find=0x80430a, avoid=0x9aa442)
```

… would search for address **`0x80430a`** and terminate anything that reaches **`0x9aa442`**.

Angr CTF levels

- Feel free to work with others, but you must run on your own to solve your own levels
  - `00_angr_find`
  - `01_angr_avoid`
  - `02_angr_find_condition`

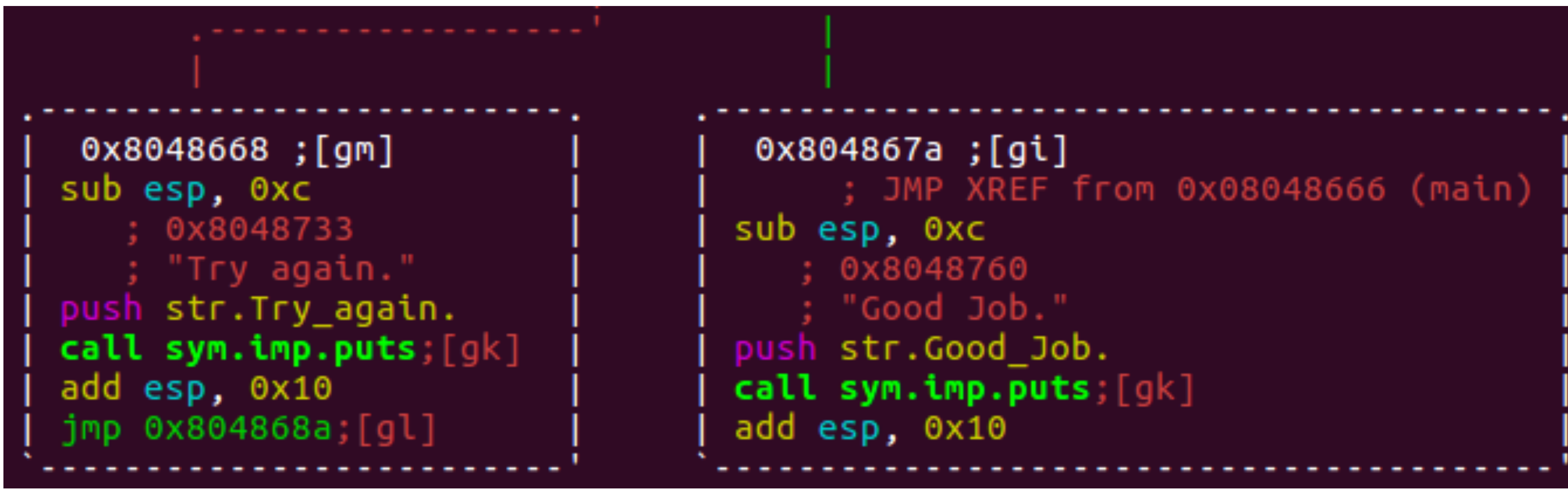
00_angr_find

- Our "Hello World" of angr
  - Simple CTF level, but with `complex_function` added in order to…
    - Make manual analysis difficult
    - But, allow a symbolic execution engine to solve in under a minute

```
int main(int argc, char* argv[]) {
  char buffer[9];

  printf("Enter the password: ");
  scanf("%8s", buffer);

  for (int i=0; i<LEN_USERDEF; ++i) {
    buffer[i] = complex_function(buffer[i], i);
  }

  if (strcmp(buffer, USERDEF)) {
    printf("Try again.\n");
  } else {
    printf("Good Job.\n");
  }
}
```

00_angr_find

- Analyze binary to find where it prints what you want (`0x804867a`)



00_angr_find

- Each level includes a template solution script (`scaffoldXX.py`)
  - Full description of level in comments within each script
  - IMPORTANT: Read this description to save yourself some time
- Scaffolded to focus only on one new part of angr per level
  - `scaffold00.py`

```
path_to_binary = ???  # :string
project = angr.Project(path_to_binary)

initial_state = project.factory.entry_state()

simulation = project.factory.simgr(initial_state)


print_good_address = ???  # :integer (probably in hexadecimal)
simulation.explore(find=print_good_address)

if simulation.found:
  solution_state = simulation.found[0]

  print(solution_state.posix.dumps(sys.stdin.fileno()).decode())
else:
  raise Exception('Could not find the solution')
```

← Specify path to binary (string)

← Create project with binary

← Start at main()

← Create simulation manager with initial path

← Location to find input for

← Do entire exploration

← See if something was found

← Get first solution (symbol with Constraints)

← Concretize symbol by evaluating constraints on input (usually only 1 possible solution for our CTF). Print input that led to solution

02_angr_find_condition

- Can also use alternate conditions to find states

```
path_to_binary = argv[1]
project = angr.Project(path_to_binary)
initial_state = project.factory.entry_state()
simulation = project.factory.simgr(initial_state)


def is_successful(state):
  stdout_output = state.posix.dumps(sys.stdout.fileno())
  return ???  # :boolean


def should_abort(state):
  stdout_output = state.posix.dumps(sys.stdout.fileno())
  return ???  # :boolean


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

if simulation.found:
  solution_state = simulation.found[0]
  print(solution_state.posix.dumps(sys.stdin.fileno()).decode())
else:
  raise Exception('Could not find the solution')
```

Define a function that represents the state you want to look for.

Dumps stdout to variable.

Expression that should return true if "Good Job" is 'in' it.

Same as is_successful, but specify states that should abort immediately. States not matching will return false and continue execution.

Return if a state is_successful(). Avoid states that should_abort().

- String comparisons and conversions from stdin/stdout
  - I/O in angr done using UTF-8 bytes

```
stdout_output = state.posix.dumps(sys.stdout.fileno())
```

```
type(stdout_output) ➔ <class 'bytes'>
```

  - When searching for patterns in `stdout_output`, must perform encoding
    - Helpful Python hint for CTF
    - What is the difference in Python between..

```
'Good Job'.encode() in stdout_out
```

and

```
stdout_out == 'Good Job'.encode()
```

Symbolic Execution CTF: Part 2

Introducing Symbols and Constraints

Why are the first three levels of the CTF solved without injecting symbols?

- Angr injects them for you!

- Angr can automatically inject a symbol when user input is provided from `stdin` *
  - Can also handle simple calls of '`scanf`' (without complex format strings.)

- You will need to inject symbols manually if the input is more complex, for example:
  - Arbitrary format string for `scanf`
  - From a file
  - From the network
  - From interactions with a UI

- How?

* Using what are called SimProcedures, which we will cover later.

Bitvectors

- Angr represents symbols using its bitvector data structure
- Bitvectors have a size, the number of bits they represent.
- As with all data in programming, bitvectors can represent any type that can fit.
  - Most commonly, they represent either integers or strings for our exercises
- Difference between typical variables and bitvectors
  - Typical variables store a single value
  - Bitvectors represent every possible value that meets the execution path's constraints.

Bitvectors

- Consider a symbol λ
  - An 8 bit value constrained by $(\ \lambda > 0,\quad \lambda \leq 4,\quad \lambda \bmod 2 = 0\ ) \lor (\ \lambda = 1\ )$
  - Bitvector data structure stores size and constraints together.
- Types
  - *Concrete*: a bitvector that can take on exactly 1 value.
    (Example: $\{\ \lambda : \lambda = 1\ \}$)
  - *Symbolic*: a bitvector that can take on more than 1 value.
    (Example: $\{\ \lambda : \lambda > 10\ \}$)
  - *Unsatisfiable*: a bitvector that cannot take on any values.
    (Example: $\{\ \lambda : \lambda = 10, \lambda \neq 10\ \}$)
  - *Unconstrained*: a bitvector that can take on any value, within the bounds of its size.

Using Symbols in the Context of a Program

```
user_input = λ
if user_input == 'hunter2':
  print 'Good Job.'
else:
  print 'Try again.'
```

We can inject symbols into variables as long as the size of the bitvector is equal to the size of the variable.

Constraints are then automatically accumulated as engine executes a given path (ex: $\lambda =$ ‘`hunter2`’, or for the other path, $\lambda \neq$ ‘`hunter2`’)

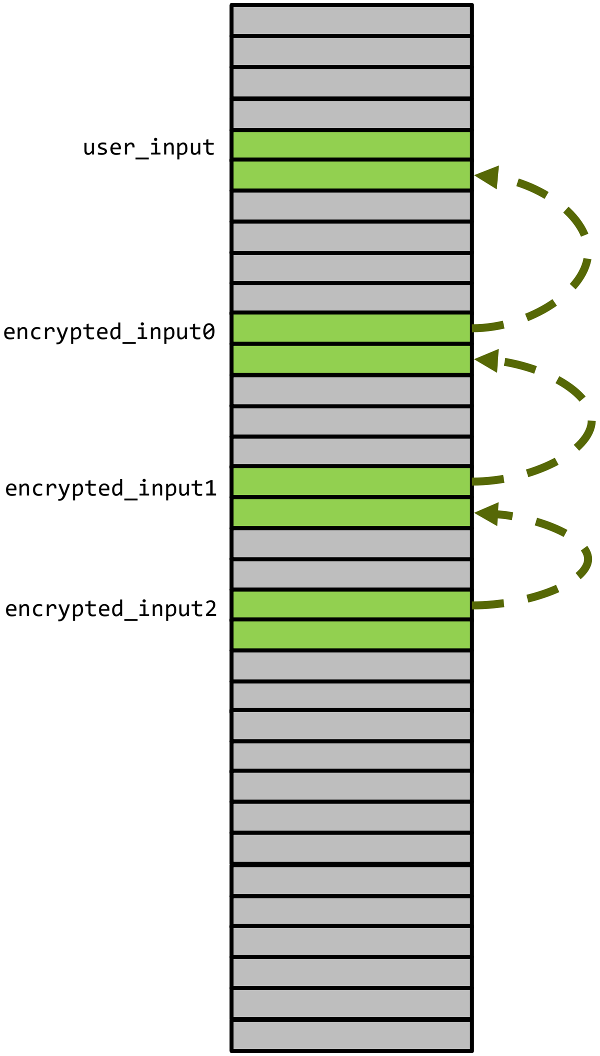
Constraints can also be manually generated and added to any bitvector at any time during the execution of the program (seen in later levels).

Automatic symbol propagation

- Symbols propagate when values are transferred.

