Lab 5 Report **Abhishek Srivastava**
CS260-001: Computer Security Student Id: 861307778
Prof. Heng Yin February 21, 2017

Symbolic Execution Using Angr

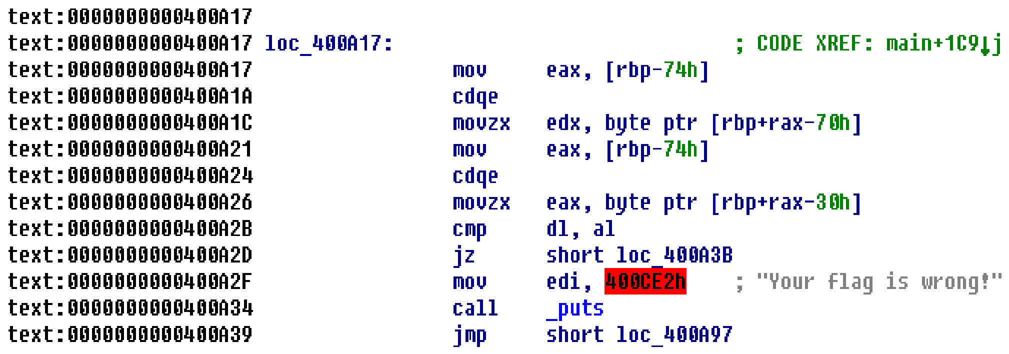
Problem 1

The first binary **re**, when given a correct input, it will print out right. Please write an Angr script to find this correct input.

Solution

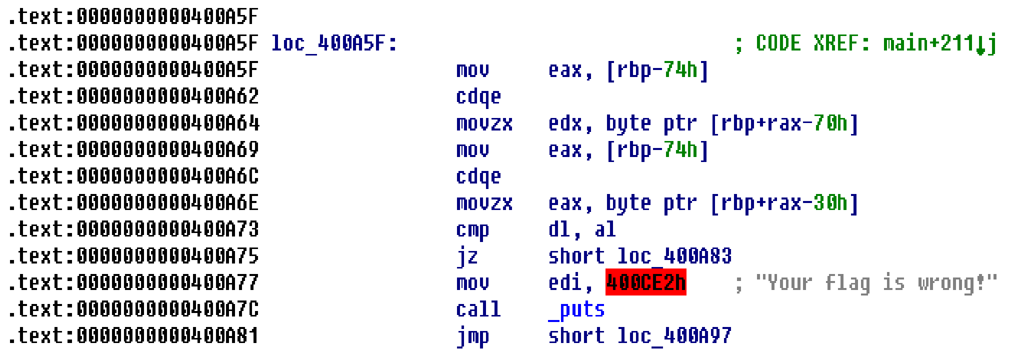
```
#!/usr/bin/env python2

# Author: Abhishek Kumar Srivastava
# Assignment 5 - Problem 1

import angr

#Address of the block we need to traverse
FIND_ADDR  = 0x0000000000400A8D

#Addresses of the blocks we have to avoid
AVOID_ADDR = (0x0000000000400A2F, 0x0000000000400A77)

def re_flag():
    #load the binary into angr project and disable auto loading libraries
    proj = angr.Project('re', load_options={"auto_load_libs": False})

    #entry_state constructor generates a SimState that is a generic
    #representation of the possible program states at the program
    #entry point.
    state = proj.factory.entry_state()

    #To perform symbolic execution we need a Path.
    #Paths wrap states and act as interface for stepping
    #forward and tracking their history.
    path = proj.factory.path(state)

    #Path group are used for symoblic execution process as it is a
    #collection of paths with various tags
    path_group = proj.factory.path_group()

    #find attribute tells which path to be included and avoid parameter
    #describes which paths are to be avoided while execution
    path_group.explore(find=FIND_ADDR, avoid=AVOID_ADDR)

    #This is used to print the information.
    #found state is in pathgroup whose information is being dumped as output
    return path_group.found[0].state.posix.dumps(0)

if __name__ == '__main__':
    print(re_flag())
```

Above program is used for capturing the flag. Figure 1,2 and 3 shows the screenshot of the disassembled data of the interested section of the programs. In the program you can see that we are trying to find data in a certain path and avoid other paths. I am trying the path where flag captured success indication is given. **loc_400A87** seems to be good position to check but when checked there only half of the flag is captured may be because of the compare and jump to other location so I used the next address to check **loc_400A8D**. I am avoiding the sections which give the wrong flag captured indications (**loc_400A2F, loc_400A77**).



