PRAETORIAN

Symbolic Execution Workshop

HOW NOT TO BE ANGR-Y

$ id

- Cory Duplantis
- Senior Security Engineer @ Praetorian
- Avid CTF competitor - ctfhacker.com
- Sing barbershop (not the hair cutting kind)
- @ctfhacker

- Anthony Weems
- Staff Engineer @ Praetorian
- Loves appsec & home-rolled crypto
- @amlweems

$ env

- Follow along: demo.praetorian.com
- Slides available: https://github.com/praetorian-inc/bsidesaustin
- Local wireless: HowToNotBeAngr-y
  PSK: <bsides></bsides>
  Hosts: 192.168.1.[12]00
- These demo boxes are for everyone, please don’t try to take them down ;-)

Quick background

- Symbolic Execution simply means analyzing a program by watching particular input values to determine what inputs will exercise a given path

- Layman's terms: How can we traverse from Point A to Point B in a program

- Accomplished by converting an execution path into a boolean satisfiability problem and throwing the problem at a SAT solver

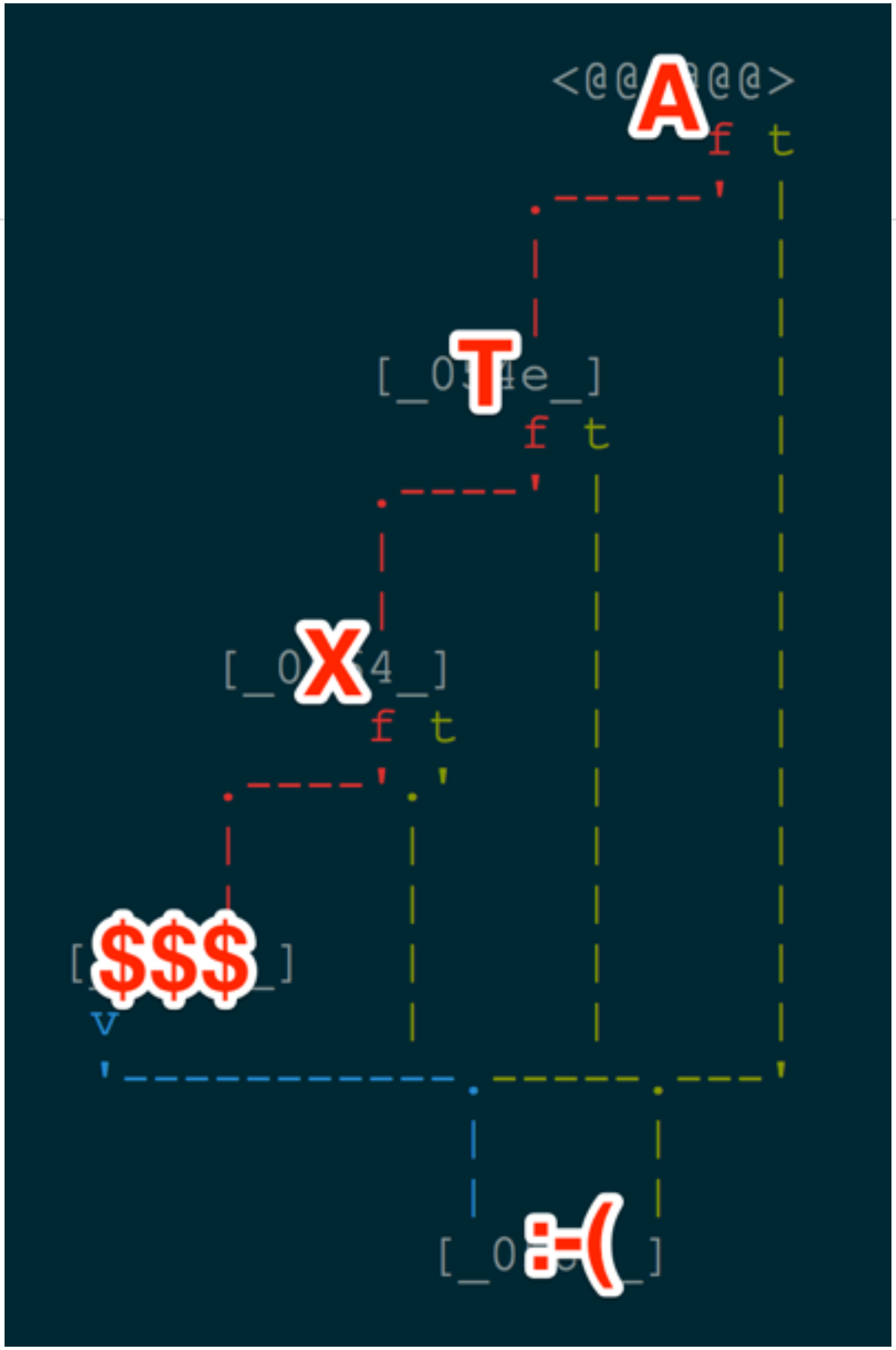
- By definition, the solution to these equations is an input that will follow the given execution path

Quick background

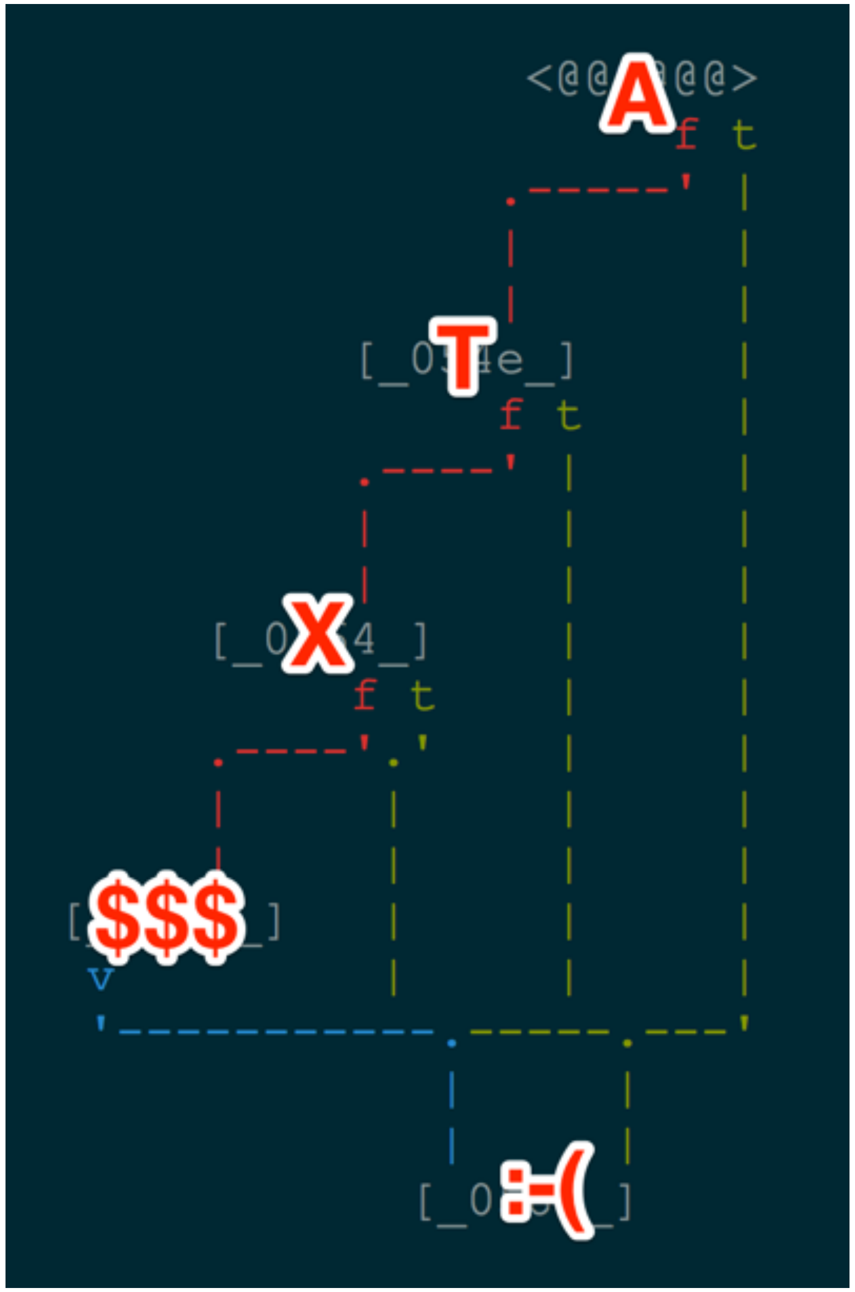
- Has been a thriving academic research topic since the 1980’s
- The problem was the computational power required for SAT solvers to solve equations in a reasonable amount of time
- Symbolic execution has been growing in popularity with the advances in publicly accessible SAT solvers (see Microsoft’s Z3)
- Usage of symbolic execution has also been seen in DARPA’s recent Cyber Grand Challenge

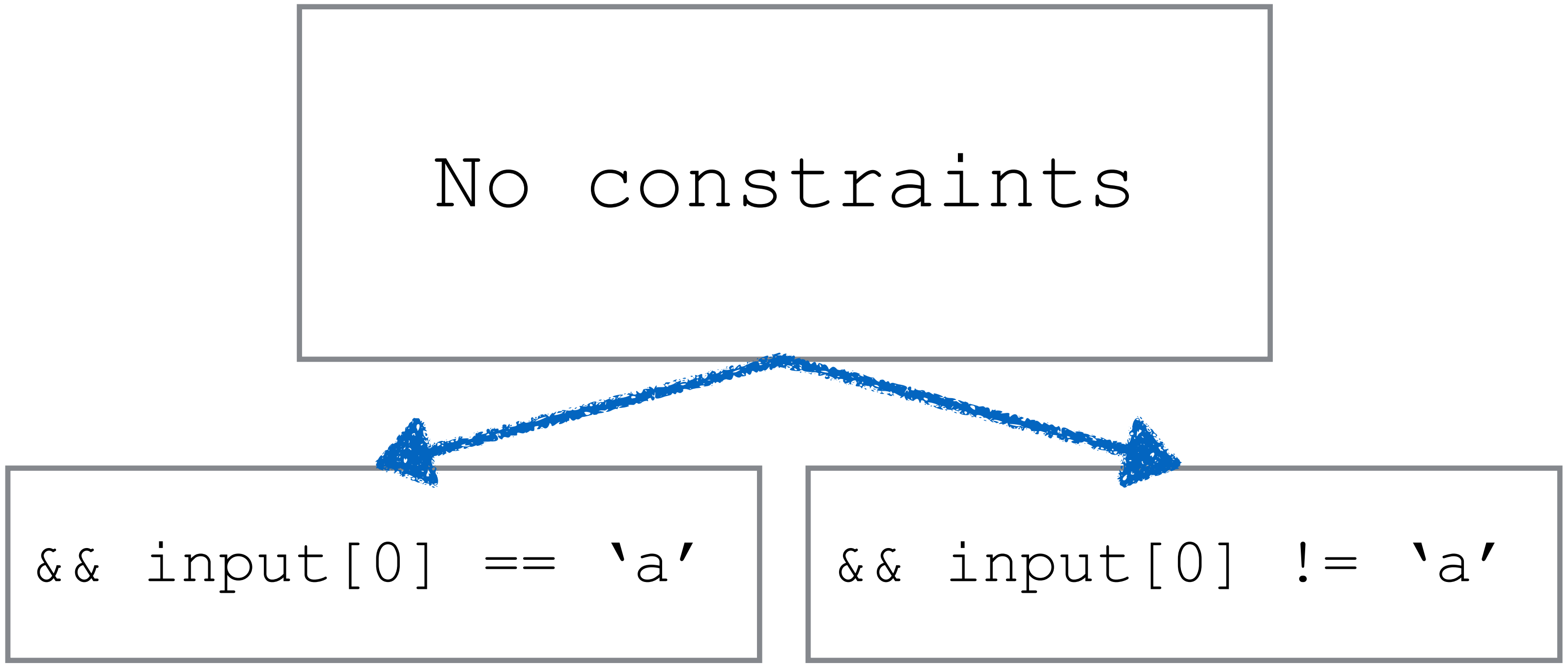
Goal: Easy crackme

```
int main(int argc, char** argv){
  if(argv[1][0] == 'a'){
    if(argv[1][1] == 't'){
      if(argv[1][2] == 'x'){
        printf("Yay BSIDES");
      }
    }
  }
}
```