```
user_input = λ
encrypted_input0 = user_input - 3
encrypted_input1 = encrypted_input0 + 15
encrypted_input2 = encrypted_input1 * 7
```

- Memory layout with variables marked
  - Locations in green now contain symbolic values.
  - `encrypted_input2` depends entirely on `user_input`, denoted by the arrows



Constraint propagation

- Automatically done by engine
- Example: Forward propagation through program

```
user_input = λ
encrypted_input0 = user_input - 3
encrypted_input1 = encrypted_input0 + 15
encrypted_input2 = encrypted_input1 * 7
```

If we add the constraint: $\lambda = 10$, then:

```
user_input = λ = 10
encrypted_input0 = user_input - 3 = 10 – 3 = 7
encrypted_input1 = encrypted_input0 + 15 = 7 + 15 = 22
encrypted_input2 = encrypted_input1 * 7 = 22 * 7 = 154
```

We solved for encrypted_input2!

- Example: Reverse propagation to move backwards through program
  - Constraints applied to propagated symbolic values to solve for earlier state
  - Allows us to solve for initial conditions (sound like something you might be interested in?)

```
user_input = λ
encrypted_input0 = user_input - 3
encrypted_input1 = encrypted_input0 + 15
encrypted_input2 = encrypted_input1 * 7
```

Add the constraint: `encrypted_input2 = 14`, then:

We solved for λ!

```
λ = user_input, λ = -10
user_input - 3 = encrypted_input0 = -13, user_input = -10
encrypted_input0 + 15 = encrypted_input1 = 2, encrypted_input0 = -13
encrypted_input1 * 7 = encrypted_input2 = 14, encrypted_input1 = 2
```

Start here, work backwards

Concretizing Bitvectors based on constraints

- Angr provides a nice frontend to `z3`, an open-source constraint solver from Microsoft used with SAGE.

- It has the following functionality for producing concrete values:
  - Find any (single) value of a bitvector
  - Find up to `n` possible values of a bitvector
  - Find the maximum or minimum possible values of a bitvector
  - Determine if a bitvector is satisfiable

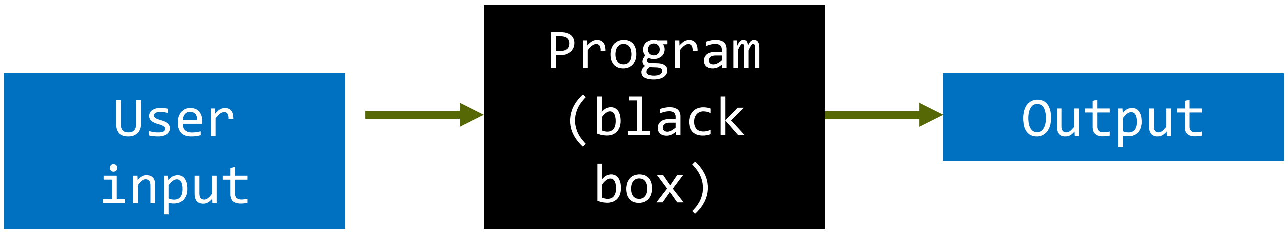
Injecting symbols with Angr (this set of levels)

- Complexity requires one to limit scope of symbolic execution task

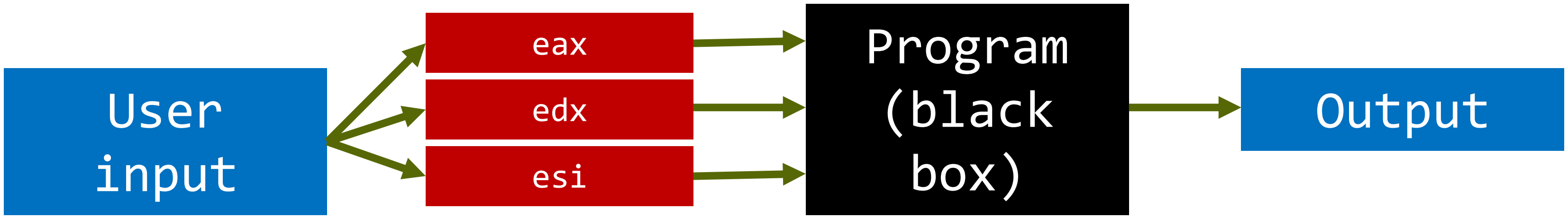
- Angr supports manual symbol injection across a variety of resources for doing so
  - Registers
  - Global memory
  - Stack locations
  - Heap locations
  - File system
  - Network connections

Injecting symbols into Registers

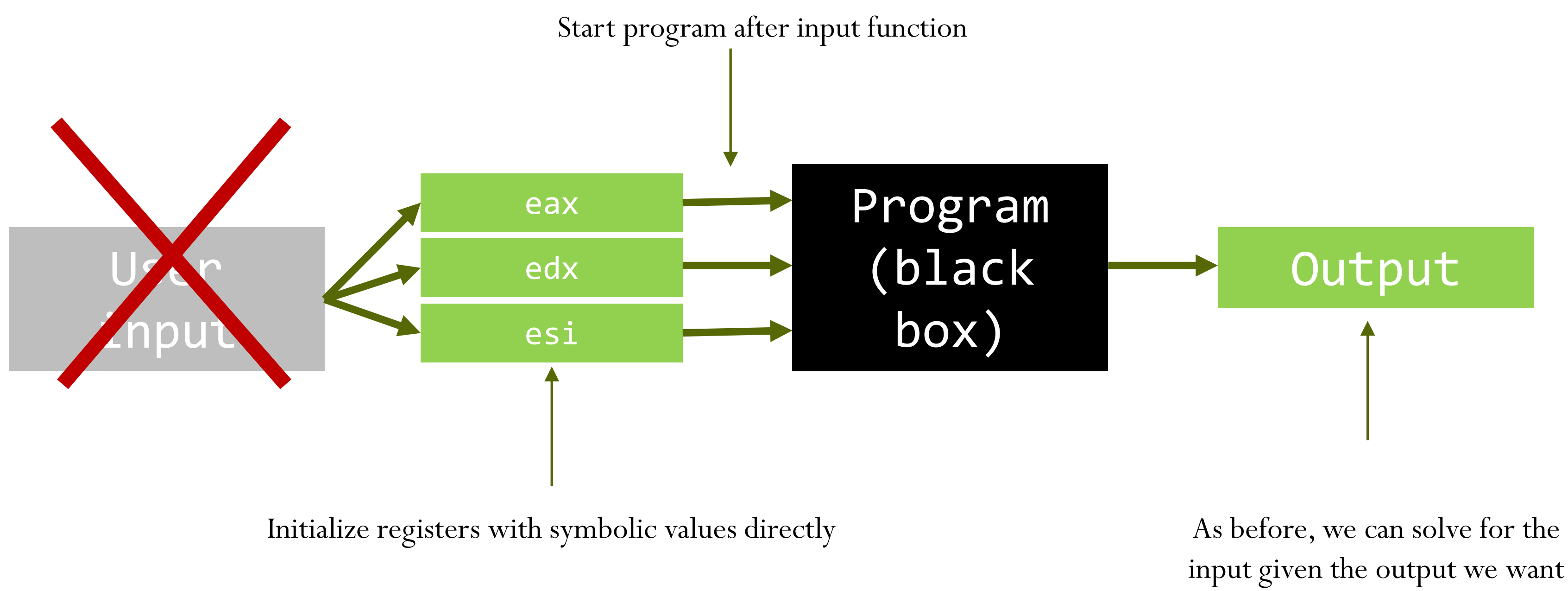
- Initial 3 CTF levels
  - Angr symbolically executes user input routine…



  - …and automatically propagates symbolic values into program state such as registers



- For complex user input functions (e.g. highly formatted input), symbolic execution not efficient
  - Must go through all paths the input function supports
  - Can inject the symbols manually *after* user input and initialize specific registers/memory with symbolic values…



- Example
  - `get_user_input` function returns values by writing them to registers.
  - Instead of having angr symbolically execute `get_user_input`, write symbolic values to the registers, then symbolically execute after `get_user_input`

Start symbolic execution immediately after the call, here.