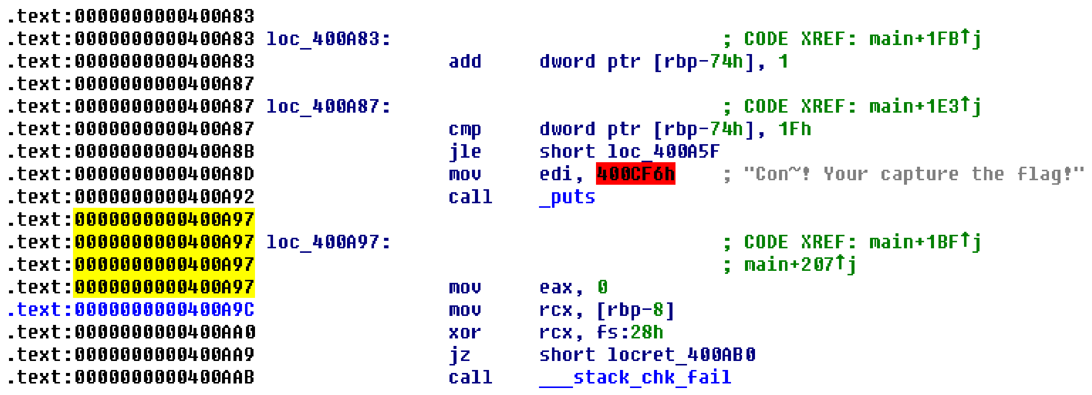
```
text:0000000000400A17
text:0000000000400A17 loc_400A17:                             ; CODE XREF: main+1C9↓j
text:0000000000400A17                 mov     eax, [rbp-74h]
text:0000000000400A1A                 cdqe
text:0000000000400A1C                 movzx   edx, byte ptr [rbp+rax-70h]
text:0000000000400A21                 mov     eax, [rbp-74h]
text:0000000000400A24                 cdqe
text:0000000000400A26                 movzx   eax, byte ptr [rbp+rax-30h]
text:0000000000400A2B                 cmp     dl, al
text:0000000000400A2D                 jz      short loc_400A3B
text:0000000000400A2F                 mov     edi, 400CE2h    ; "Your flag is wrong!"
text:0000000000400A34                 call    _puts
text:0000000000400A39                 jmp     short loc_400A97
```

Figure 1: Output Screen Shot of disassembled path for wrong flag indication.



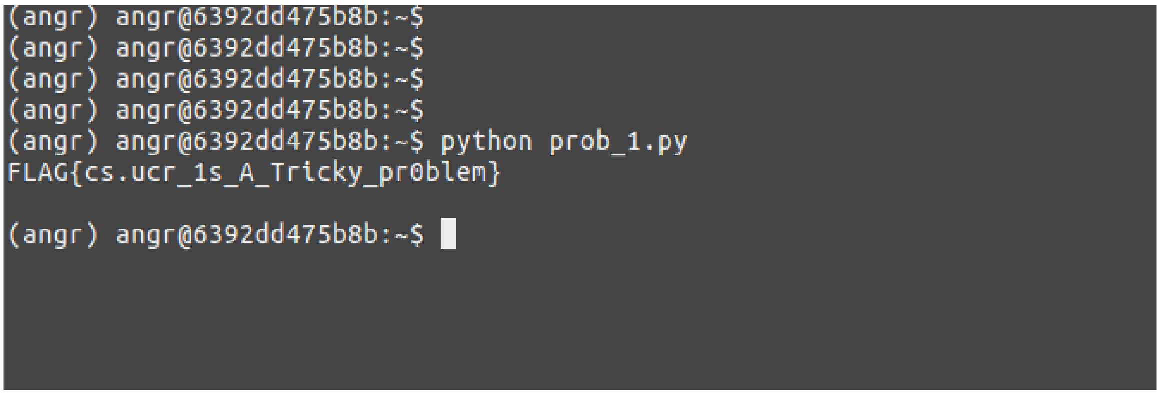
```
.text:0000000000400A5F
.text:0000000000400A5F loc_400A5F:                             ; CODE XREF: main+211↓j
.text:0000000000400A5F                 mov     eax, [rbp-74h]
.text:0000000000400A62                 cdqe
.text:0000000000400A64                 movzx   edx, byte ptr [rbp+rax-70h]
.text:0000000000400A69                 mov     eax, [rbp-74h]
.text:0000000000400A6C                 cdqe
.text:0000000000400A6E                 movzx   eax, byte ptr [rbp+rax-30h]
.text:0000000000400A73                 cmp     dl, al
.text:0000000000400A75                 jz      short loc_400A83
.text:0000000000400A77                 mov     edi, 400CE2h    ; "Your flag is wrong!"
.text:0000000000400A7C                 call    _puts
.text:0000000000400A81                 jmp     short loc_400A97
```