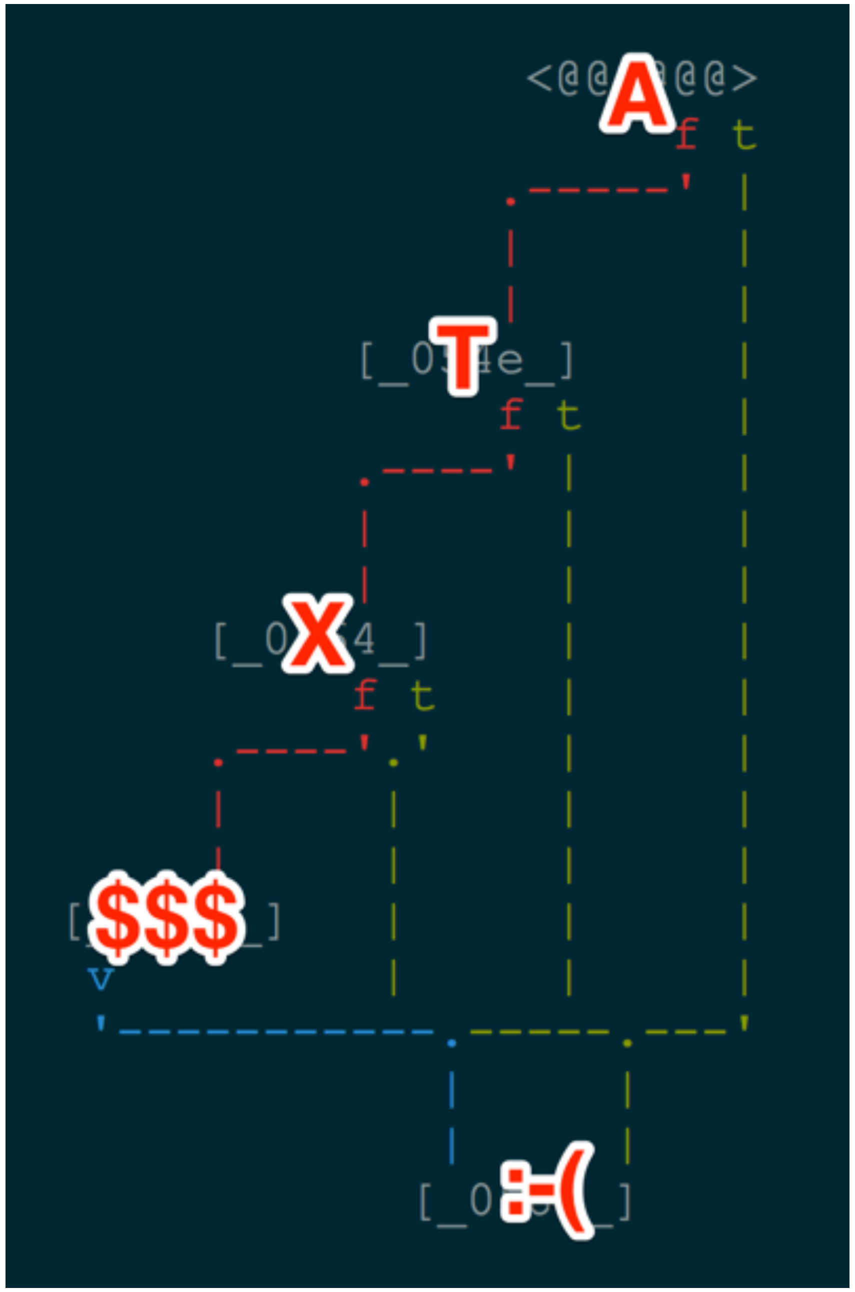


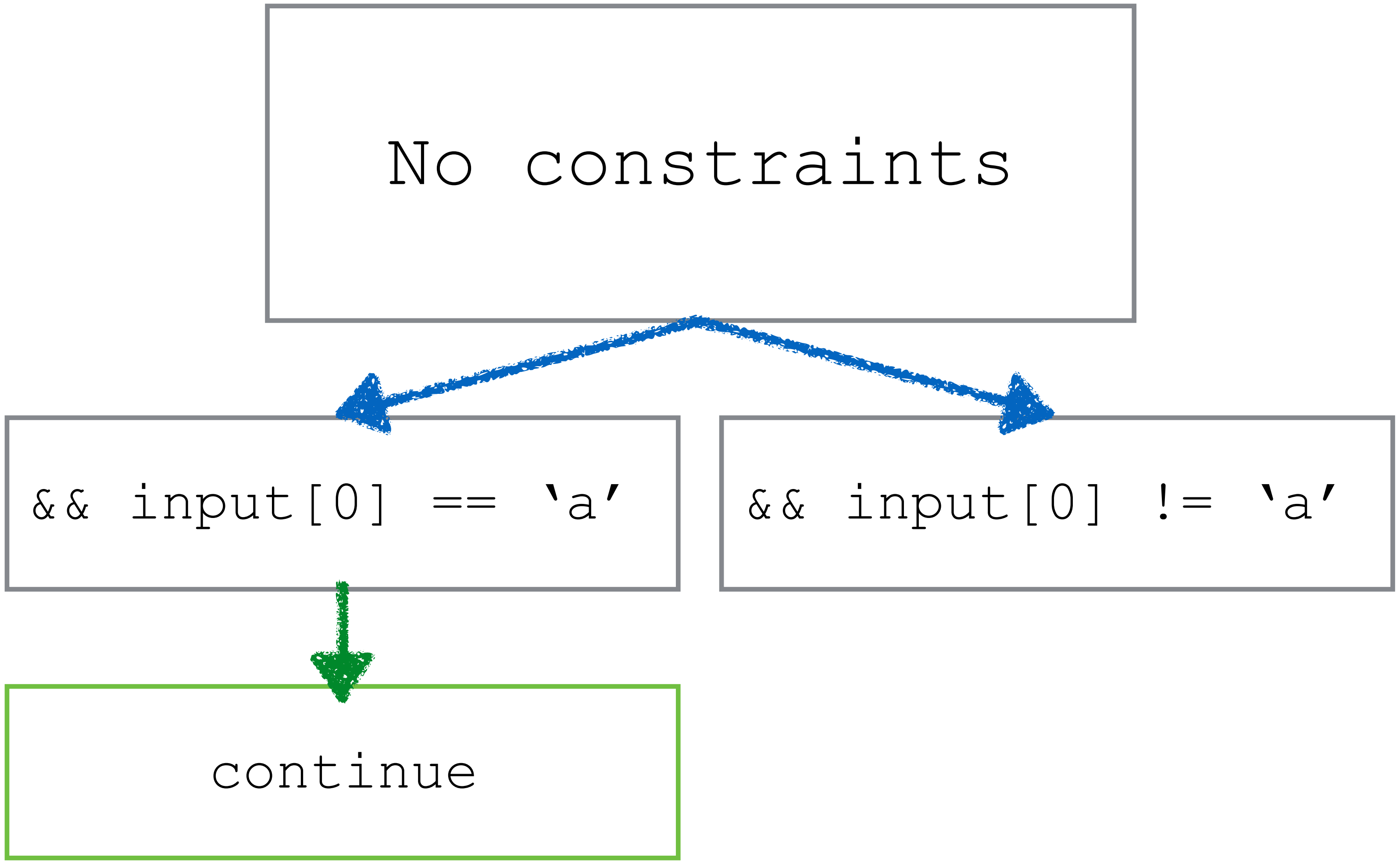
Goal: Easy crackme





Goal: Easy crackme





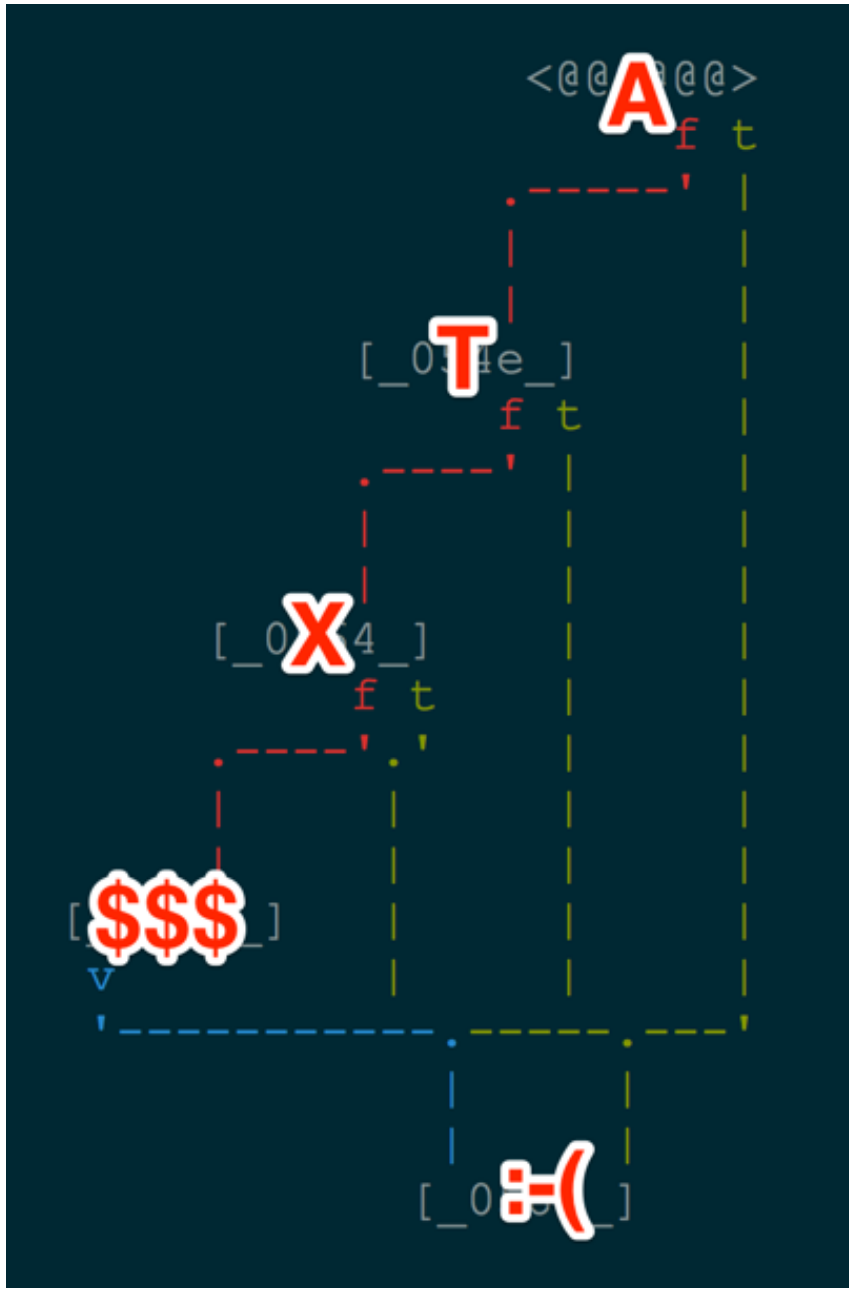
No constraints

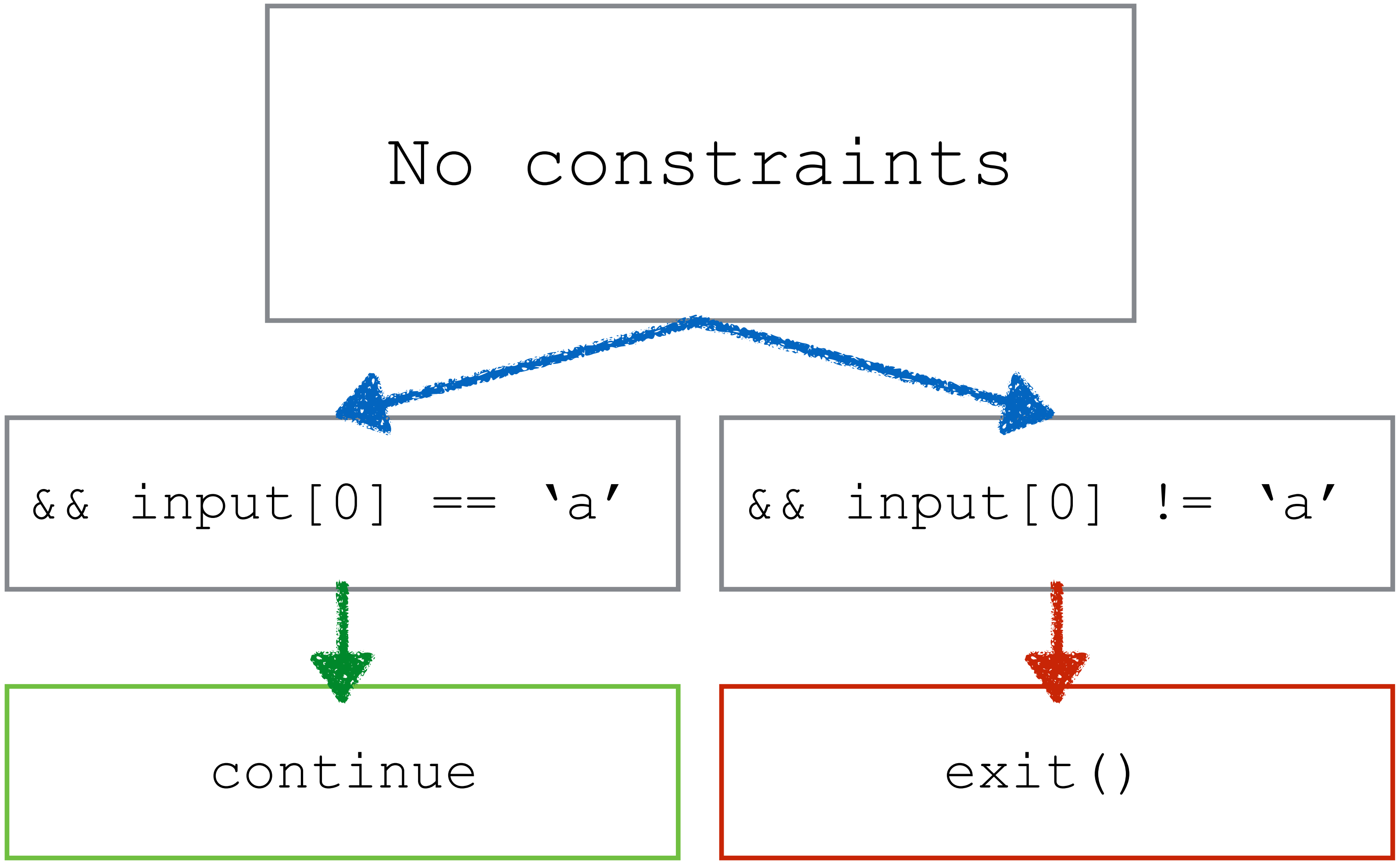
&& input[0] == 'a'