```
call   80487a5 <get_user_input>
mov    %eax,-0xc(%ebp)
mov    %ebx,-0x8(%ebp)
mov    %ecx,-0x4(%ebp)
```

`get_user_input` function returned the user input into registers.

In Angr, you can write to a register with either a concrete or a symbolic value:

```
state.regs.eax = 0x0

state.regs.eax = my_bitvector
```

(writes the symbolic value `my_bitvector` to `eax`)

Angr CTF level

- `03_angr_symbolic_registers`

03_angr_symbolic_registers

- Three numbers used as password via a formatted `scanf` within `get_user_input()`
  - Difficult to run `scanf` through symbolic execution
  - Numbers returned from `get_user_input` in 3 registers, then moved to memory locations
- Set registers to symbolic bitvectors after call to `get_user_input`
  - Begin symbolic execution from address just after function returns (`0x080488d1`)

```
0x080488c9      83c410          add esp, 0x10
0x080488cc      e887ffffff      call sym.get_user_input        ;[5]
0x080488d1      8945ec          mov dword [local_14h], eax
0x080488d4      895df0          mov dword [local_10h], ebx
0x080488d7      8955f4          mov dword [local_ch], edx
0x080488da      83ec0c          sub esp, 0xc
```

03_angr_symbolic_registers

- Claripy: Package for storing and manipulating symbolic values

```
start_address = ???
initial_state = project.factory.blank_state(addr=start_address)

password0_size_in_bits = ???  # :integer
password0 = claripy.BVS('password0', password0_size_in_bits)
password1_size_in_bits = ???  # :integer
password1 = claripy.BVS('password1', password1_size_in_bits)
password2_size_in_bits = ???  # :integer
password2 = claripy.BVS('password2', password2_size_in_bits)

initial_state.regs.e???x = password0
initial_state.regs.e???x = password1
initial_state.regs.e???x = password2

simulation = project.factory.simgr(initial_state)

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

if simulation.found:
  solution_state = simulation.found[0]
  solution0 = solution_state.solver.eval(???)
  solution1 = solution_state.solver.eval(???)
  solution2 = solution_state.solver.eval(???)
```

Address where symbolic values needed. (After scanf, but before moved. In prior example: 0x80488d1)

Set symbolic execution start address.

Create symbolic bitvectors for each part of password using claripy

Must specify size in bits (e.g. an integer would be 32 bits, an 8 char string would be 64 bits)

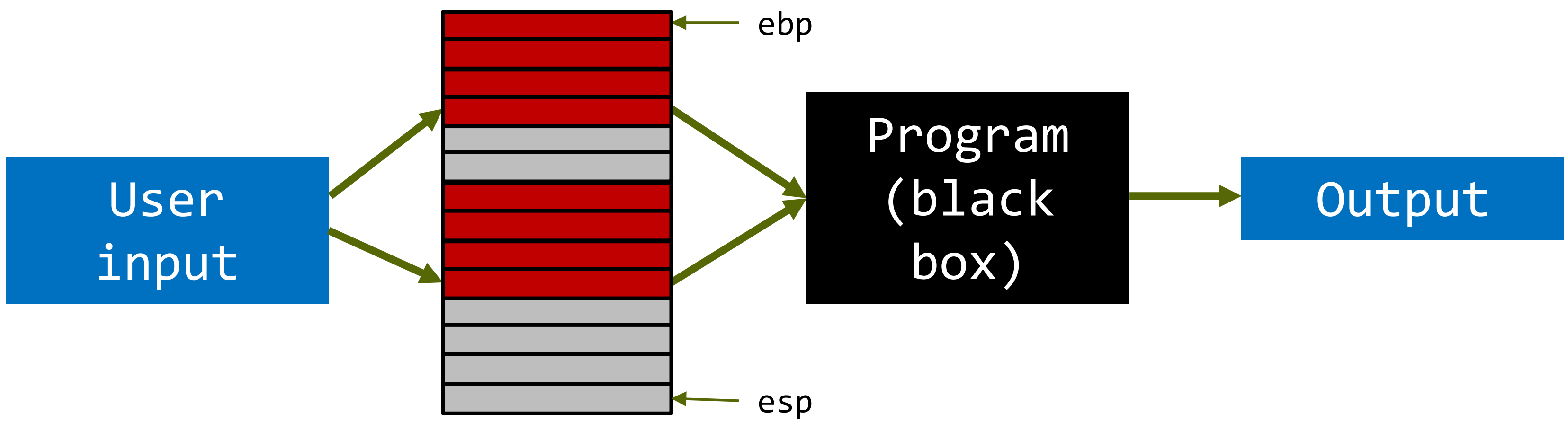
Attach symbolic values to specific registers that get_user_input set

As in 02_angr_find_condition

Evaluate symbolic bitvectors to get concrete integer values that satisfy constraints.

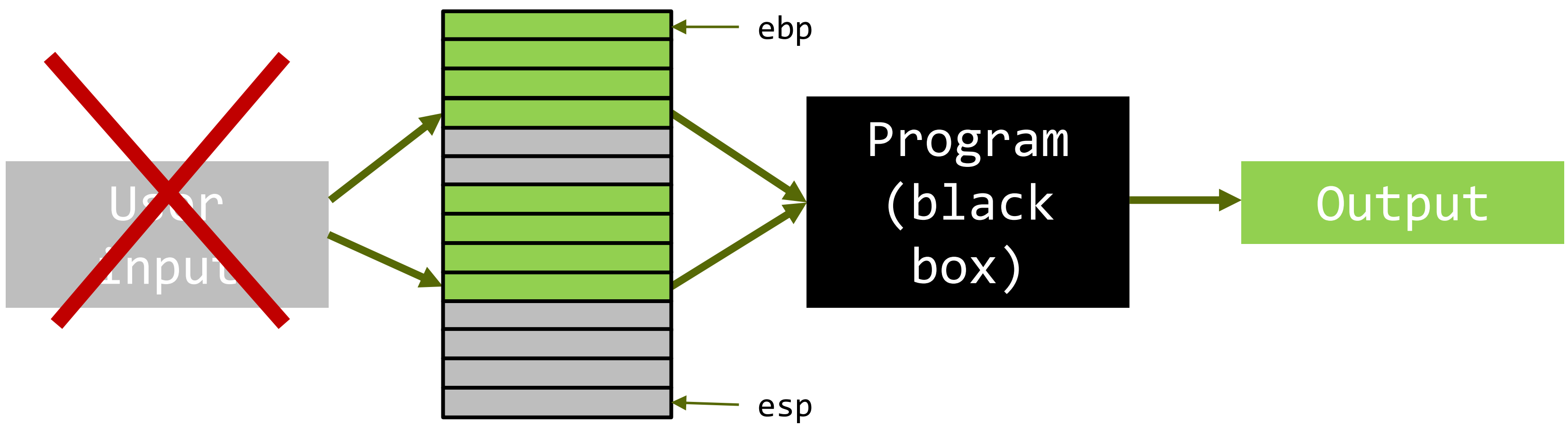
Injecting symbols into the Stack

- Input routine writes to stack memory
  - Can try to run input routine symbolically



Injecting symbols into the Stack

- But, if input routine complex
  - Need to use prior technique



- Example

Allocate memory for local variables

Format string

```
sub     $0x20,%esp
lea     -0x8(%ebp),%eax
push    %eax
push    $0x80489c3
call    8048370 <scanf@plt>
```

Specify a specific local variable as a parameter to `scanf`

In Angr, you can push to the stack with either a concrete or a symbolic value:

`state.stack_push(my_bitvector)`

will push the value of `my_bitvector` to the top of the stack.

Before pushing values, you may need to update state of the stack pointer to reflect what the stack looks like at the address you begin execution from

`state.regs.esp -= 4`

Angr CTF level

- 04_angr_symbolic_stack

04_angr_symbolic_stack

- Password read onto the stack in `handle_user()`
  - `password1 var_10h ebp-0x10`
  - `password0 var_ch ebp-0xc`
  - Then, complex function run on each location before values checked
  - Want to inject symbols and start execution after input read into stack (`0x80486ae`)

```
124: sym.handle_user ();
          ; var int32_t var_10h @ ebp-0x10
          ; var uint32_t var_ch @ ebp-0xc
          0x08048690      55            push ebp
          0x08048691      89e5          mov ebp, esp
          0x08048693      83ec18        sub esp, 0x18
          0x08048696      83ec04        sub esp, 4
          0x08048699      8d45f0        lea eax, dword [var_10h]
          0x0804869c      50            push eax
          0x0804869d      8d45f4        lea eax, dword [var_ch]
          0x080486a0      50            push eax
          0x080486a1      68c3870408    push str.u__u
          0x080486a6      e8c5fcffff    call sym.imp.__isoc99_scanf
          0x080486ab      83c410        add esp, 0x10
          0x080486ae      8b45f4        mov eax, dword [var_ch]
          0x080486b1      83ec0c        sub esp, 0xc
          0x080486b4      50            push eax
          0x080486b5      e806feffff    call sym.complex_function0
```

- To execute within function, must recreate the stack upon function entry
  - Need to ensure `ebp` is set so that `ebp-0x10` contains `password1` bitvector and `ebp-0xc` has `password0` bitvector.
  - Use stack operations to programmatically inject bitvectors onto stack

```
#               /-------- The stack --------\
# ebp ->        |          padding          |
#               |---------------------------|
# ebp - 0x01    |       more padding        |
#               |---------------------------|
# ebp - 0x02    |     even more padding     |
#               |---------------------------|
#                           . . .
#               |---------------------------|
# ebp - 0x0b    |   password0, second byte  |
#               |---------------------------|
# ebp - 0x0c    |   password0, first byte   |
#               |---------------------------|
#                           . . .
#               |---------------------------|
# ebp - 0x0f    |   password1, second byte  |
#               |---------------------------|
# ebp - 0x10    |   password1, first byte   |
#               |---------------------------|
#                           . . .
```

- Algorithm
  - Set `%ebp` and `%esp` to initially point to the same location
  - Offset `%esp` to move beyond frame padding
    - If `password0` at offset `0xc` is 4 bytes, 8 bytes of frame padding to move beyond
    - If `password0` at offset `0xc` is 8 bytes, 4 bytes of frame padding to move beyond
  - Push `password0` bitvector, then push `password1` bitvector

```
#             /-------- The stack --------\
# ebp ->      |          padding           |
#             |----------------------------|
# ebp - 0x01  |       more padding         |
#             |----------------------------|
# ebp - 0x02  |     even more padding      |
#             |----------------------------|
#                        . . .                <- How much padding? Hint: how
#             |----------------------------|     many bytes is password0?
# ebp - 0x0b  |   password0, second byte   |
#             |----------------------------|
# ebp - 0x0c  |   password0, first byte    |
#             |----------------------------|
```

04_angr_symbolic_stack

```
start_address = ???
initial_state = project.factory.blank_state(addr=start_address)

password0 = claripy.BVS('password0', ???)
...

initial_state.regs.ebp = initial_state.regs.esp
padding_length_in_bytes = ???  # :integer
initial_state.regs.esp -= padding_length_in_bytes