Figure 2: Output Screen Shot of disassembled path for worng flag indication.



```
.text:0000000000400A83
.text:0000000000400A83 loc_400A83:                             ; CODE XREF: main+1FB↑j
.text:0000000000400A83                 add     dword ptr [rbp-74h], 1
.text:0000000000400A87
.text:0000000000400A87 loc_400A87:                             ; CODE XREF: main+1E3↑j
.text:0000000000400A87                 cmp     dword ptr [rbp-74h], 1Fh
.text:0000000000400A8B                 jle     short loc_400A5F
.text:0000000000400A8D                 mov     edi, 400CF6h    ; "Con~! Your capture the flag!"
.text:0000000000400A92                 call    _puts
.text:0000000000400A97
.text:0000000000400A97 loc_400A97:                             ; CODE XREF: main+1BF↑j
.text:0000000000400A97                                         ; main+207↑j
.text:0000000000400A97                 mov     eax, 0
.text:0000000000400A9C                 mov     rcx, [rbp-8]
.text:0000000000400AA0                 xor     rcx, fs:28h
.text:0000000000400AA9                 jz      short locret_400AB0
.text:0000000000400AAB                 call    ___stack_chk_fail
```

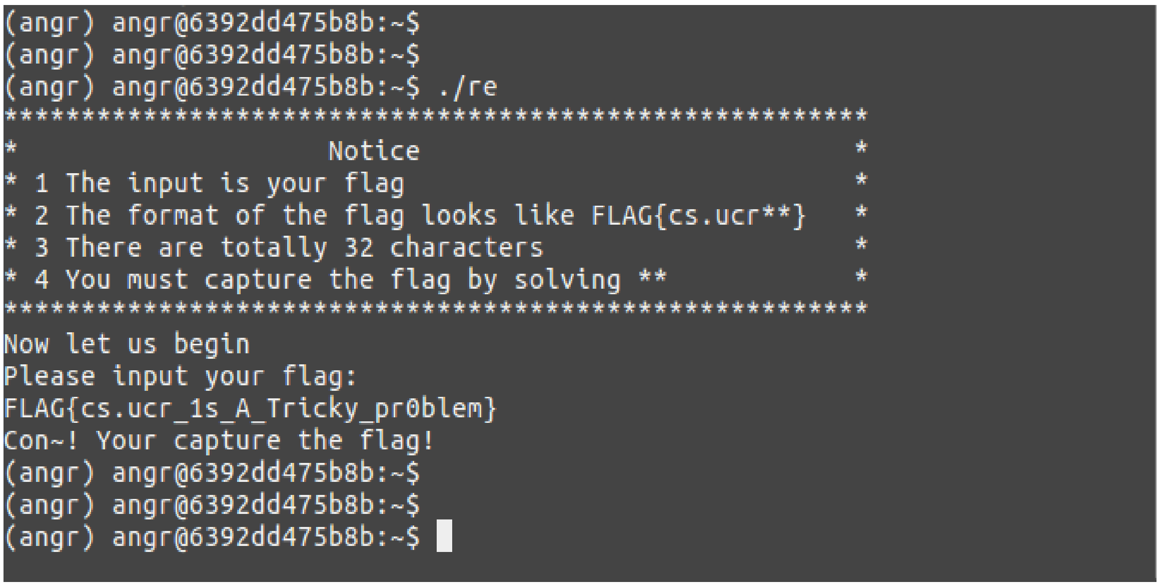
Figure 3: Output Screen Shot of disassembled path for correct flag captured indication.



```
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$ python prob_1.py
FLAG{cs.ucr_1s_A_Tricky_pr0blem}

(angr) angr@6392dd475b8b:~$
```