&& input[0] != 'a'

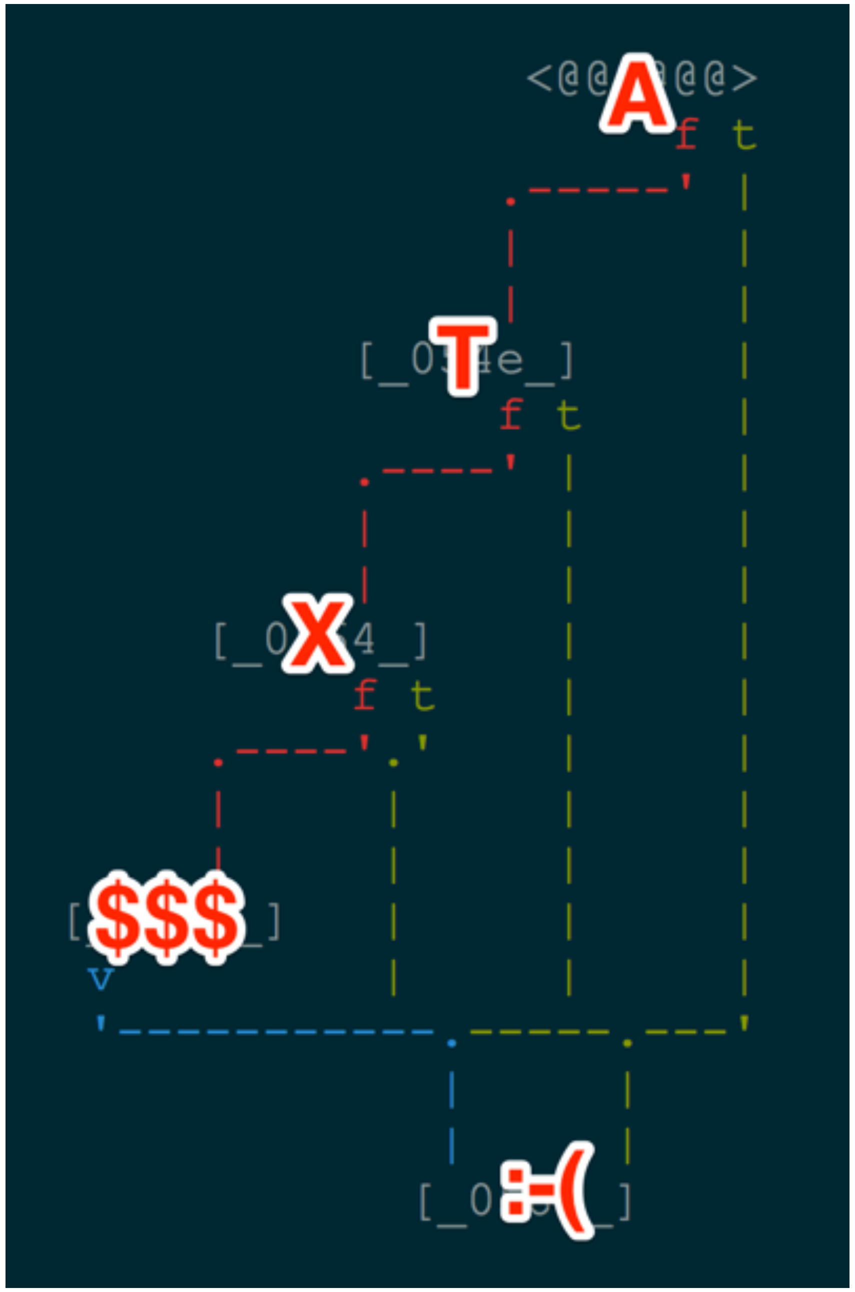
continue

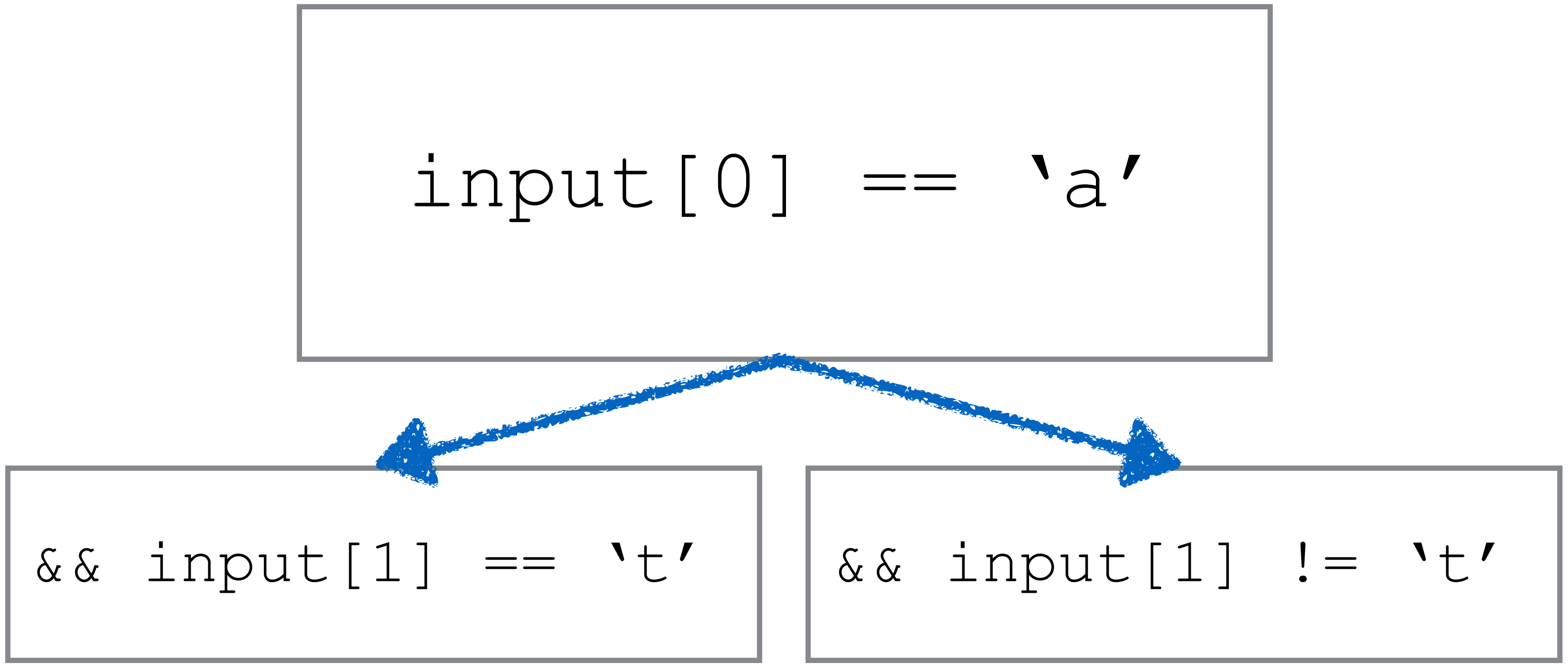
Goal: Easy crackme





Goal: Easy crackme



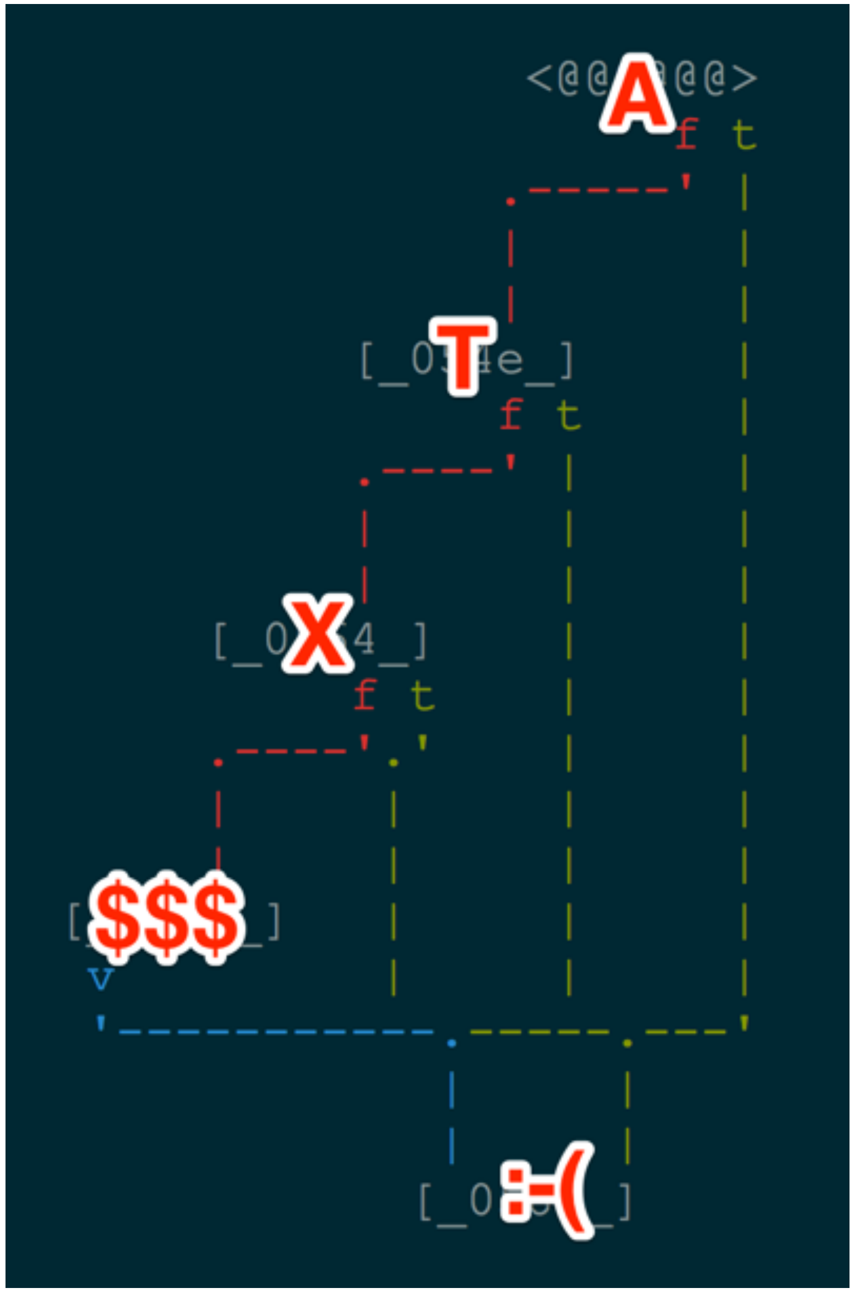


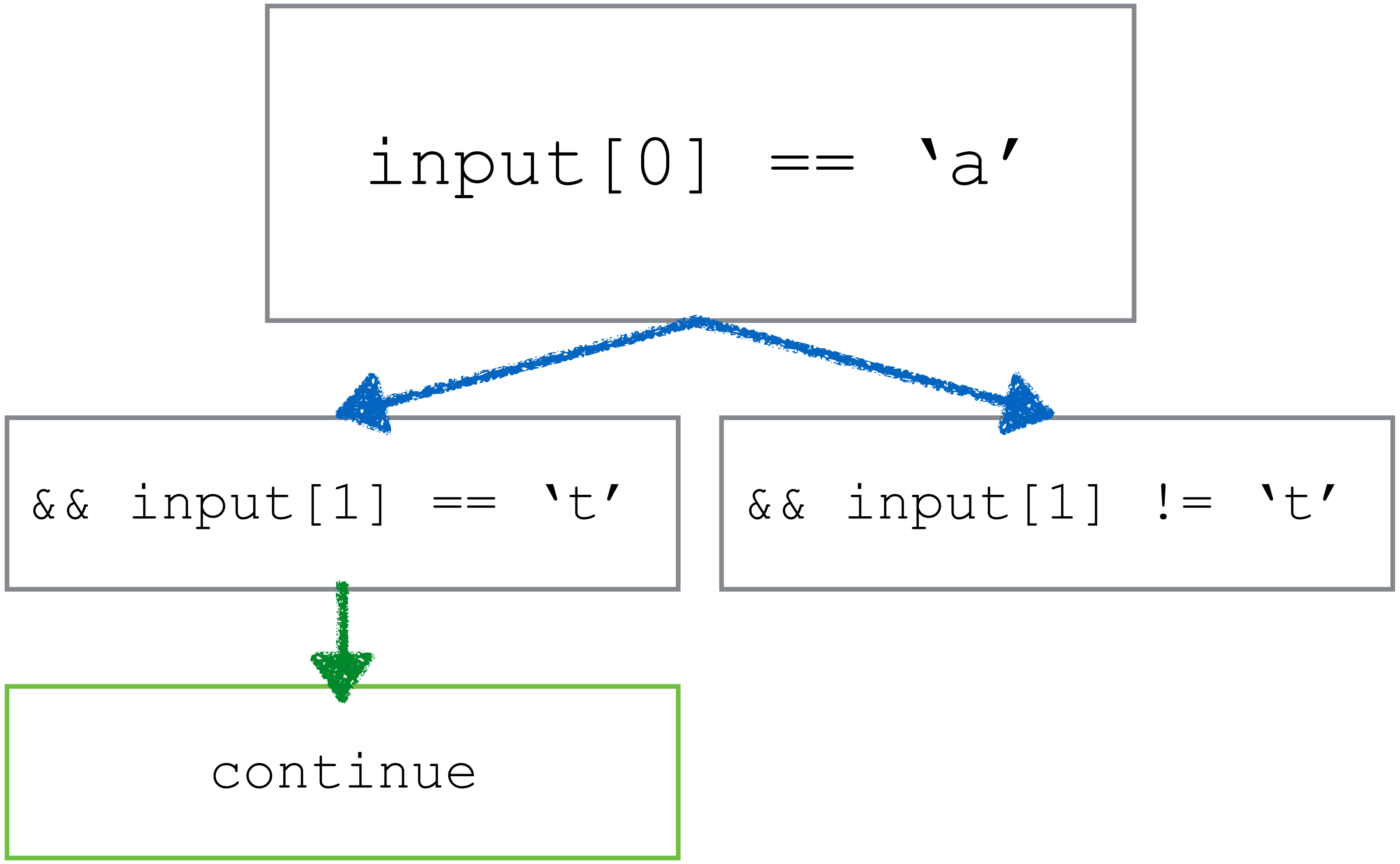
```
input[0] == 'a'
```

```
&& input[1] == 't'
```

```
