Graham Dungan
March 28, 2025
UIN: 332001764

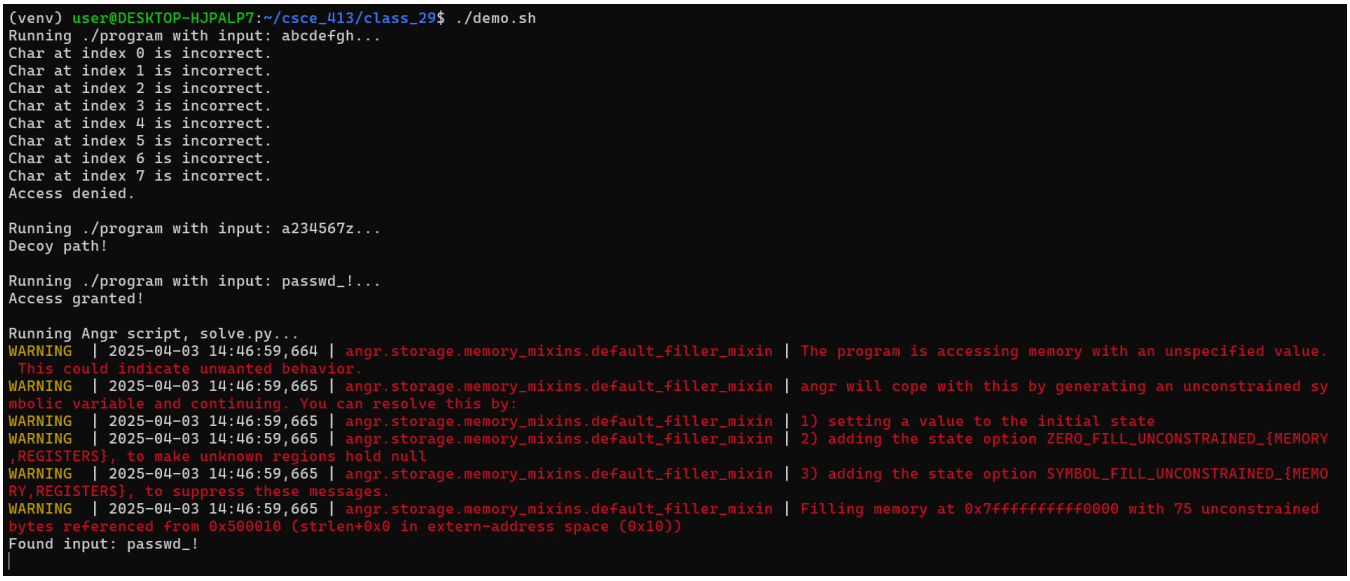
CSCE 413: Software Security
Class 29: Symbolic Execution

Demonstration Quickstart

A demonstration script `demo.sh` has been included. The demonstration script will show the three classes of outputs from `program` and will then run the Angr script `solve.py`.
To run this demo,

1. Enter the `Class29` directory.
   `cd Class29`
2. *(Optional)* If the script does not have execution permission, add such permissions.
   `chmod +x ./demo.sh`
3. If you do not have angr installed on your system, create a Python virtual environment, activate it, and install angr
   ```
   python3 -m venv venv
   source venv/bin/activate
   pip install angr
   ```
4. Run the `demo.sh` script.
   `./demo.sh`

What follows is a screenshot of the output of `demo.sh`.



1 Create an application with hidden paths

For this project, I created a simple application called `program.c`

```c
int main(int argc, char *argv[]) {
    // Useage statement
    if (argc != 2) {
        printf("Usage: %s <8-char-key>\n", argv[0]);
        return 1;
    }

    // Length check
    char *input = argv[1];
    if (strlen(input) != 8) {
        printf("Input must be 8 characters long.\n");
        return 1;
    }

    // XOR each character with a key
    // Shift the value into its respective position in the string
    // passwd_! = 66 77 65 65 61 72 49 37
    char key = 0x16;

    uint32_t part1 =
        ((uint32_t)('p' ^ key)) |
        ((uint32_t)('a' ^ key) << 8) |
        ((uint32_t)('s' ^ key) << 16) |
        ((uint32_t)('s' ^ key) << 24);

    uint32_t part2 =
        ((uint32_t)('w' ^ key)) |
        ((uint32_t)('d' ^ key) << 8) |
        ((uint32_t)('_' ^ key) << 16) |
        ((uint32_t)('!' ^ key) << 24);

    // Copy each one into its respective position into the array
    char obfuscated[8];
    memcpy(obfuscated, &part1, 4);
    memcpy(obfuscated + 4, &part2, 4);

    // Decoy Path, if it starts with an a or a z, call a decoy path and terminate
    if (input[0] == 'a' && input[7] == 'z') {
        decoy_path();
        return 1;
    }

    // If any index check fails, 1 goes to 0
    int valid = 1;

    // Check all indices via a loop
    for (int i = 0; i < 8; i++) {
        valid &= check_char(input[i], obfuscated[i], key, i);
    }

    // Evalute all checks
    if (valid) {
        printf("Access granted!\n");
        return 0;
    } else {
        printf("Access denied.\n");
        return 1;
    }
}
```

Note that, for the sake of brevity, the helper functions for this have been excluded. This program checks whether the input password (supplied as an argument) matches an obfuscated string.