Figure 4: Output Screen Shot of flag captured for re.

Figure 4 shows the execution of the program written above on the binary provided. From this execution we captured the flag which was **FLAG{cs.ucr_1s_A_Tricky_pr0blem}**. To check the correctness of the flag captured I ran the program and gave the input for which I got the flag captured output from the program. Figure 5 shows the output of the execution of testing flag captured.



```
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$ ./re
*********************************************************
*                    Notice                             *
* 1 The input is your flag                              *
* 2 The format of the flag looks like FLAG{cs.ucr**}    *
* 3 There are totally 32 characters                     *
* 4 You must capture the flag by solving **             *
*********************************************************
Now let us begin
Please input your flag:
FLAG{cs.ucr_1s_A_Tricky_pr0blem}
Con~! Your capture the flag!
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
```

Figure 5: Output Screen Shot of execution when flag is entered.

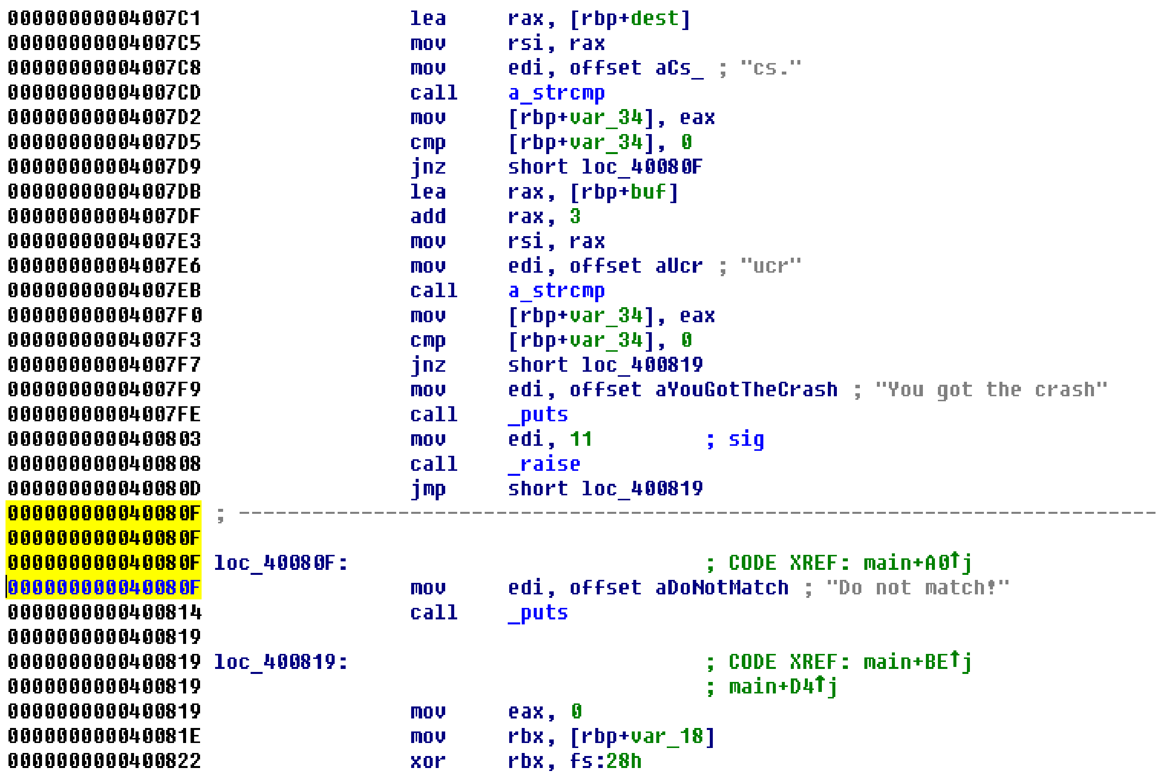
Problem 2

The second binary **afl_strcmp** has a vulnerability. When given a right input, it will crash. Please write an Angr script to trigger the crash.