&& input[1] != 't'
```

Goal: Easy crackme



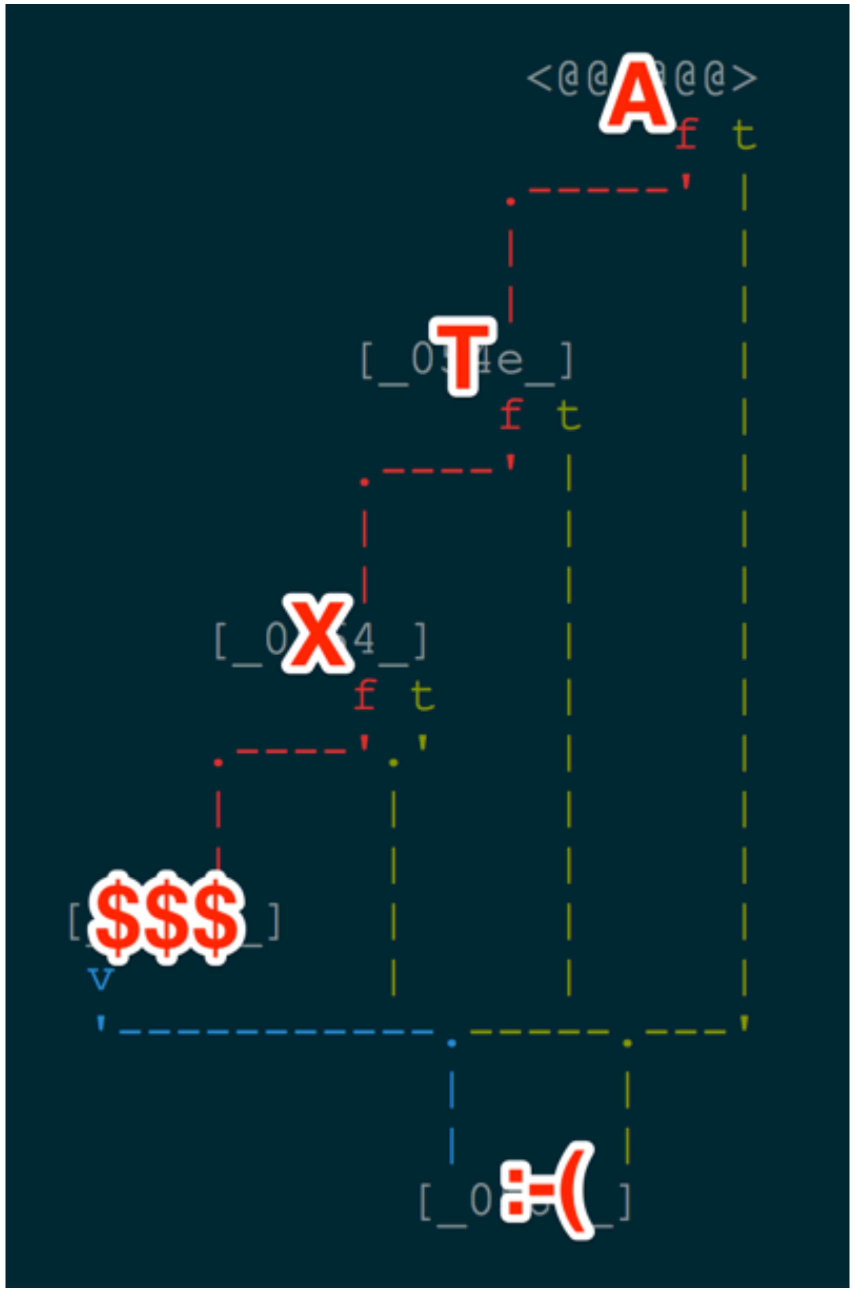

input[0] == ‘a’

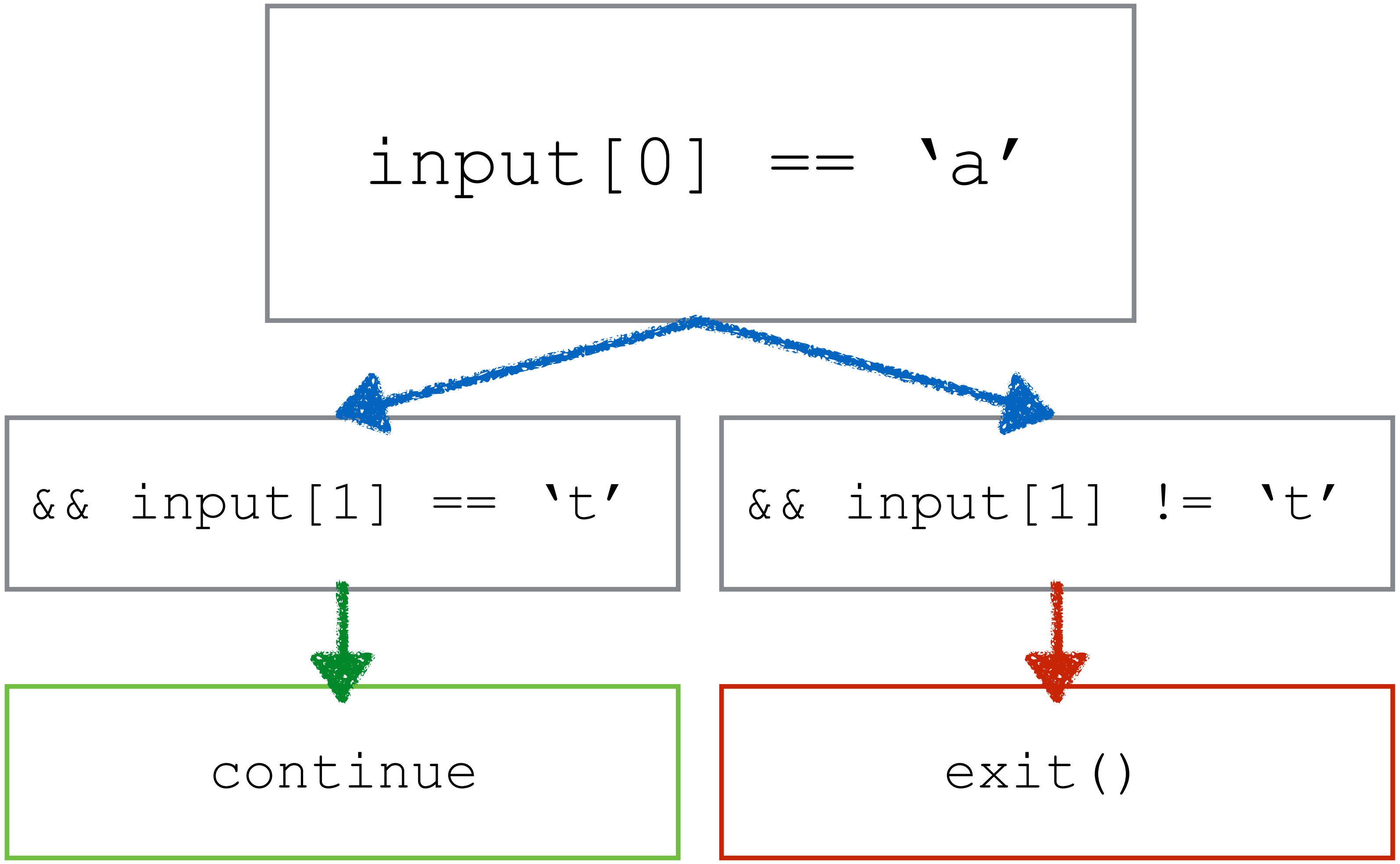
&& input[1] == ‘t’

&& input[1] != ‘t’

continue

Goal: Easy crackme




input[0] == 'a'

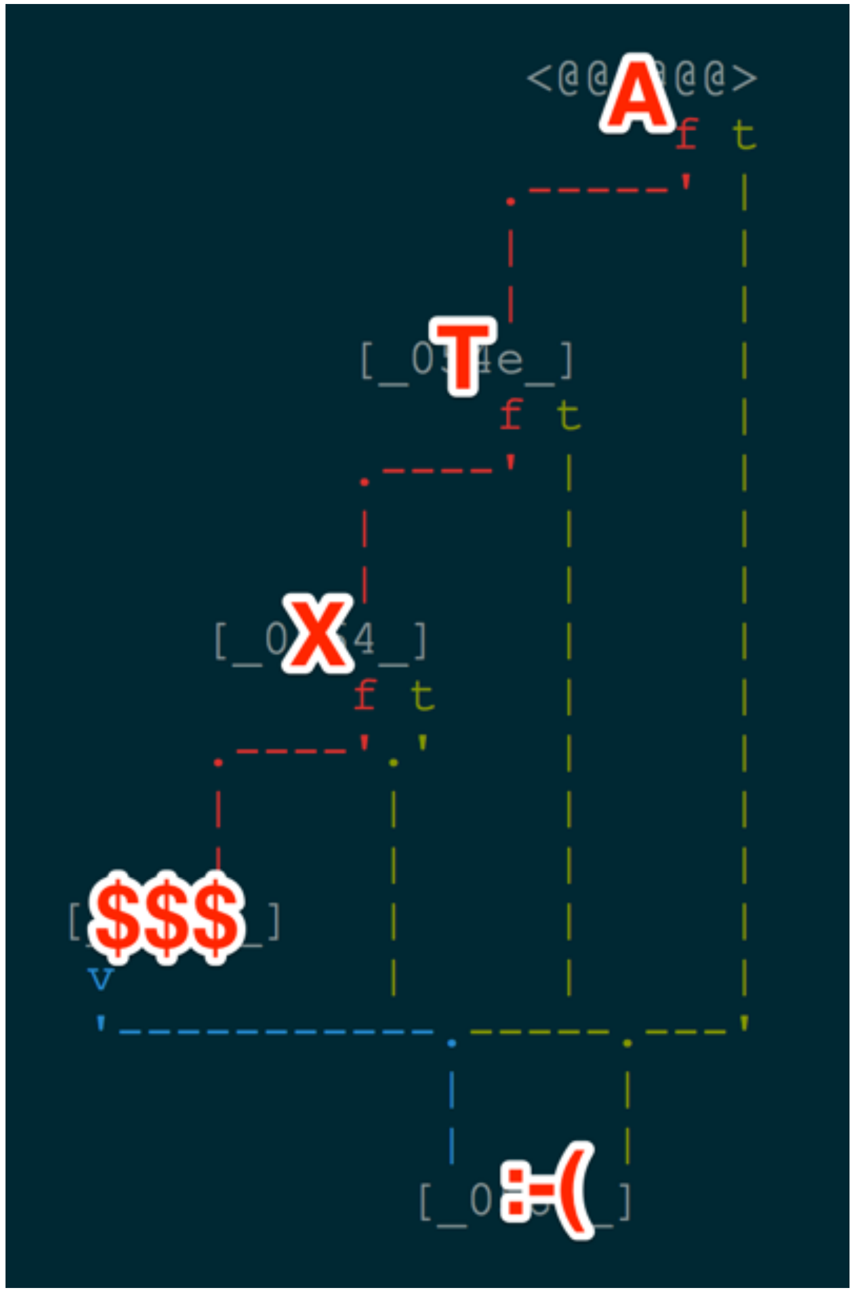
&& input[1] == 't'