...

initial_state.stack_push(???)  # :bitvector

...

simulation = project.factory.simgr(initial_state)

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

if simulation.found:
  solution_state = simulation.found[0]

  solution0 = solution_state.solver.eval(password0)
  ...
```

Start at address when symbolic values are needed. Within handle_user where values are moved..0x80486ae)

Set symbolic execution start address.

Create bitvectors for each of the password values being read.

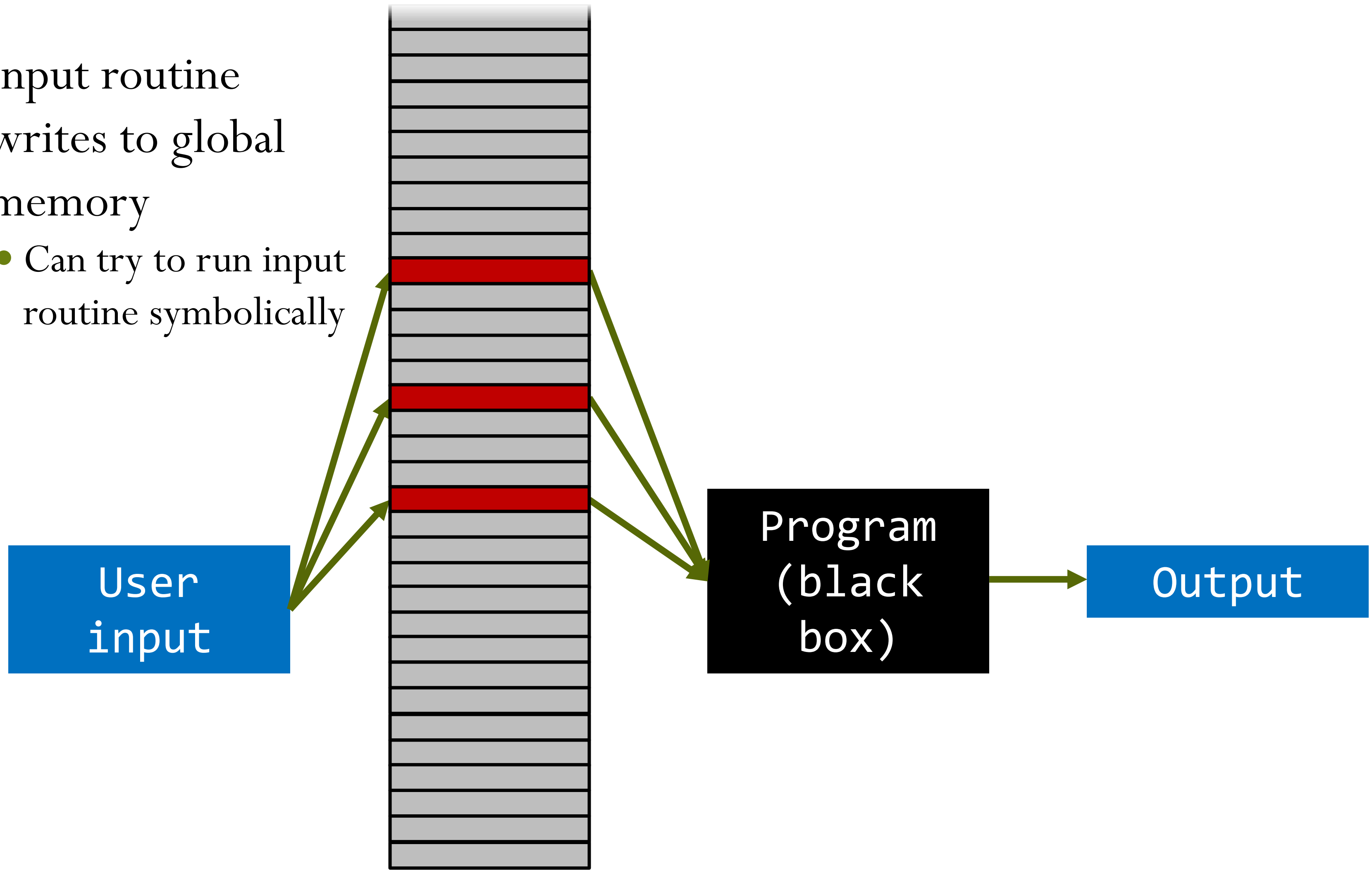
Manually set up stack to run from within function. Set ebp to esp, then update esp past padding in between ebp and the last byte of password0

Push symbolic bitvector for each password onto the stack to recreate initial stack setup

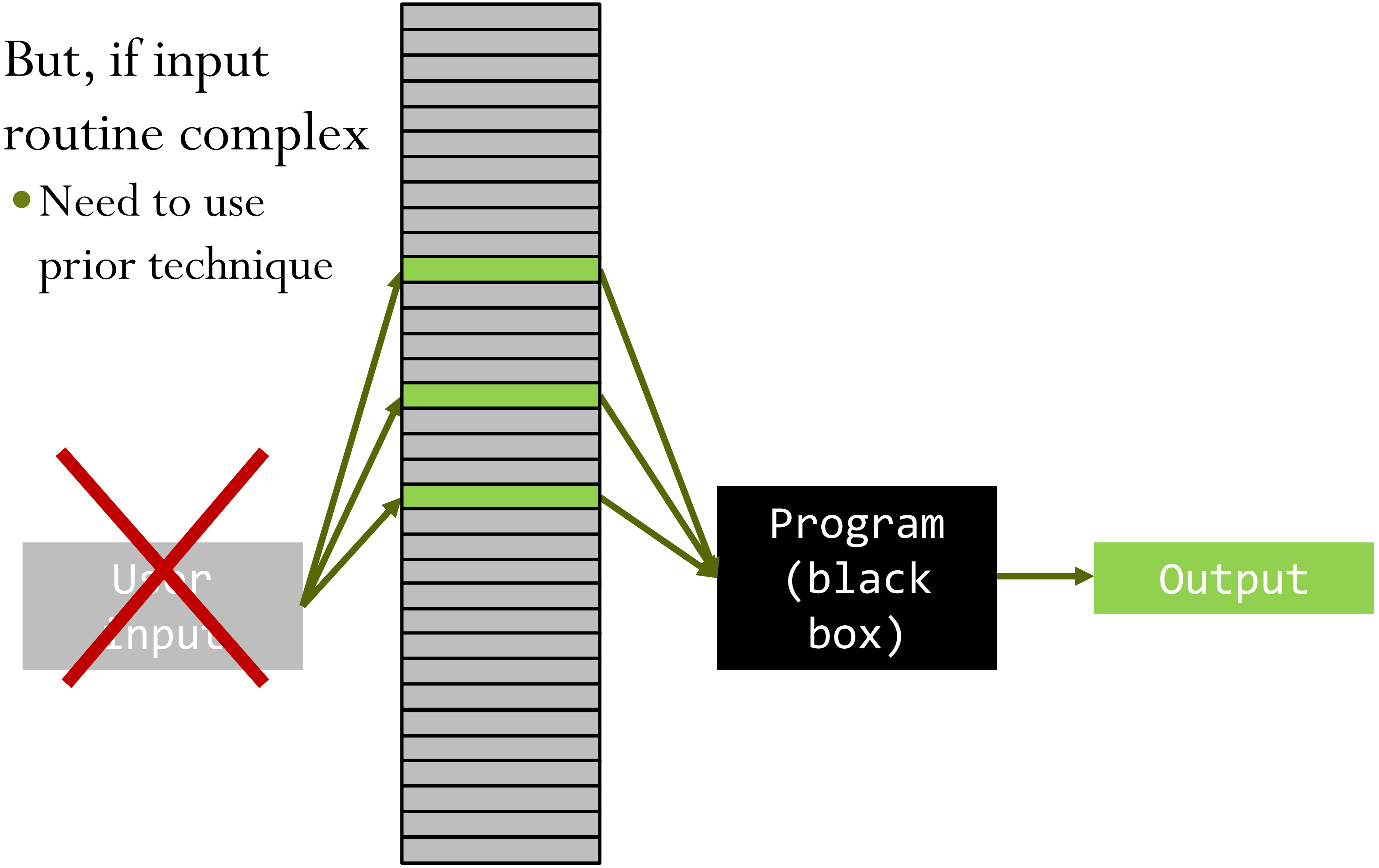
Run as before

Injecting symbols into Global Memory

- Input routine writes to global memory
  - Can try to run input routine symbolically



- But, if input routine complex
  - Need to use prior technique



- Example
  - `get_user_input` function calls `scanf` which returns values by writing them to addresses determined at compile time.
  - Instead of symbolically executing `scanf`, write symbolic values to the addresses.

```
scanf("%u %u", &a, &b)
```

Parameters (`scanf` will write user input to these addresses)

Format string
`"%u %u"`

```
push   $0xaf84128
push   $0xaf84120
push   $0x8048733
call   8048400 <scanf@plt>
```

In Angr, you can write to an address with either a concrete or a symbolic value:

- `state.memory.store(0xaf84120, 0x1)`

- `state.memory.store(0xaf84120, my_bitvector)`

will write the symbolic value `my_bitvector` to `0xaf84120`.

Angr CTF level

- `05_angr_symbolic_memory`

05_angr_symbolic_memory

- 4 strings used as password via a formatted `scanf`
  - Each read into a distinct buffer in memory
  - Can you identify their individual locations?
- Use symbolic memory to set them after input

```
0x080485e2      83ec0c          sub esp, 0xc
0x080485e5      68b8fa290a      push 0xa29fab8
0x080485ea      68b0fa290a      push 0xa29fab0
0x080485ef      68a8fa290a      push 0xa29faa8
0x080485f4      68a0fa290a      push obj.user_input        ; 0xa29faa0
0x080485f9      6833870408      push str._8s__8s__8s__8s   ; 0x8048733 ; "%8s %8s %8s %8s"
0x080485fe      e8fdfdffff      call sym.imp.__isoc99_scanf ;[2]
0x08048603      83c420          add esp, 0x20
```

05_angr_symbolic_memory

```
start_address = ???
initial_state = project.factory.blank_state(addr=start_address)
```

Start at address when symbolic values are needed after ***scanf***

```
password0 = claripy.BVS('password0', ???)
...
```

Define symbolic bitvector for each part of password based on its size in bits.