Solution

```
#!/usr/bin/env python2

# Author: Abhishek Kumar Srivastava
# Assignment 5 - Problem 2

import angr
#Address of the block we need to traverse
FIND_ADDR  = 0x00000000004007F9
#Addresses of the blocks we have to avoid
AVOID_ADDR = 0x000000000040080F

def main():
    #load the binary into angr project and disable auto loading libraries
    proj = angr.Project('afl_strcmp',
                        load_options={"auto_load_libs": False})

    #entry_state constructor generates a SimState that is a generic
    #representation of the possible program states at the program
    #entry point.
    state = proj.factory.entry_state()

    #To perform symbolic execution we need a Path.
    #Paths wrap states and act as interface for stepping
    #forward and tracking their history.
    path = proj.factory.path(state)
    #Path group are used for symoblic execution process as it is a
    #collection of paths with various tags
    path_group = proj.factory.path_group()

    #find attribute tells which path to be included and avoid parameter
    #describes which paths are to be avoided while execution
    path_group.explore(find=FIND_ADDR, avoid=AVOID_ADDR)

    #This is used to print the information.
    #found state is in pathgroup whose information is being dumped as output
    return path_group.found[0].state.posix.dumps(0)

if __name__ == '__main__':
    print(main())
```

Above program follow the same path as the previous problem we have to identify the correct path in the program to execute angr correctly.

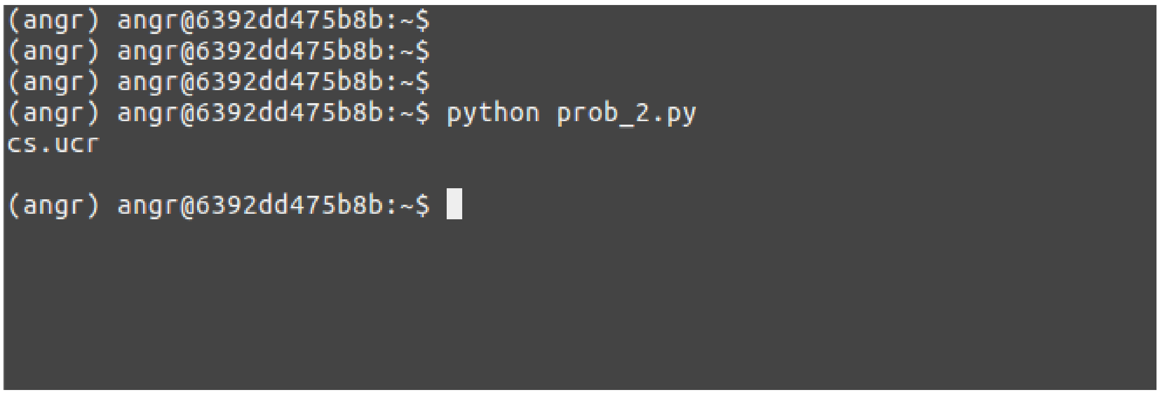


```
00000000004007C1                 lea     rax, [rbp+dest]
00000000004007C5                 mov     rsi, rax
00000000004007C8                 mov     edi, offset aCs_ ; "cs."
00000000004007CD                 call    a_strcmp
00000000004007D2                 mov     [rbp+var_34], eax
00000000004007D5                 cmp     [rbp+var_34], 0
00000000004007D9                 jnz     short loc_40080F
00000000004007DB                 lea     rax, [rbp+buf]
00000000004007DF                 add     rax, 3
00000000004007E3                 mov     rsi, rax
00000000004007E6                 mov     edi, offset aUcr ; "ucr"
00000000004007EB                 call    a_strcmp
00000000004007F0                 mov     [rbp+var_34], eax
00000000004007F3                 cmp     [rbp+var_34], 0
00000000004007F7                 jnz     short loc_400819
00000000004007F9                 mov     edi, offset aYouGotTheCrash ; "You got the crash"
00000000004007FE                 call    _puts
0000000000400803                 mov     edi, 11         ; sig
0000000000400808                 call    _raise
000000000040080D                 jmp     short loc_400819
000000000040080F ; ---------------------------------------------------------------------------
000000000040080F
000000000040080F loc_40080F:                             ; CODE XREF: main+A0↑j
000000000040080F                 mov     edi, offset aDoNotMatch ; "Do not match!"
0000000000400814                 call    _puts
0000000000400819
0000000000400819 loc_400819:                             ; CODE XREF: main+BE↑j
0000000000400819                                         ; main+D4↑j
0000000000400819                 mov     eax, 0
000000000040081E                 mov     rbx, [rbp+var_18]
0000000000400822                 xor     rbx, fs:28h
```