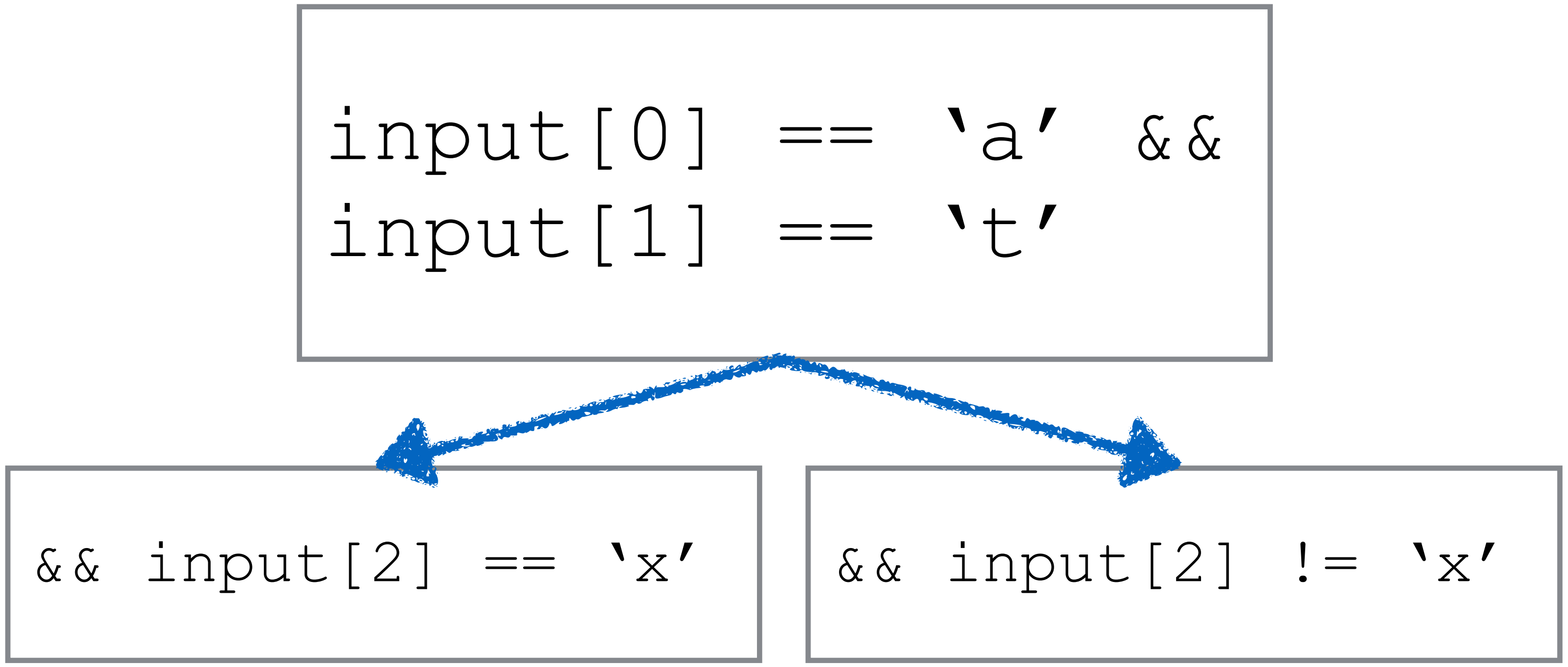
&& input[1] != 't'

continue

exit()

Goal: Easy crackme



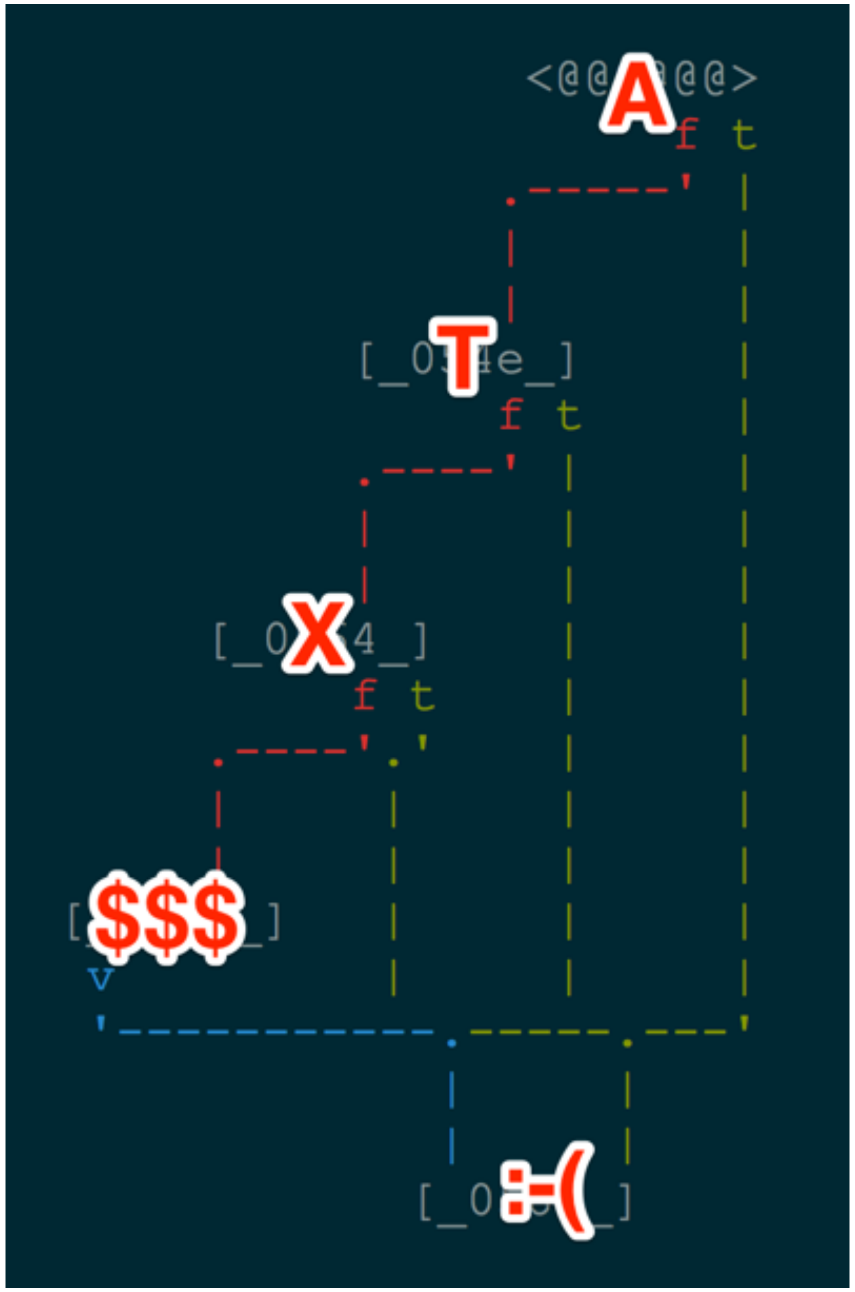


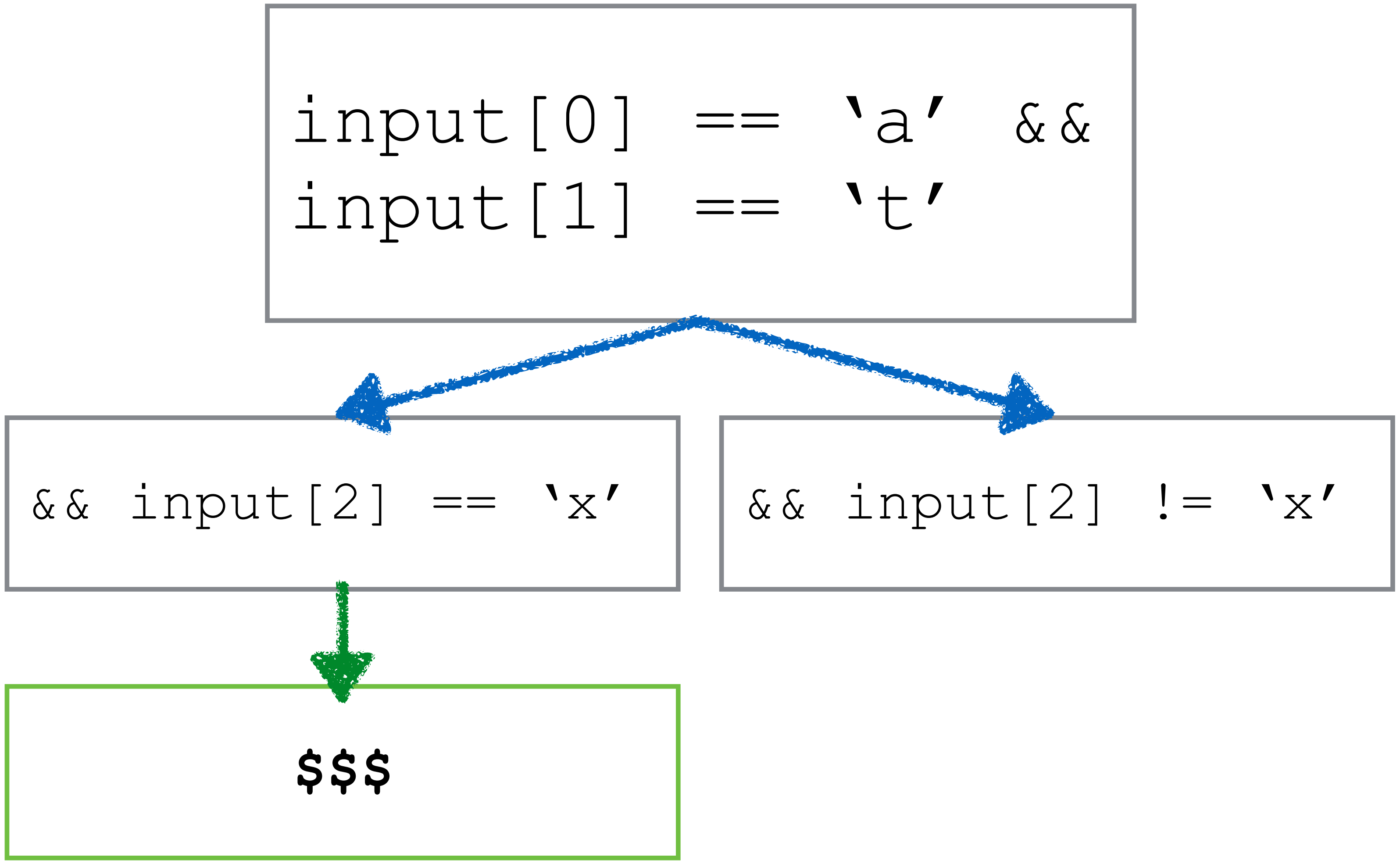
```
input[0] == ‘a’ &&
input[1] == ‘t’
```

```
&& input[2] == ‘x’
```

```
&& input[2] != ‘x’
```

Goal: Easy crackme





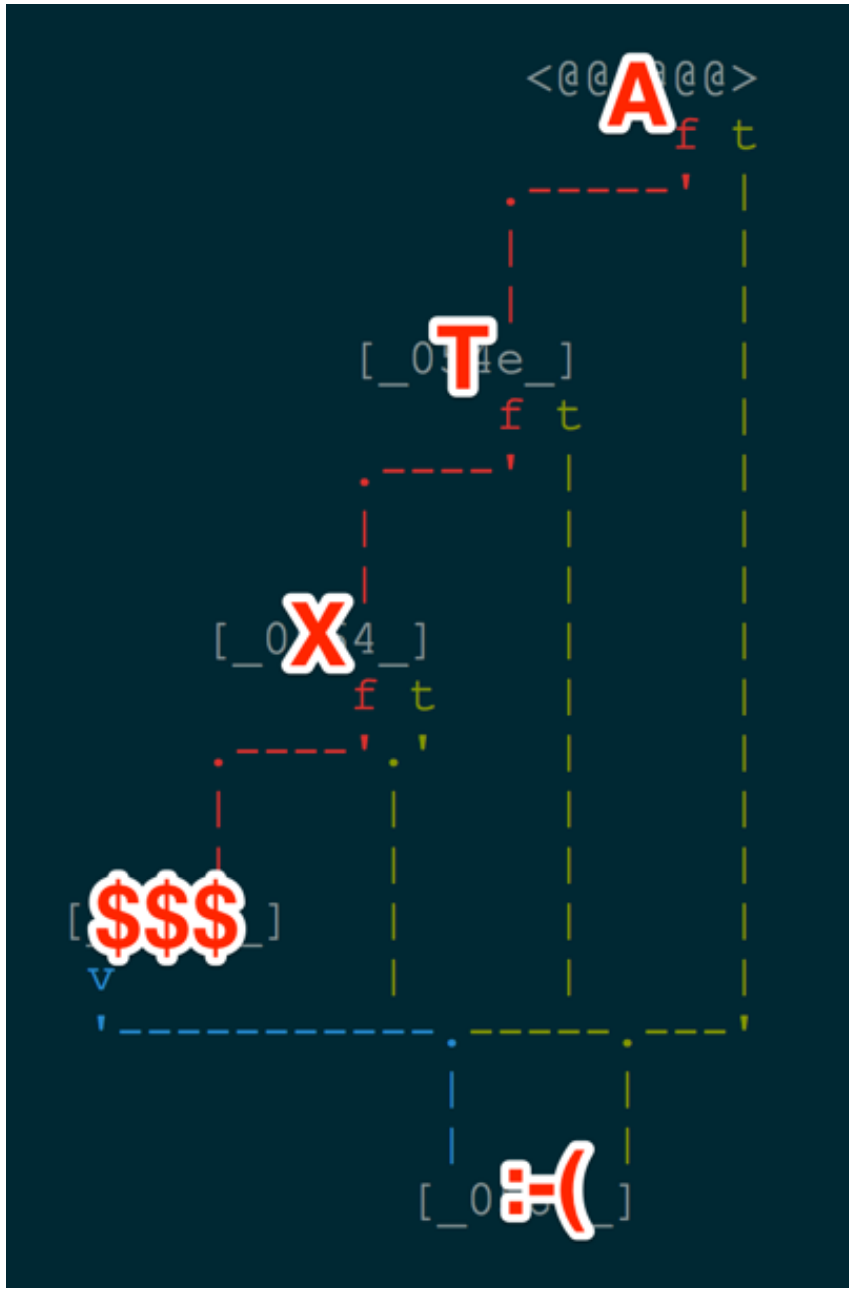
```
input[0] == 'a' &&
input[1] == 't'
```