String Obfuscation

The password for this program is "passwd_!". Lines 20-30 obfuscate each character of the correct password by XORing each character with a key, this being the hexadecimal value `0x16`. Once each value is XOR'd, it is bit-shifted relative to its expected position in the string and essentially concatenated.

Lines 37-40 create a fake path for the program by checking for the characters "a" and "z" at the beginning and end of the string. Lines 43-49 call a function to check the user-supplied input against the obfuscated password character by character.

```
int check_char(char input, char expected, char key, int index) {
    char expected_xor = expected ^ key;
    if (input != expected_xor) {
        printf("Char at index %d is incorrect.\n", index);
        return 0;
    }
    return 1;
}
```

The `check_char` function simply recalculates the expected input by applying another XOR with the same key. The validity of the password is then checked, and a string corresponding to its success is printed at the end of `main`. The program can be compiled with the command;

```
gcc program.c -o program
```

And can be run as follows;

```
(venv) user@DESKTOP-HJPALP7:~/csce_413/class_29$ ./program abcdefgh
Char at index 0 is incorrect.
Char at index 1 is incorrect.
Char at index 2 is incorrect.
Char at index 3 is incorrect.
Char at index 4 is incorrect.
Char at index 5 is incorrect.
Char at index 6 is incorrect.
Char at index 7 is incorrect.
Access denied.
(venv) user@DESKTOP-HJPALP7:~/csce_413/class_29$ ./program a234567z
Decoy path!
(venv) user@DESKTOP-HJPALP7:~/csce_413/class_29$ ./program passwd_!
Access granted!
(venv) user@DESKTOP-HJPALP7:~/csce_413/class_29$
```

2 Demonstrate how to find this path/input using Angr

To run Angr we will need to create a Python script named `solve.py`,

```
1  import angr
2  import claripy
3
4  proj = angr.Project("./program", auto_load_libs=False)
5
6  # Create an argument input of 8 bytes
7  arg = claripy.BVS("arg", 8 * 8)
8  argv = [proj.filename, arg]
9  state = proj.factory.full_init_state(args=argv)
10
11 simgr = proj.factory.simgr(state)
12
13 # Search for access granted string
14 simgr.explore(find=lambda s: b"Access granted!" in s.posix.dumps(1))
15
16 if simgr.found:
17     found = simgr.found[0]
18     solution = found.solver.eval(arg, cast_to=bytes)
19     print("Found input:", solution.decode())
20 else:
21     print("No solution found.")
```

This program first creates an Angr project and loads the previously compiled `program` binary on line 4. The program then creates a symbolic input of length 8, variable `arg`, and simulates a command-line argument in the starting state of the program, `state`, on lines 7-9. Following, a simulation manager `simgr` is created on line 11 and is instructed to explore until it finds the string "Access granted!" from standard out on line 14, as evident by the lambda function. Finally, if the "Access granted!" string is found, it prints the input associated with the success, otherwise, it states that no solution was found.
Once the Angr Python script has been created, it can be run with,

```
python3 ./solve.py
```

Note that it must be in the same directory as the `program` binary. The output is as follows,

```
(venv) user@DESKTOP-HJPALP7:~/csce_413/class_29$ python3 ./solve.py
WARNING  | 2025-04-03 14:39:37,291 | angr.storage.memory_mixins.default_filler_mixin | The program is accessing memory with an unspecified value.
 This could indicate unwanted behavior.
WARNING  | 2025-04-03 14:39:37,292 | angr.storage.memory_mixins.default_filler_mixin | angr will cope with this by generating an unconstrained sy
mbolic variable and continuing. You can resolve this by:
WARNING  | 2025-04-03 14:39:37,292 | angr.storage.memory_mixins.default_filler_mixin | 1) setting a value to the initial state
WARNING  | 2025-04-03 14:39:37,292 | angr.storage.memory_mixins.default_filler_mixin | 2) adding the state option ZERO_FILL_UNCONSTRAINED_{MEMORY
,REGISTERS}, to make unknown regions hold null
WARNING  | 2025-04-03 14:39:37,292 | angr.storage.memory_mixins.default_filler_mixin | 3) adding the state option SYMBOL_FILL_UNCONSTRAINED_{MEMO
RY,REGISTERS}, to suppress these messages.
WARNING  | 2025-04-03 14:39:37,293 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at 0x7fffffffff0000 with 75 unconstrained
bytes referenced from 0x500010 (strlen+0x0 in extern-address space (0x10))
Found input: passwd_!
(venv) user@DESKTOP-HJPALP7:~/csce_413/class_29$
```

It is seen that Angr has successfully resolved the password `passwd_!`.