```
password0_address = ???
initial_state.memory.store(password0_address, password0)
...
```

Attach each symbolic bitvector to its corresponding memory locations as passed to ***scanf***

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

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

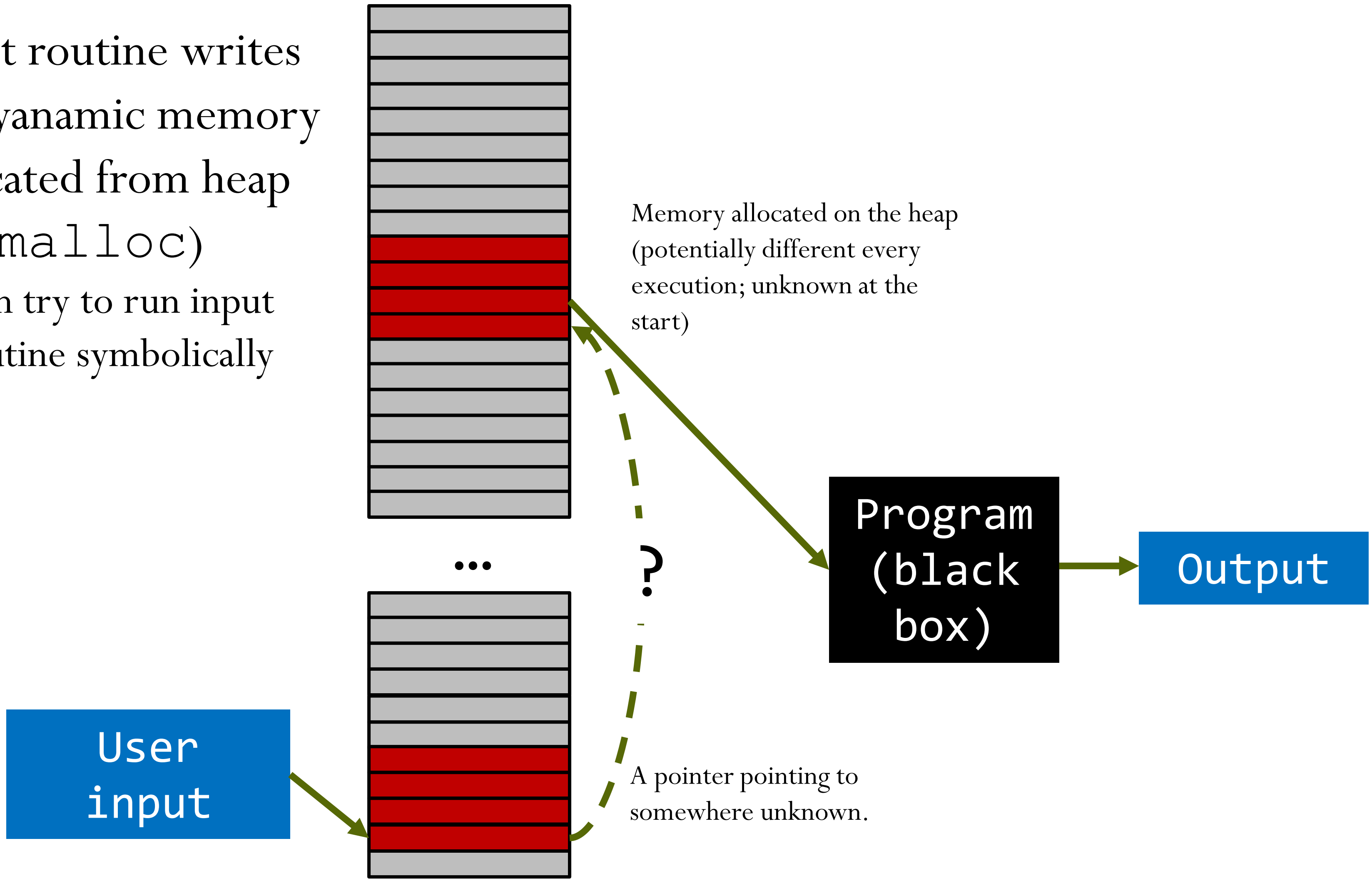
Run as before

```
if simulation.found:
  solution_state = simulation.found[0]

  solution0 = solution_state.solver.eval(password0,cast_to=bytes).decode()
  ...
```

Check if solution found. If so, solve the constraints to find an input that satisfies them. Note ***eval()*** returns an integer by default. To concretize into a byte string, use the ***cast_to*** named parameter

Injecting symbols into Dynamic Memory

- Input routine writes to dyanamic memory allocated from heap (via `malloc`)
  - Can try to run input routine symbolically



- Example

scanf writes to the address stored in the pointer located at 0xaf84dd8 that has been allocated via malloc

```
mov     0xaf84dd8,%edx
push    %edx
push    $0x8048843
call    8048460 <scanf@plt>
```

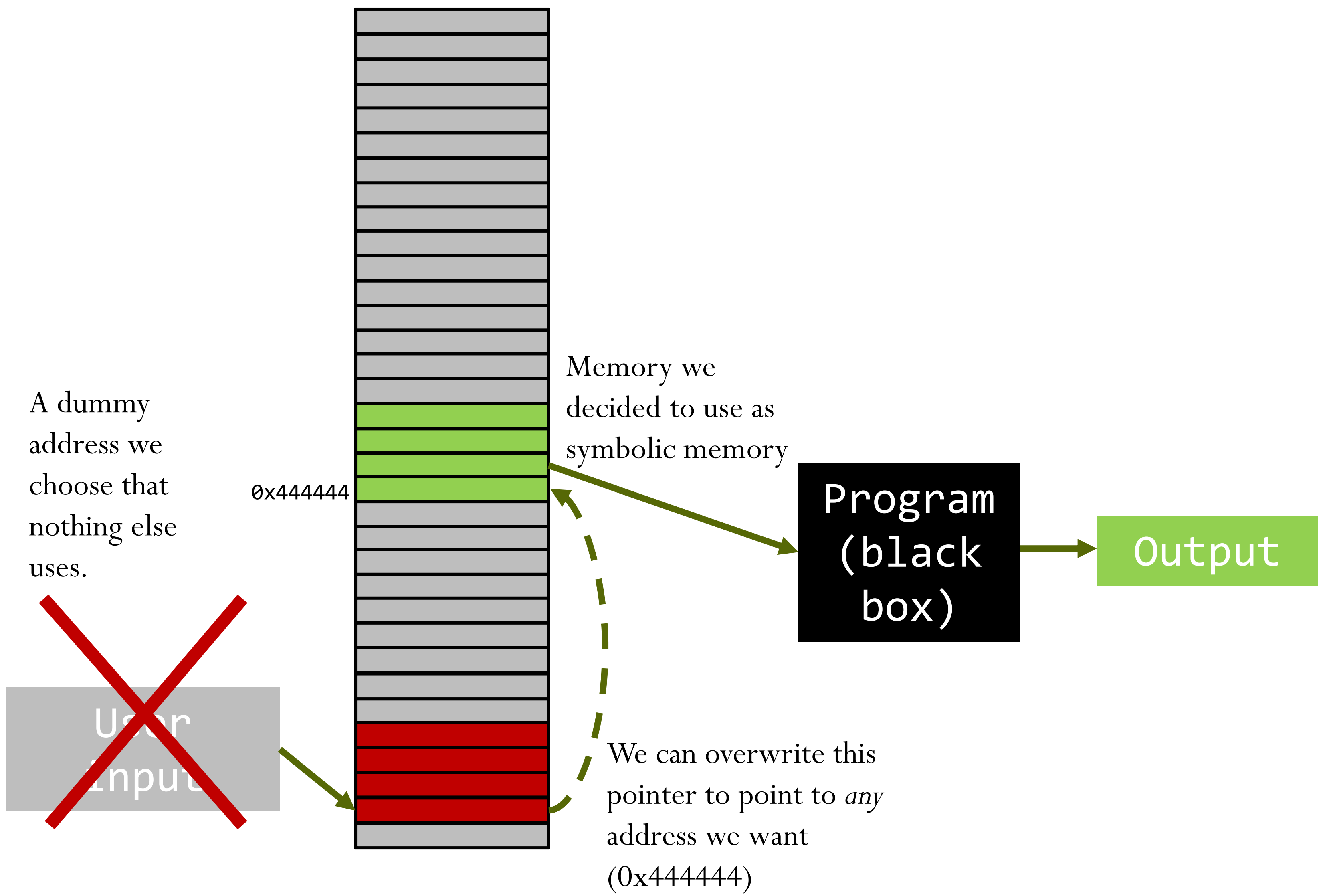
- Cannot determine the address to which scanf writes because it is stored in a pointer returned from the heap (e.g. the address stored in 0xaf84dd8)
  - Overwrite the value of the pointer to point to an unused location of your choice (in this example, 0x44444444)

    ```
    state.memory.store(0xaf84dd8, 0x444444)
    ```

  - Then set memory location to symbolic value

    ```
    state.memory.store(0x444444, my_bitvector)
    ```

  - Pointer at 0xaf84dd8 now points to 0x44444444, which stores your bitvector.


A dummy address we choose that nothing else uses.
0x444444
Memory we decided to use as symbolic memory
Program (black box)
Output
User Input
We can overwrite this pointer to point to *any* address we want (0x444444)

Angr CTF level

- 06_angr_symbolic_dynamic_memory

06_angr_symbolic_dynamic_memory

- 2 buffers dynamically allocated