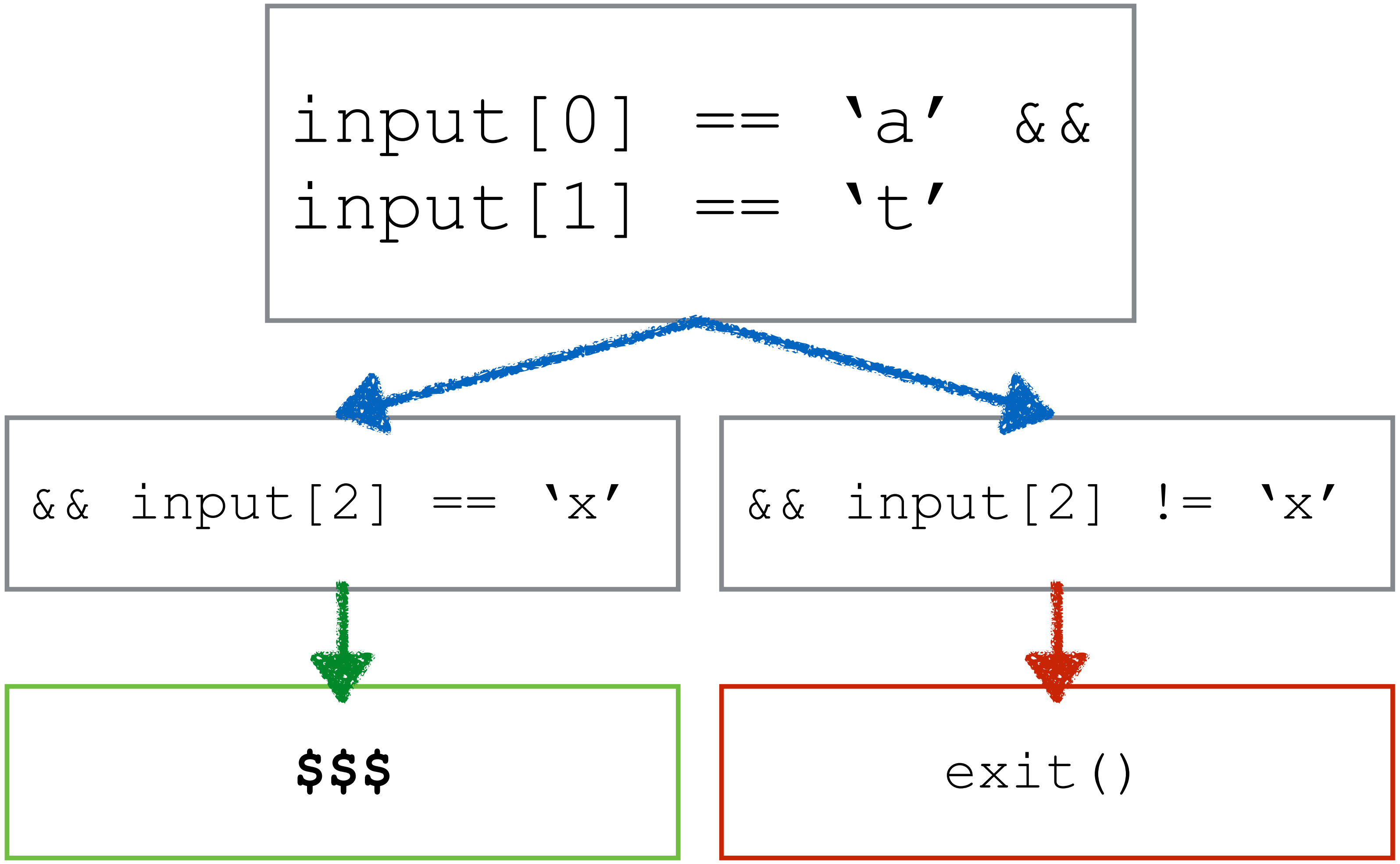
```
&& input[2] == 'x'
```

```
&& input[2] != 'x'
```

$$$

Goal: Easy crackme





```
input[0] == 'a' &&
input[1] == 't'
```

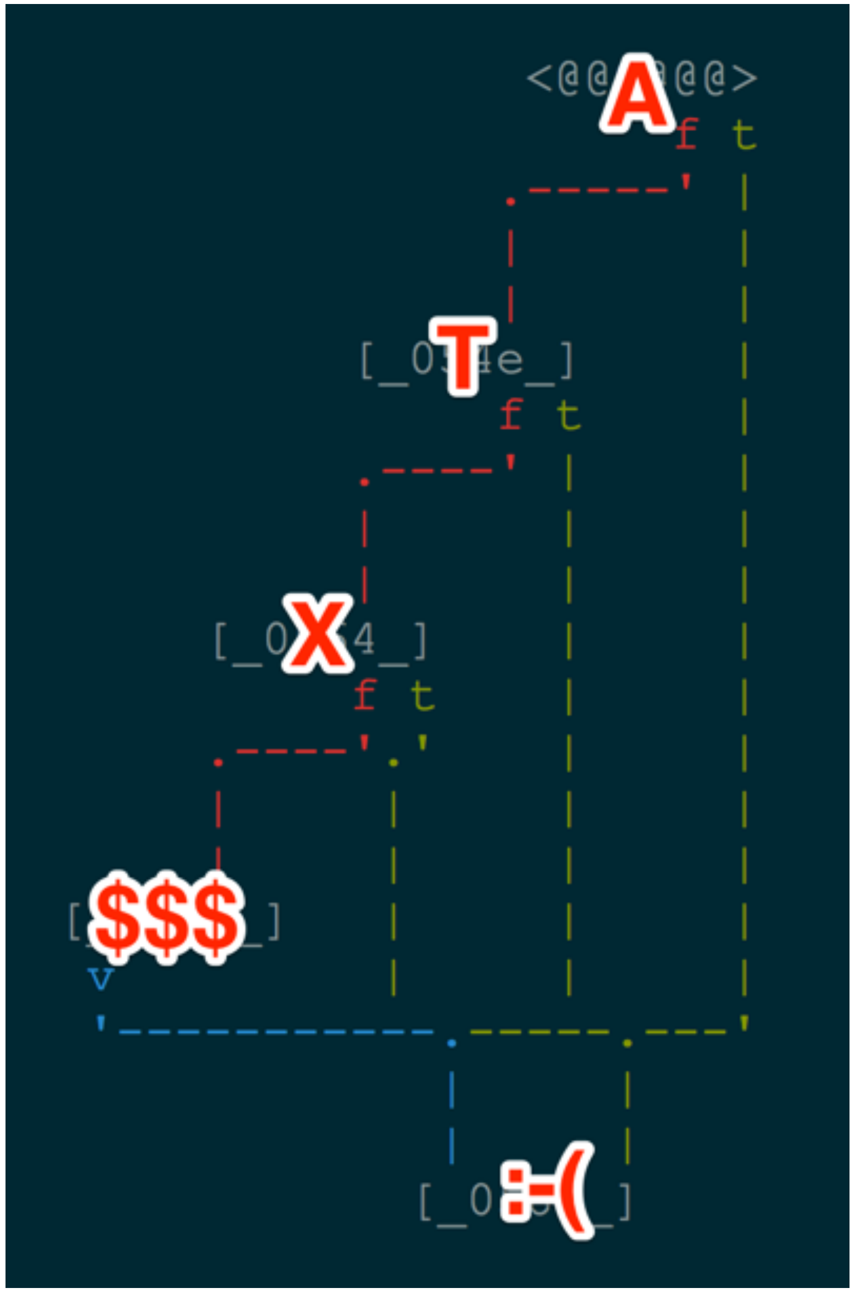
```
&& input[2] == 'x'
```

```
&& input[2] != 'x'
```

$$$

```
exit()
```

Goal: Easy crackme



```
input[0] == 'a' &&
input[1] == 't' &&
input[2] == 'x'
```

```
print("Yay BSIDES")
```

Simply solve the found constraints to extract the answer

Practical exercise 1

- We will be using angr as our symbolic engine of choice
- Developed by researchers from UC Santa Barbara
- Shellphish CTF team
- DARPA Cyber Grand Challenge finalists
- Powered part of their Cyber Reasoning System
- Python ;-)

SOURCE: HTTPS://GITHUB.COM/ANGR

Practical exercise 1

- Quick angr terminology:
  - project - executable (or binary blob) currently being analyzed
  - state - an emulated machine state
  - path - series of basic blocks representing the current execution flow
  - path-group - collection of paths with an easy interface with explore all paths at once
- Input sources
  - A common problem with symbolic execution is telling the engine what to consider symbolic for analysis
  - Common locations for input: stdin and argv

Practical exercise 1

- Basic reverse engineering before starting angr:
  - Included in EpicTreasure is radare2, a command line disassembler
  - Can use IDA Pro, Hopper, or even ObjDump for this portion as well
  - Find the address of the basic block housing the printf to guide angr to the solution we want

HTTPS://AVATARS3.GITHUBUSERCONTENT.COM/U/917142?V=3&S=400

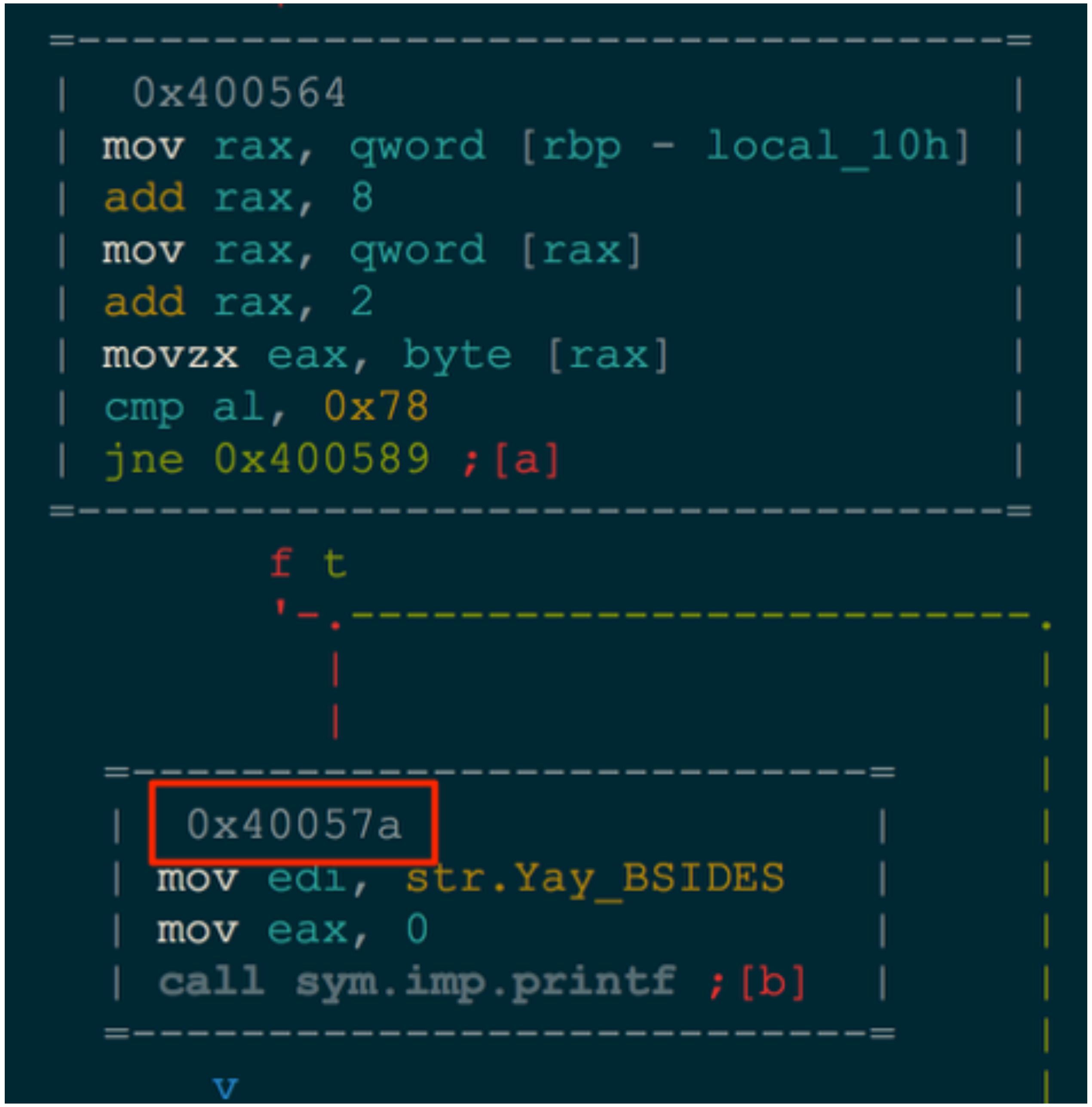
Practical exercise 1

- Quick guide for navigating around radare2:
  - `r2 binary - Start the binary in radare2`
  - `aaa - Analyze all the things in the binary`
  - `afl - List functions`
  - `s main - Seek to an address for analysis (i.e. main)`
  - `VV - Visual mode`
    - `‘hjkl’ or Arrow keys for movement`
    - `q to quit`
  - `? appended to any commands for lots of usage`
    - `?, a?, af?`