Figure 6: Disassembled output from IDA of the binary afl_strcmp.

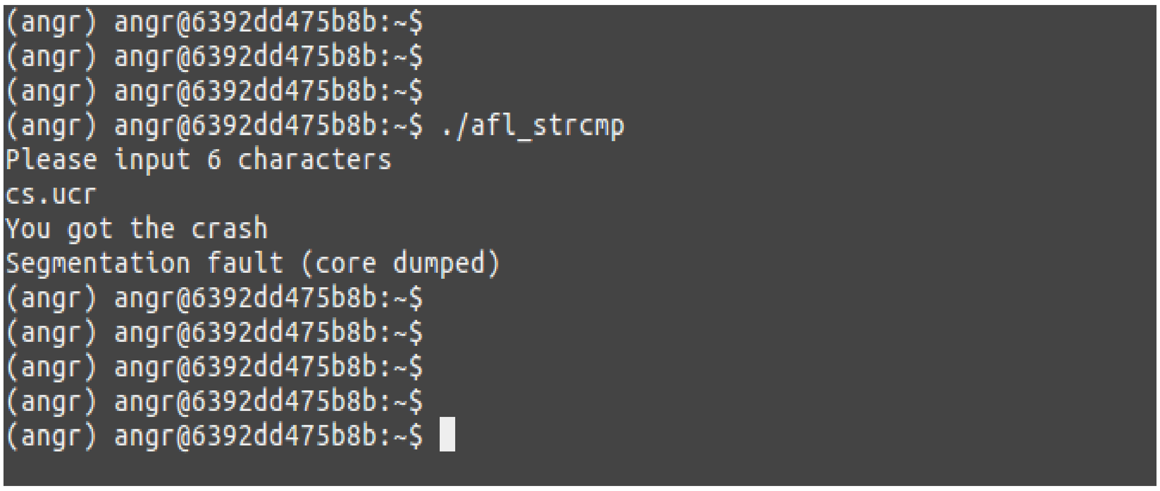
Figure 6 shows the disassembled output of the binary provided **afl_strcmp**. Ideally we should check the path after the both string compare has been done which is **loc_4007F0** but there are compare and jump methods are called so I tested **loc_4007F9**. As usual I avoided the path which gives the indication of input does not match which is **loc_40080F**.

Figure 7 shows the execution of the program written above. On execution we get the string whose input in the program can cause segmentation fault. To test the output captured is correct or not I ran the program and inputed the string which caused the program to give the segmentation fault indication. Figure 8 shows the execution done.



```
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$ python prob_2.py
cs.ucr

(angr) angr@6392dd475b8b:~$
```

Figure 7: Output Screen Shot of input for program afl_strcmp.



```
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$ ./afl_strcmp
Please input 6 characters
cs.ucr
You got the crash
Segmentation fault (core dumped)
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
(angr) angr@6392dd475b8b:~$
```

Figure 8: Output Screen Shot execution of crashing program with input captured.