```
0x08048637      6a09            push 9
0x08048639      e8d2fdffff      call sym.imp.malloc
0x0804863e      83c410          add esp, 0x10
0x08048641      a374ef2d0a      mov dword [obj.buffer0], eax
0x08048646      83ec0c          sub esp, 0xc
0x08048649      6a09            push 9
0x0804864b      e8c0fdffff      call sym.imp.malloc
0x08048650      83c410          add esp, 0x10
0x08048653      a37cef2d0a      mov dword [obj.buffer1], eax
```

- Pointers to the malloc'd regions are set (`obj.buffer1` at `0xa2def74`) and (`obj.buffer2` at `0xa2def7c`)

```
0x8048637 <main+20>    push   $0x9
0x8048639 <main+22>    call   0x8048410 <malloc@plt>
0x804863e <main+27>    add    $0x10,%esp
0x8048641 <main+30>    mov    %eax,0xa2def74
0x8048646 <main+35>    sub    $0xc,%esp
0x8048649 <main+38>    push   $0x9
0x804864b <main+40>    call   0x8048410 <malloc@plt>
0x8048650 <main+45>    add    $0x10,%esp
0x8048653 <main+48>    mov    %eax,0xa2def7c
```

- Password values read into each via a formatted `scanf`

```
0x08048692      8b157cef2d0a    mov edx, dword [obj.buffer1]
0x08048698      a174ef2d0a      mov eax, dword [obj.buffer0]
0x0804869d      83ec04          sub esp, 4
0x080486a0      52              push edx
0x080486a1      50              push eax
0x080486a2      6863880408      push str.8s__8s
0x080486a7      e8b4fdffff      call sym.imp.__isoc99_scanf
0x080486ac      83c410          add esp, 0x10
0x080486af      c745f4000000.   mov dword [var_ch], 0
```

- Want to inject symbolic memory after `scanf` and execute from there

- Do the addresses of the passwords on the heap matter?
  - No
- Are they known to us ahead of time?
  - No
- Strategy
  - Fix arbitrary locations in heap memory to store passwords being read in (e.g. `0x444444` and `0x444454`)
  - Create symbolic bitvectors for each password and insert them at the locations
  - Set pointers in program (`0xa2def74 0xa2def7c`) to the two locations above
  - Symbolically execute

06_angr_symbolic_dynamic_memory

```
start_address = ???
initial_state = project.factory.blank_state(addr=start_address)

password0 = claripy.BVS('password0', ???)
...

fake_heap_address0 = ???
pointer_to_malloc_memory_address0 = ???
initial_state.memory.store(
    pointer_to_malloc_memory_address0,
    fake_heap_address0,
    endness=project.arch.memory_endness)
...

initial_state.memory.store(fake_heap_address0, password0)
...

simulation = project.factory.simgr(initial_state)

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

if simulation.found:
  solution_state = simulation.found[0]
  solution0 = solution_state.solver.eval(password0,cast_to=bytes).decode()
  ...
  solution = ???
```

Start at address when symbolic values are needed after ***scanf*** (e.g. 0x80486af)

Declare symbolic bitvectors for each

Set location of one password (0x4444...)

Specify address of pointer pointing to it (0xa2de...)

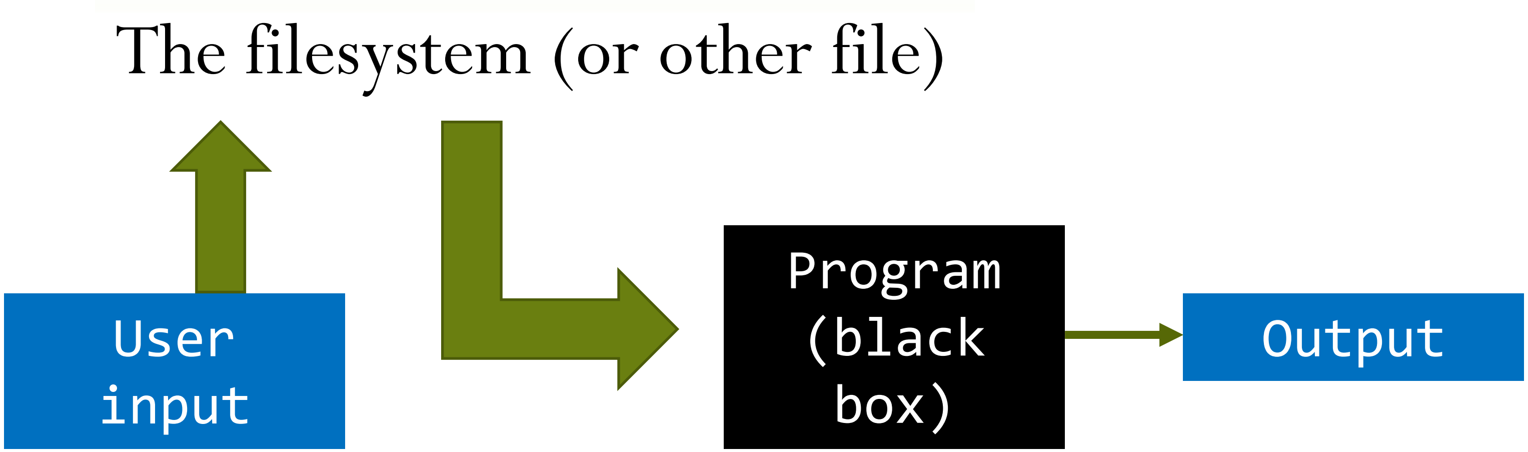
Set pointer to point to above location

Insert symbolic bitvector for one password

Check if solution found. If so, solve the constraints to find an input that satisfies them. Note ***eval()*** returns an integer by default. To concretize into a byte string, use the ***cast_to*** named parameter, then decode bytes to a Python string to print

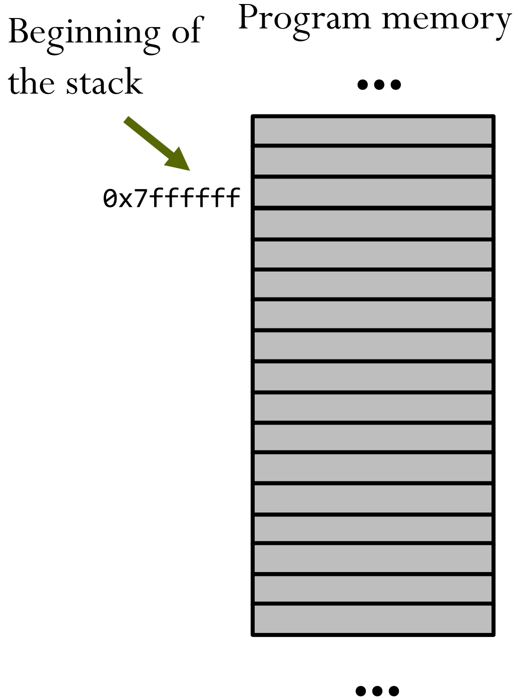
Injecting symbols via the File System

What if our user input function queries from the file system (or any other Linux file, including the network, the output of another program, /dev/urandom, etc)?



Representing a File as Memory

- Recall Angr allowing us to specify symbolic memory
- Angr allows you to specify an alternate, symbolic filesystem of your own.
  - Files treated in a way similar to a memory mapped file
  - Uses the same Python type as `state.memory` (e.g. `SimMemory`)

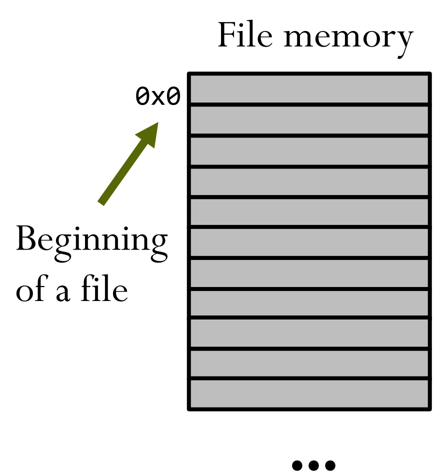


A program has access to an address space, which it uses to store instructions, the stack, the heap, and static data.

We have used it in Angr using the following functions:

```
state.memory.store(…)
state.memory.load(…)
```

Contents of files treated as if mapped into a memory region

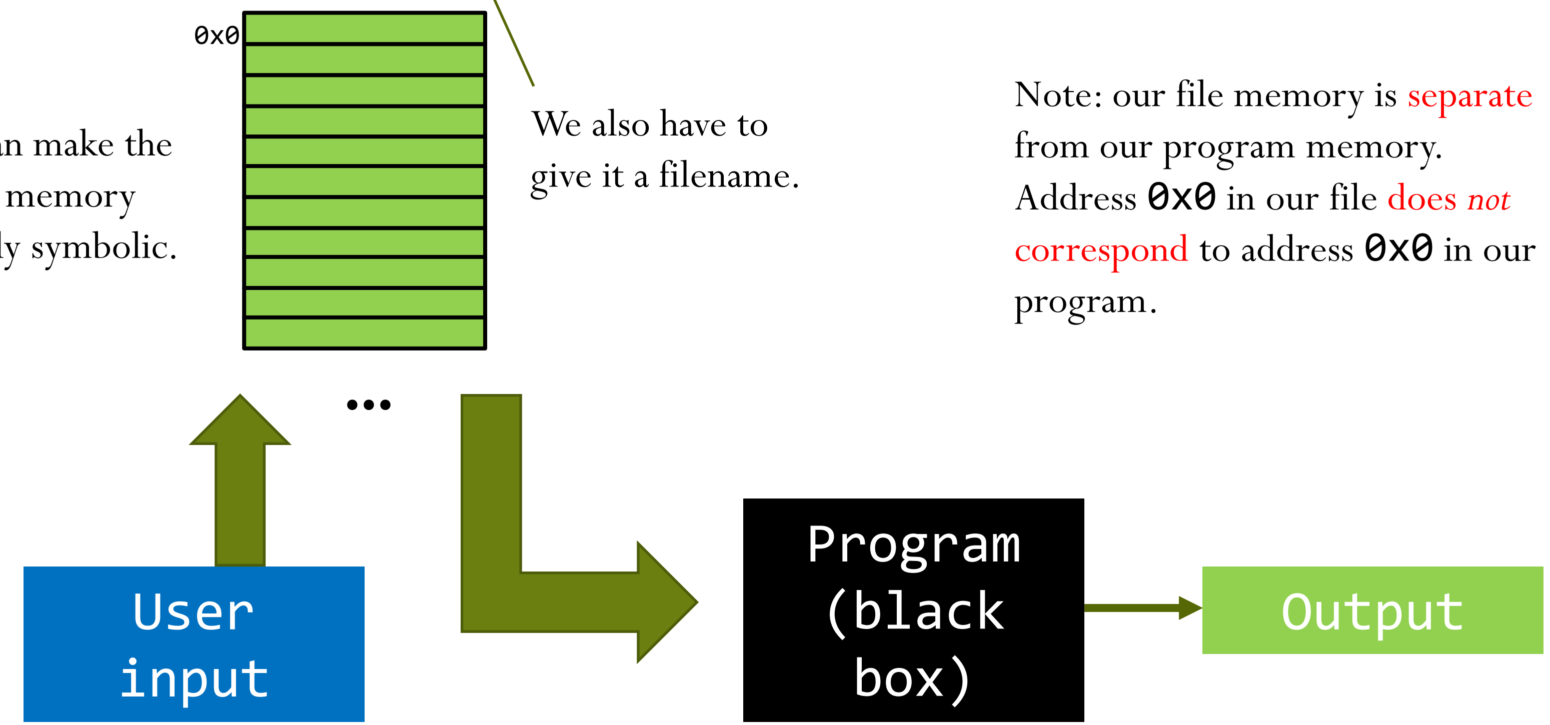


File memory (`/tmp/hello.txt`)

We also have to give it a filename.

We can make the file memory entirely symbolic.

Note: our file memory is separate from our program memory. Address `0x0` in our file does *not* correspond to address `0x0` in our program.



Angr CTF level

- `07_angr_symbolic_file`

07_angr_symbolic_file

- Password read into `obj.buffer`

```
0x08048898      68a0a00408      push obj.buffer
0x0804889d      685f8a0408      push str.64s
0x080488a2      e8f9fcffff      call sym.imp.__isoc99_scanf
```

- Stored into file-system using a random file name
- Then, program calls `fopen()` on file to `fread()` 64 bytes back to `buffer` before `complex_function` is executed

```
0x080488d3      68798a0408      push 0x8048a79
0x080488d8      684e8a0408      push str.FOQVSBZB.txt
0x080488dd      e89efcffff      call sym.imp.fopen
0x080488e2      83c410          add esp, 0x10
0x080488e5      a380a00408      mov dword [obj.fp], eax
0x080488ea      a180a00408      mov eax, dword [obj.fp]
0x080488ef      50              push eax
0x080488f0      6a40            push 0x40
0x080488f2      6a01            push 1
0x080488f4      68a0a00408      push obj.buffer
0x080488f9      e842fcffff      call sym.imp.fread
```

07_angr_symbolic_file

```
start_address = ???
initial_state = project.factory.blank_state()

filename = ???  # :string
symbolic_file_size_bytes = ???

password = claripy.BVS('password',
                       symbolic_file_size_bytes * 8)


password_file = angr.storage.SimFile(filename, content=???)

initial_state.fs.insert(filename, password_file)

simulation = project.factory.simgr(initial_state)
simulation.explore(find=is_successful, avoid=should_abort)

if simulation.found:
  solution_state = simulation.found[0]
  solution = solution_state.solver.eval(password,cast_to=bytes).decode()
```

Start just before file is opened (0x80488d3)

Find the file that is opened and the amount of data it holds in bytes

Define a symbolic bitvector for the file's contents

Create the file and set its contents to symbolic data

Insert file into the simulated file system

Run as before

Symbolic Execution CTF: Part 3

Handling Non-trivial Behavior

Motivation: A Simple Example

- Blowing up symbolic execution
  - Program iterates through 16 elements, each time it branches.
  - By the end of the loop, there will be a total of 2^{16} or 65,536 branches.
  - How many branches would it be if the code checked each bit of the input one at a time?

```
def check_all_Z(user_input):
  num_Z = 0
  for i in range(0, 16):
    if user_input[i] == 'Z':
      num_Z += 1
    else:
      pass
  return num_Z == 16
```

- Could be reduced to one branch!

```
user_input == 'ZZZZZZZZZZZZZZZZ'
```

This is the opposite problem from coverage-based fuzzing!
(e.g. AFL lab #4's path_based vs. strcmp_based)

Solutions

- There are algorithms to deduce the insight on the previous slide.
  - None work as well as human intuition for many cases.
- For complex functions, we can use Angr to replace the code with its summary in Python.
- Two approaches
  - Constraints
  - Hooks

Constraints

- Add constraints at specific execution locations

If string at `0x804a420` is `'ZZZZZZZZZZZZZZZZ'`, then `'Good Job.'` is printed

```
...
8048774:   add    %edx,%eax
8048776:   sub    $0x4,%esp
804877c:   push   $0x804a420
8048781:   call   8048460 <check_all_Z>
8048786:   add    $0x10,%esp
8048789:   test   %eax,%eax
804878b:   jne    8048794 <main+0x19f>
...
```

- Add a constraint at `0x804877c` to ensure the string at `0x804a420` is `'ZZZZZZZZZZZZZZZZ'`
  - If constraint satisfiable, add state to solution and end search.

Angr CTF level

- `08_angr_constraints`

08_angr_constraints

- Input is sent to `complex_function` to make reverse-engineering hard
- After, it is checked against a static string (`check_equals_XXX`), one character at a time to make symbolic execution hard

```
printf("Enter the password: ");
scanf("%16s", buffer);
for (int i=0; i<16; ++i) {
  buffer[i] = complex_function(buffer[i], -i+15);
}

if (!check_equals_AABBCCDDEEFFGGHH(buffer, 16)) {
  printf("Try again.\n");
} else {
  printf("Good Job.\n");
}
```

```
#define REFERENCE_PASSWORD "AABBCCDDEEFFGGHH"
int check_equals_AABBCCDDEEFFGGHH(char* to_check, size_t length) {
  uint32_t num_correct = 0;
  for (int i=0; i<length; ++i) {
    if (to_check[i] == password[i]) {
      num_correct += 1;
    }
  }
  return num_correct == length;
}
```

- `check_equals...` function equivalent to

  `if *to_check == "AABBCCDDEEFFGGHH"` but causes 2^16 branches

  - Need input that generates `"AABBCCDDEEFFGGHH"` from `complex_function`
  - Use angr to stop before `check_equals`, then constrain `to_check` variable to be equal to `"AABBCCDDEEFFGGHH"`

```
printf("Enter the password: ");
scanf("%16s", buffer);
for (int i=0; i<16; ++i) {
  buffer[i] = complex_function(buffer[i], -i+15);
}

if (!check_equals_AABBCCDDEEFFGGHH(buffer, 16)) {
  printf("Try again.\n");
} else {
  printf("Good Job.\n");
}
```

```
#define REFERENCE_PASSWORD "AABBCCDDEEFFGGHH"
int check_equals_AABBCCDDEEFFGGHH(char* to_check, size_t length) {
  uint32_t num_correct = 0;
  for (int i=0; i<length; ++i) {
    if (to_check[i] == password[i]) {
      num_correct += 1;
    }
  }
  return num_correct == length;
}
```

- Password buffer 16 bytes stored at `0x804a050`
- Can skip `scanf` by making buffer symbolic and starting just after `scanf` returns

```
  0x08048612      83c410       add esp, 0x10
  0x08048615      83ec08       sub esp, 0x8
  ; DATA XREF from 0x0804a050 (unk)
  0x08048618      6850a00408   push sym.buffer ; 0x0804a050
  ; DATA XREF from 0x08048763 (fcn.08048738)
  0x0804861d      6863870408   push str.16s ; 0x08048763
  0x08048622      e8a9fdffff   call 0x1080483d0 ; (sym.imp.__isoc99_scanf) ;[1]
     sym.imp.__isoc99_scanf(unk, unk)
  0x08048627      83c410       add esp, 0x10
```

- Want to constrain symbolic buffer at `0x804a050` to be `"BWYRUBQCMVSBRGFU"` at `0x8048671`

```
  ; DATA XREF from 0x00000010 (unk)
  0x08048671      6a10         push 0x10 ; 0x00000010
  ; DATA XREF from 0x0804a050 (unk)
  0x08048673      6850a00408   push sym.buffer ; 0x0804a050
  0x08048678      e8e8feffff   call 0x108048565 ; (fcn.08048565) ;[2]
     fcn.08048565(unk, unk) ; sym.check_equals_BWYRUBQCMVSBRGFU
  0x0804867d      83c410       add esp, 0x10
  0x08048680      85c0         test eax, eax
```

08_angr_constraints

```
start_address = ???
initial_state = project.factory.blank_state(addr=start_address)

password = claripy.BVS('password', ???)

password_address = ???
initial_state.memory.store(password_address, password)

simulation = project.factory.simgr(initial_state)

address_to_check_constraint = ???
simulation.explore(find=address_to_check_constraint)

if simulation.found:
  solution_state = simulation.found[0]

  constrained_parameter_address = ???
  constrained_parameter_size_bytes = ???
  constrained_parameter_bitvector = solution_state.memory.load(
    constrained_parameter_address,
    constrained_parameter_size_bytes
  )

  constrained_parameter_desired_value = ??? # :string (encoded)

  solution_state.add_constraints(
    constrained_parameter_bitvector == constrained_parameter_desired_value
  )

  solution = ???
```

← Start at address when symbolic value is needed after scanf

← Create symbolic bitvector

← Get address of password and store symbolic value at it

← Stop exploration just before “check” function that blows up symbolic execution

← Once found, rind address of value to constrain and its size in bytes (e.g. password above)

← Read bitvector at address

← Specify contraint value encoded as UTF-8

← Apply constraint to state

← Concretize symbol as a string to obtain solution

Hooks

- Another approach for avoiding state explosion
- Skip problematic code and implement an alternative in Python
  - Via function hooks
  - Via SimProcedures (syntactic sugar for function hooks)

Hooks

We want to skip this and instead run our own code.