Practical exercise 1

Locate the basic block address with the printf

Practical exercise 1



```
=------------------------------------=
|  0x400564                            |
| mov rax, qword [rbp - local_10h]     |
| add rax, 8                           |
| mov rax, qword [rax]                 |
| add rax, 2                           |
| movzx eax, byte [rax]                |
| cmp al, 0x78                         |
| jne 0x400589 ;[a]                    |
=------------------------------------=
        f t
        '_.-------------------------.
          |                         |
          |                         |
  =------------------------------=  |
  |  0x40057a                      |  |
  | mov edi, str.Yay_BSIDES        |  |
  | mov eax, 0                     |  |
  | call sym.imp.printf ;[b]       |  |
  =------------------------------=  |
      v                             |
```

Practical exercise 1

Wanted address: 0x40057a

Practical exercise 1

- Begin by loading the binary into angr

```
import angr
proj = angr.Project(‘demo1’)
```

- Create the symbolic input for argv using the provided solver engine

```
import claripy
input = state.se.BVS(‘input’, 3 * 8)
```

- The BVS is a symbolic bit vector.
  Each BVS needs an id (“input”) and a length in bits (3 bytes or 3 * 8).

- We want to tell angr to set the current state to the entry point with our argv

```
state = proj.factory.entry_state(args=[‘demo1’, input])
```

Practical exercise 1

- Create a path group from the entry state

```
pg = proj.factory.path_group(state)
```

- Tell angr to continue searching paths until we reach the target block

```
pg.explore(find=0x40057a)
```

- At this point, angr has found a path to reach the given destination block

- Extract the found state

```
state = pg.found[0].state
```

Practical exercise 1

- We can look at the constraints angr found to reach the destination

```
print(state.simplify())
```

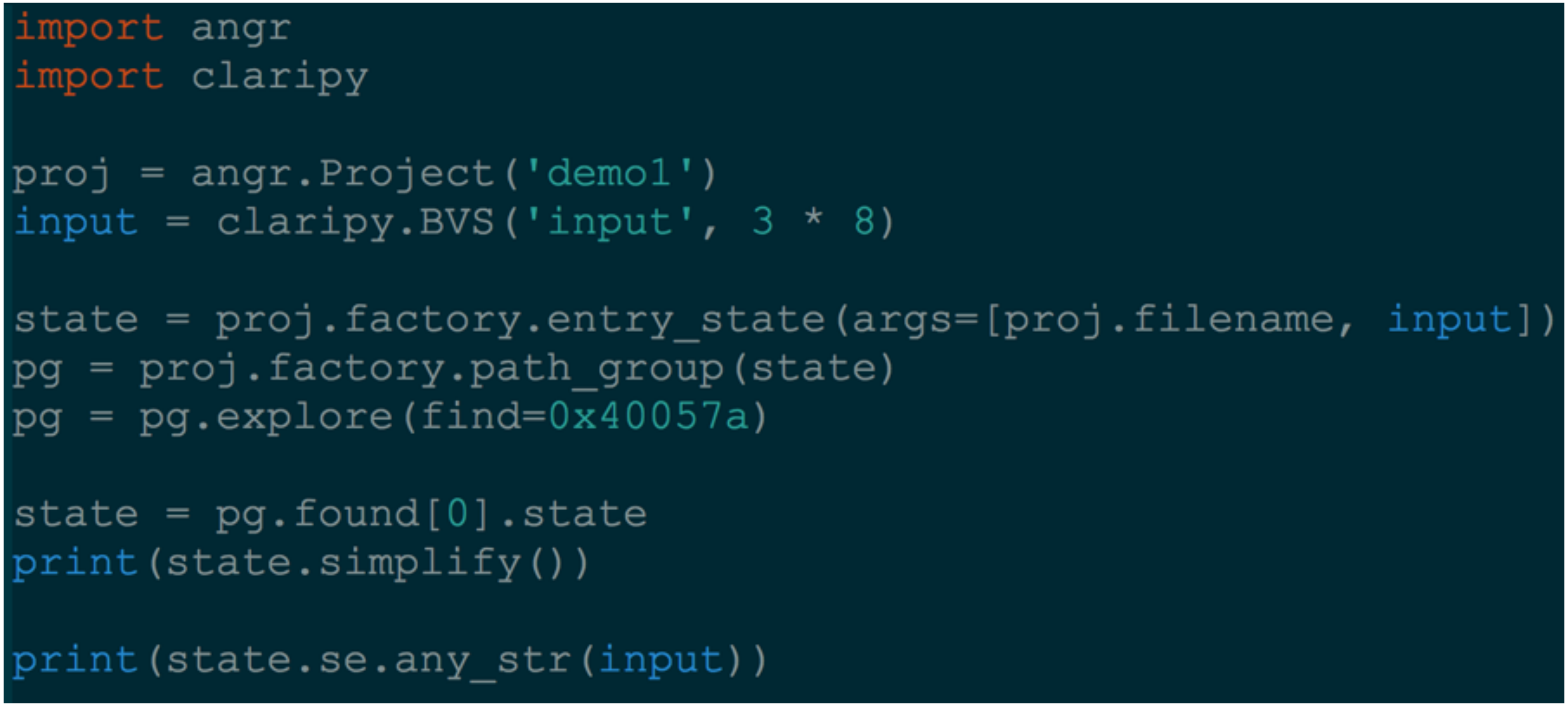
```
[<Bool And((input_8_24[23:16] == 97),
           (input_8_24[15:8] == 116),
           (input_8_24[7:0] == 120))>]
```

- Finally, ask angr to solve these constraints for our answer

```
print(state.se.any_str(input))
```

```
atx
```

Practical exercise 1



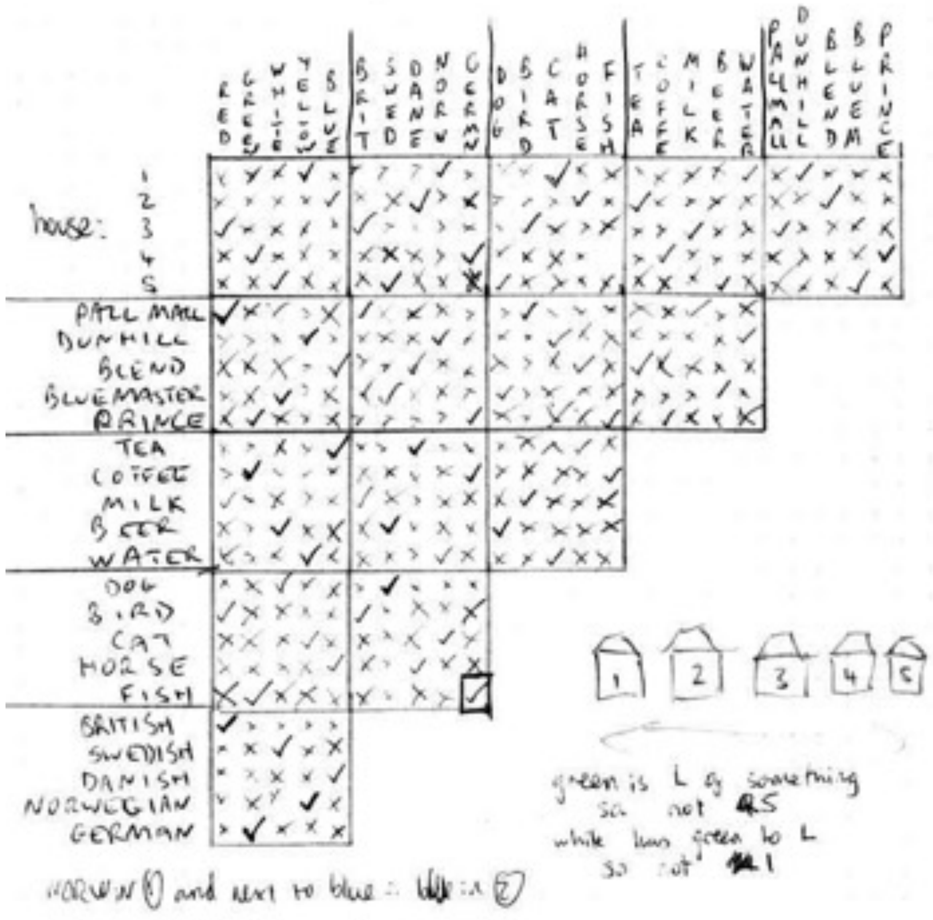
```
import angr
import claripy

proj = angr.Project('demo1')
input = claripy.BVS('input', 3 * 8)

state = proj.factory.entry_state(args=[proj.filename, input])
pg = proj.factory.path_group(state)
pg = pg.explore(find=0x40057a)

state = pg.found[0].state
print(state.simplify())

print(state.se.any_str(input))
```

Practical exercise 1

Make Exercise1 print “You won!”

Exercise 2: Einstein Riddle

- There are 5 houses in five different colors...
- The Brit lives in the red house... etc.
- Lots of boolean logic, prime target for symbolic execution (z3)



green is L of something
so not 5
white has green to L
so not 1
NORW in 1 and next to blue ∴ blue in 2

Exercise 2: Einstein Riddle

- `puzzle` program reads binary data from stdin
- Assigns the data to colors, drinks, nations, etc.
- Matches against Einstein Riddle rules

  ```
  - $ printf "%025d" | ./puzzle
    48:Blue
    48:Green
    48:Ivory
    48:Red
    48:Yellow
    ...
    womp womp... try again
  ```

Exercise 2: Einstein Riddle Input

- Byte values 1-5 represents house number of item
  - e.g. Milk in middle house represented by 3
  - e.g. Norwegian in first house represented by 1
- Specify items in order (found inside binary, but not important)
  -
    ```
    $ echo -ne "\x01\x02\x03\x04\x05" | ./puzzle
    1:Blue
    2:Green
    3:Ivory
    4:Red
    5:Yellow
    ...
    womp womp... try again
    ```

Exercise 2: Einstein Riddle Constraints

Libraries can be nasty, we can disable them and use a simulated libc

```
opts = {'auto_load_libs': False}
proj = angr.Project('puzzle', load_options=opts)
st   = proj.factory.entry_state(args=['puzzle'])

for _ in range(25):
    # works like reading from a fd, moves seek head
    e = st.posix.files[0].read_from(1)
```

Add constraints to SAT problem

```
    st.add_constraints(e >= 1)
    st.add_constraints(e <= 5)

e = st.posix.files[0].read_from(1)
st.add_constraints(e == 0)
```

Set the length of stdin

```
st.posix.files[0].seek(0)
st.posix.files[0].length = 25
```

Exercise 2: Einstein Riddle

Make puzzle print “congratulations!”

Who drinks water? Who owns the zebra?

Exercise 3

Can we only solve reverse engineering problems?

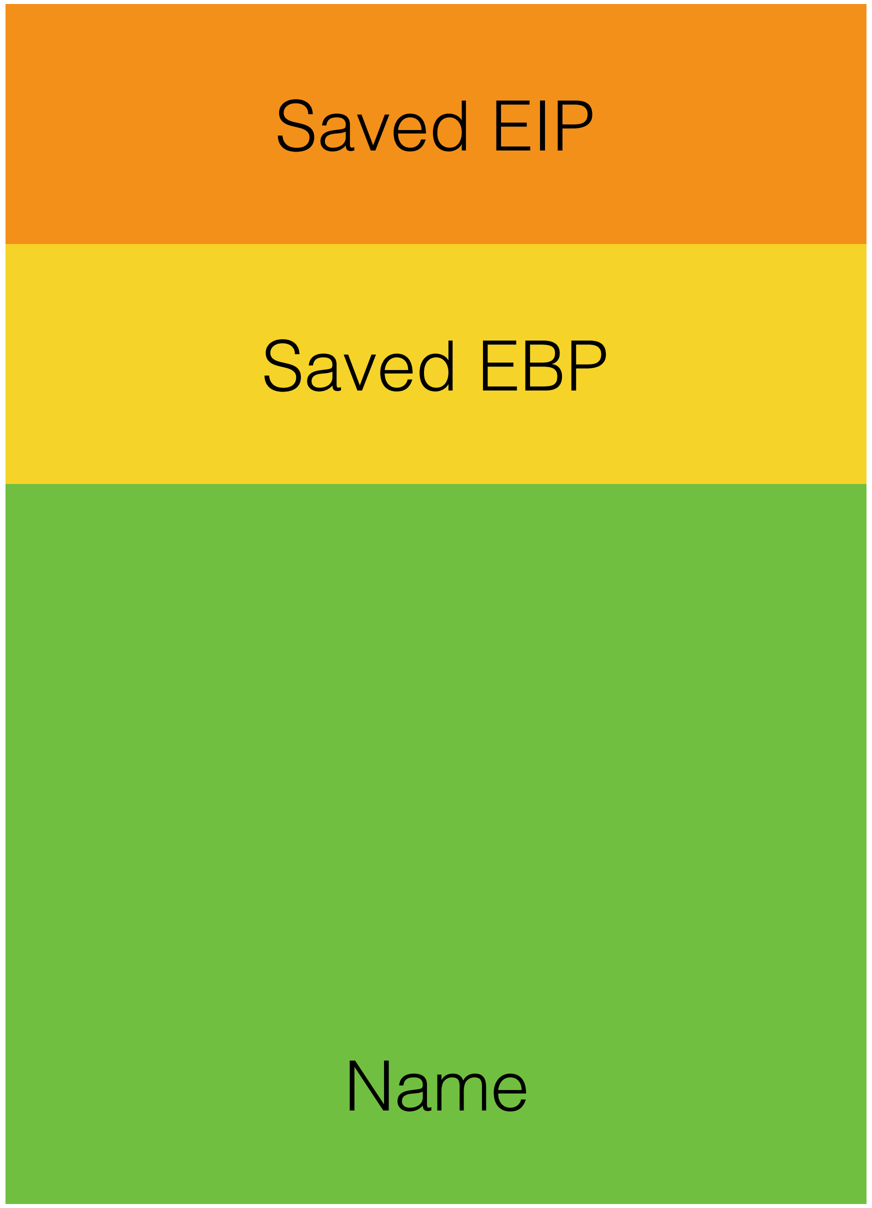
Exercise 3: Stack overflow

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

void overflow_me(){
    char name[24];
    printf("Welcome.. what is your name?\n");
    read(0, name, 80);
    return;
}

int main(int argc, char** argv){
    char vuln[32];
    printf("Password protected. Enter password:\n");

    read(0, vuln, 32);
    if(strstr(vuln, "badpassword") == vuln)
        overflow_me();
    else
        printf("Wrong password\n");

    return 0;
}
```

Stack overflow review

```
void overflow_me(){
    char name[24];
    read(0, name, 80);
    return;
}
```

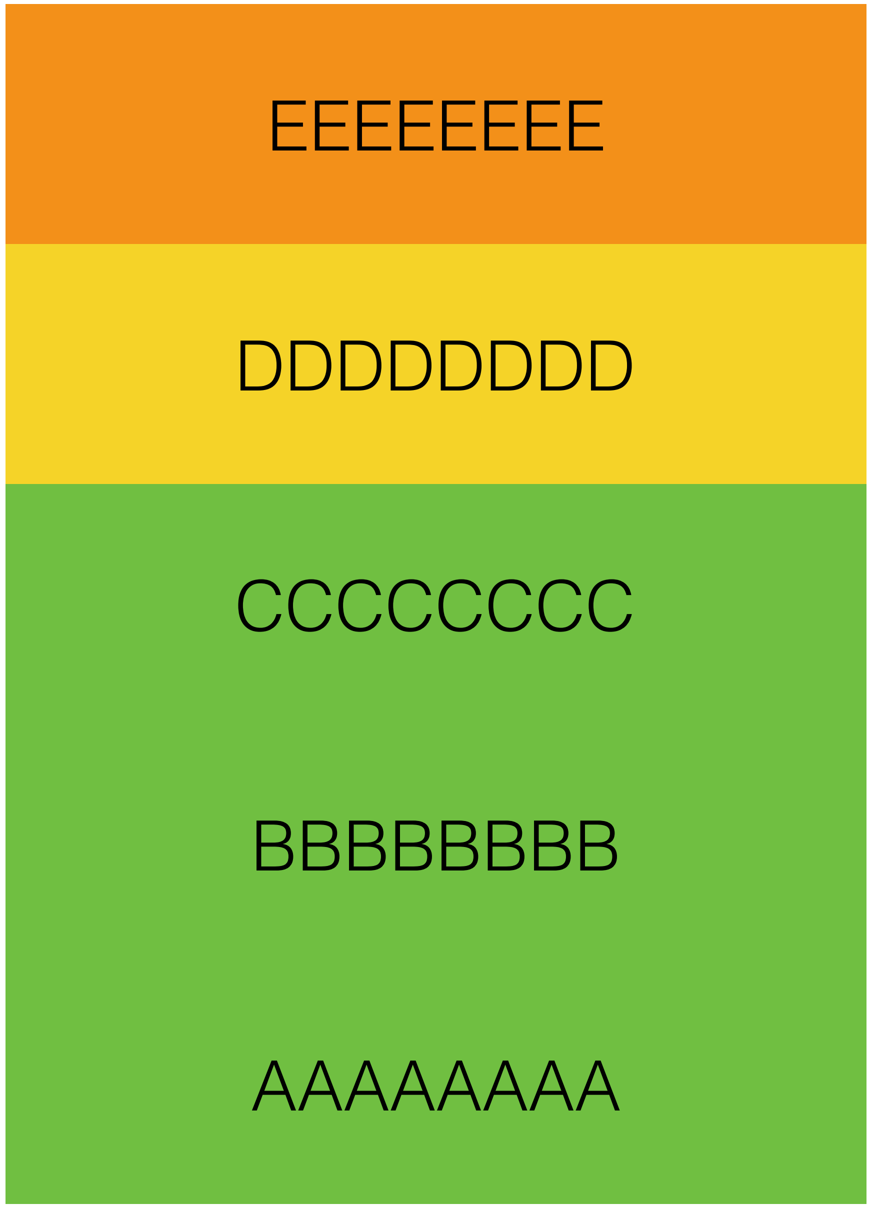
Begin with the stack frame from the vulnerable function.



Stack overflow review

```
void overflow_me(){
    char name[24];
    read(0, name, 80);
    return;
}
```

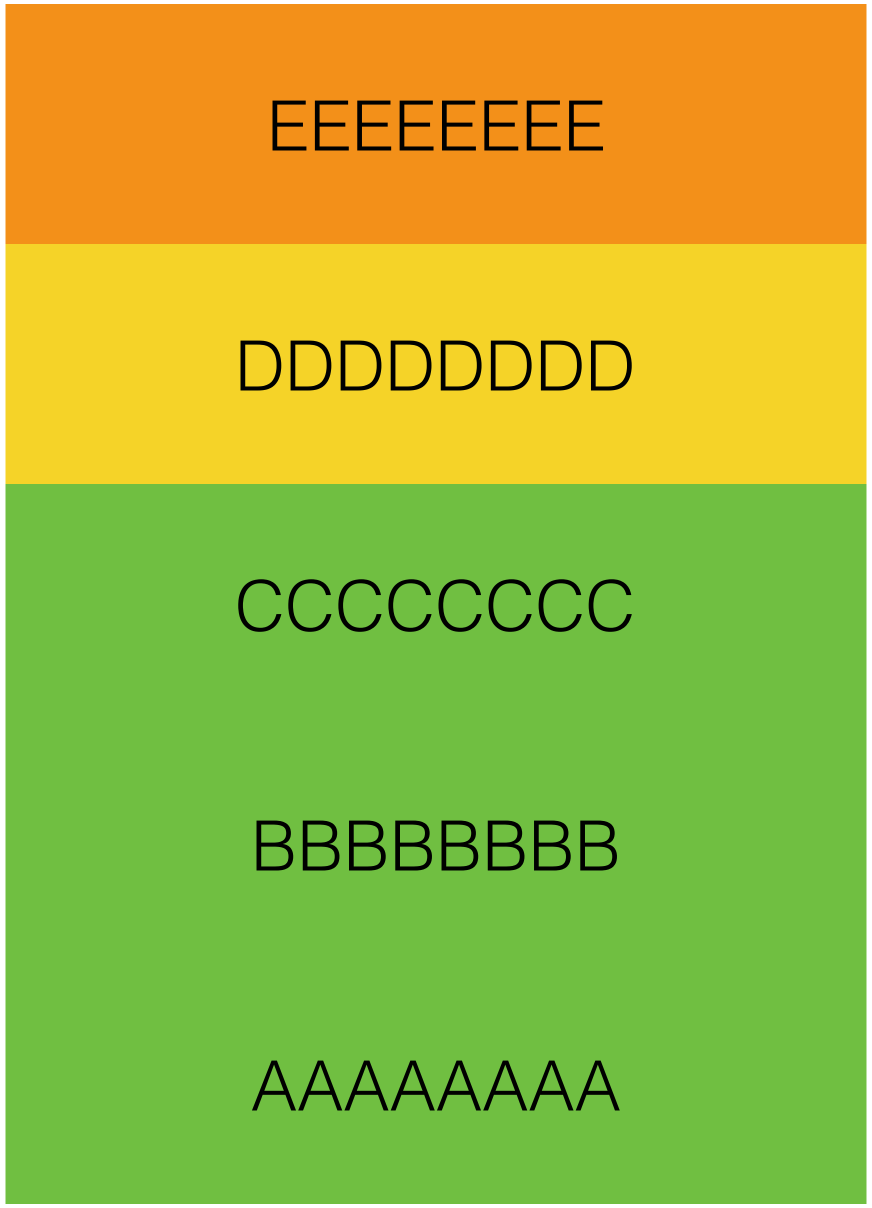
We read beyond the bounds of the char array, overwriting the saved EIP.



Stack overflow review

```
void overflow_me(){
    char name[24];
    read(0, name, 80);
    return;
}
```

On return, we **SEGFAULT** because our EIP is now at an invalid address.



EEEEEEEE

DDDDDDDD

CCCCCCCC

BBBBBBBB

AAAAAAAA

Exercise 3: Stack overflow

- By default, angr discards any unconstrained paths. In this example, we are exactly looking for unconstrained paths:
  - `pg = proj.factory.path_group(state, save_unconstrained=True)`
- We can explore all paths until we have a constrained path. Instead of `explore`, we will manually `step` each path until the number of unconstrained paths is greater than 0.
  - `pg.step(until=lambda x: len(x.unconstrained) > 1)`
- After each path is stepped forward once, it is checked for unconstrained paths. If we found one, `step` will return and hand us our wanted path.

Exercise 3: Stack overflow

- Grab our path from the unconstrained array.
  - `state = pg.unconstrained[0].state`
- We now can simply set a constraint such that the state’s Instruction Pointer equals whatever we want.
  - `crash_ip = claripy.BVV(int('deadbeefcafebabe', 16), 8 * 8)`
  - `state.se.add(state.regs.ip == crash_ip)`
- Because we added the IP constraint, any solved solution will by definition crash with `IP == deadbeefcafebabe`
  - `payload = state.posix.dumps(0)`

Exercise 3: Stack overflow

Demo3

Exercise 3: Stack overflow

```
$ xxd solution
 0000000: 6261 6470 6173 7377 6f72 6400 0000 0000  badpassword.....
 0000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 0000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 0000040: 0000 0000 0000 0000 beba feca efbe adde  ................
 0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Exercise 3: Stack overflow

Crash Exercise3 with 0xdeadc0dedeadc0de.

What was the necessary password?

Exercise 3: Stack overflow

```
$ xxd solution
 0000000: 7469 6d65 5f32 5f73 6f6c 7665 5f61 6c6c  time_2_solve_all
 0000010: 5f74 6865 5f74 6869 6e67 730a 0000 0000  _the_things.....
 0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 0000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 0000040: 0000 0000 0000 0000 dec0 adde dec0 adde  ................
 0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Further Reading

- Papers
  - http://www.internetsociety.org/sites/default/files/11_1_2.pdf
- angr docs
  - https://github.com/angr/angr-doc
  - Loads of examples and detailed explanation of API
- Cyber Grand Challenge
  - http://www.cybergrandchallenge.com/
  - https://github.com/CyberGrandChallenge/samples
    - cqe-challenges contains lots of juicy exploitable binaries

PRAETORIAN

Symbolic Execution Workshop

HOW NOT TO BE ANGR-Y