```
...
8048774:   add    %edx,%eax
8048776:   sub    $0x4,%esp
804877c:   push   $0x804a420
8048781:   call   8048460 <check_all_Z>
8048786:   add    $0x10,%esp
8048789:   test   %eax,%eax
804878b:   jne    8048794 <main+0x19f>
...
```

- Defining a hook
  - Specify an address to 'hook'
  - Specify the number of instruction bytes to skip
  - Specify a Python function to run in place of the skipped instructions

Hook Walkthrough

The parameter to `check_all_Z`

Call to `check_all_Z`

```
...
8048774:  add    %edx,%eax
8048776:  sub    $0x4,%esp
804877c:  push   $0x804a420
8048781:  call   8048460 <check_all_Z>
8048786:  add    $0x10,%esp
8048789:  test   %eax,%eax
804878b:  jne    8048794 <main+0x19f>
...
```

Implement hook to replace the call to `check_all_Z`, with our own check function

```
def replacement_check_all_Z():
  eax = (*0x804a420 == 'ZZZZZZZZZZZZZZZZ')
```

Return values are stored in `eax`

The parameter to `check_all_Z`

Hook Walkthrough

```
...
8048774:   add     %edx,%eax
8048776:   sub     $0x4,%esp
804877c:   push    $0x804a420
8048781:   call    8048460 <check_all_Z>
8048786:   add     $0x10,%esp
8048789:   test    %eax,%eax
804878b:   jne     8048794 <main+0x19f>
...
```

```
def replacement_check_all_Z():
  eax = (*0x804a420 == 'ZZZZZZZZZZZZZZZZ’)
```

Call: `binary.hook(0x8048781, length=5, replacement_check_all_Z)`

Address we want to hook

`call` instruction represented with 5 bytes in memory.

Can set to 0 to run hook in addition to original code

Function to replace it with (runs when we reach our hook)

Hook Walkthrough

Call: `binary.hook(0x8048781, length=5, replacement_check_all_Z)`

```
              …
8048774:   add     %edx,%eax
8048776:   sub     $0x4,%esp
804877c:   push    $0x804a420

5 bytes
8048781:   call    8048460 <check_all_Z>   eax = (*0x804a420 == 'ZZZZZZZZZZZZZZZZ’)

8048786:   add     $0x10,%esp
8048789:   test    %eax,%eax
804878b:   jne     8048794 <main+0x19f>

              …
```

Angr CTF level

- `09_angr_hooks`

09_angr_hooks

- `check_equals`... causes symbolic execution engine to blow up

```
0x080486c3      6a10            push 0x10                                  ; 16
0x080486c5      6844a00408      push obj.buffer                            ; 0x804a044
0x080486ca      e8edfeffff      call sym.check_equals_OSIWHBXIFOQVSBZB ;[2]
```

- Replace with a Python hook for performing the check

09_angr_hooks

```
check_equals_called_address = ???
instruction_to_skip_length = ???

@project.hook(check_equals_called_address, length=instruction_to_skip_length)
def skip_check_equals_(state):

  user_input_buffer_address = ??? # :integer, probably hexadecimal
  user_input_buffer_length = ???

  user_input_string = state.memory.load(
    user_input_buffer_address,
    user_input_buffer_length
  )

  check_against_string = ??? # :string


  state.regs.eax = claripy.If(
    user_input_string == check_against_string,
    claripy.BVV(1, 32),
    claripy.BVV(0, 32)
  )
```

Address of call instruction that needs to be hooked (0x80486ca) and number of bytes to skip

Hook function to run instead

Define function

Specify buffer address (0x804a044) and length

Read current value of string from simulation state

Specify string value to check_equals encoded as UTF-8

Perform check and set return value in eax based on it

Common uses of hooks

- Injecting symbolic values partway through the execution.
- Replacing unsupported instructions (for example, most syscalls).
- Replacing complex functions
  - Extremely common
  - Motivates…

SimProcedure

- Syntactic sugar to make it easier to replace functions with a summary in Python

- But first…a review of function invocation
  1. Push parameters to the stack
  2. Push return address to the stack
  3. Jump to function address
  4. Handle parameters (if necessary)*
  5. Execute function
  6. Write return value to appropriate location
  7. Pop return address and jump to it
  8. Pop parameters

SimProcedure structure

1. Push parameters to the stack
2. Push return address to the stack
3. Jump to function address

Hooks here, at the beginning of the function address

Allows user to replace call directly in Python

4. Handle parameters
5. Execute function
6. Write return value to appropriate location

Done automatically

Skips all instructions until the function is about to return, and resumes execution here

7. Pop return address and jump to it
8. Pop parameters

Comparison

Hooks:

```
def replacement_check_all_Z():
    eax = (*0x804a420 == 'ZZZZZZZZZZZZZZZZ')
```

Manually set return value

Handle parameters ourselves

```
...
8048774:   add    %edx,%eax
8048776:   sub    $0x4,%esp
804877c:   push   $0x804a420
8048781:   call   8048460 <check_all_Z>
8048786:   add    $0x10,%esp
8048789:   test   %eax,%eax
804878b:   jne    8048794 <main+0x19f>
...
```

SimProcedures:

Uses Pythonic function arguments

```
def replacement_check_all_Z(input):
  return input == 'ZZZZZZZZZZZZZZZZ'
```

Uses Python's return functionality

SimProcedures in Practice

- Use SimProcedures to
  - Replace anything you fully understand and don't want to test
  - Replace functions not supported easily with Angr.

- Any and every function that meets the above criteria should be replaced with a SimProcedure to reduce complexity (e.g. exponential number of states)

- Currently, SimProcedures for a subset of `libc` is included with Angr.

10_angr_simprocedures

- Trolling constraints

```
0x08048ed6    e81af7ffff    call 0x1080485f5 ; (fcn.080485f5) ;[1]
   fcn.080485f5(unk, unk) ; sym.check_equals_WQNDNKKWAWOLXBAC
0x08048edb    83c410        add esp, 0x10
0x08048ede    8945d4        mov [ebp-0x2c], eax
0x08048ee1    e9881a0000    jmp loc.0804a96e ;[2]
0x08048ee6    83ec08        sub esp, 0x8
; DATA XREF from 0x00000010 (unk)
0x08048ee9    6a10          push 0x10 ; 0x00000010
```

```
% objdump -d 10_angr_simprocedures | egrep check_ | egrep call | head -5
 8048771: call   80485f5 <check_equals_WQNDNKKWAWOLXBAC>
 804878a: call   80485f5 <check_equals_WQNDNKKWAWOLXBAC>
 80487ac: call   80485f5 <check_equals_WQNDNKKWAWOLXBAC>
 80487c5: call   80485f5 <check_equals_WQNDNKKWAWOLXBAC>
 80487f0: call   80485f5 <check_equals_WQNDNKKWAWOLXBAC>
% objdump -d 10_angr_simprocedures | egrep check_equals_WQNDNKKWAW | egrep call | wc -l
256
```

```
0x08048f0d    8d45e3        lea eax, [ebp-0x1d]
0x08048f10    50            push eax
0x08048f11    e8dff6ffff    call 0x1080485f5 ; (fcn.080485f5) ;[6]
   fcn.080485f5(unk, unk) ; sym.check_equals_WQNDNKKWAWOLXBAC
```

- `check_equals…` function now called 256 times
  - Manually adding constraints or hooking call instructions painful
- Instead, hook function itself using a SimProcedure

- How? Via subclass of `angr.SimProcedure`

```c
int add_if_positive(int a, int b) {
  if (a >= 0 && b >= 0) return a + b;
  else return 0;
}
```

→

```python
class ReplacementAddIfPositive(angr.SimProcedure):
  def run(self, a, b):
    if a >= 0 and b >=0:
      return a + b
    else:
      return 0
```

- But, needs to handle symbolic values, not concrete ones

10_angr_simprocedures

```
class ReplacementCheckEquals(angr.SimProcedure):
  def run(self, to_check, ...???):
    user_input_buffer_address = ???
    user_input_buffer_length = ???

    user_input_string = self.state.memory.load(
      user_input_buffer_address,
      user_input_buffer_length
    )

    check_against_string = ???

    return claripy.If(???, ???, ???)
```

Create SimProcedure

Define run method that has the same function prototype as check_equals… (e.g. pointer to buffer and length).

Read in function parameters into variables

Load symbolic value that is at the location to_check

String checked by check_equals… encoded as UTF-8

Perform the symbolic comparison and return a 32-bit BitVector of 0 or 1 back to symbolic execution engine. Arguments to "If" are a Boolean condition (i.e. user_input_string == "foo"), followed by the "then" and "else" expressions to return (i.e. claripy.BVV())

```
check_equals_symbol = ??? # :string
project.hook_symbol(check_equals_symbol, ReplacementCheckEquals())
```

Identify function to hook by its name

Hook it with created SimProcedure

Please take the first 10 minutes of class to fill out course evaluation

Class will begin at 3:40pm