2018 NCL Gym – Enumeration And Exploitation:

1) PYTHON1.PY Answer: Password => eSffffffff

What you know from the code:

- Password length = 10
- Second character in the password is ASCII numeric value 83 or 'S'

How to:

- Use unicode-table.com or any ASCII table to get the ASCII numeric values for the letters and digits.
- Review the Python program and you will see that it gives the second letter of the password as 83 which is the letter 'S', the length of the password is 10 letters or digits long, and the password ASCII numeric total is 1000.
- Take the HTML code total 1000, subtract 83 which leaves a remainder of 917 which needs to spread across 9 letters or digits. At this point, you can play with the Unicode table to get any combination of letters/digits to add up to 917.
- When you put your password together, remember that the 2nd letter in your password must be 'S' then any combination of 9 letter and digits on the remaining spaces. I can up with 1 'e' (101) and 8 'f's (102).

PYTHON1.PY code:

```
#!/usr/bin/python

import sys

def main():
  if len(sys.argv) != 2:
    print "Invalid args"
    return
  password = sys.argv[1]
  builder = 0
  for c in password:
    builder += ord(c)
  if builder == 1000 and len(password) == 10 and ord(password[1]) == 83:
    print "correct"
  else:
    print "incorrect"

if __name__ == "__main__":
  main()
```

2) PYTHON2.PYC Answer: Password => mysupersecretpassword

What you know from the code:

- Entered password is being compared to 'vals'
- Password length = same as 'vals' (see code below)
- Entered password is shifted by 7 before being compared with 'vals'

How to:

- Must decompile .pyc file using uncompyle6 to get Python source code.
- Use unicode-table.com or any ASCII table to get the ASCII numeric value for the letters and digits.
- Review the Python program and you will see that it gives an encoded password to compare with your password but it performs a 7 letter shift to encode before comparing. This is a Caesar cipher or a shift cipher! You can solve by doing either of these 2 methods:
  - Hard: Do the math backwards on each letter of the encoded password.
    Example: 't' is 116, subtract 7 gives 109 which translates to 'm'. If the subtraction puts you beyond the alphabet range then add 26 after the subtraction. For example, 'f' is 102, subtract 7 gives 95 which is the character '_' so add 26 and the letter is 121 which is 'y'. Continue through each character.
  - Simple: Look up a Caesar cipher decoder and use a 7 character shift on 'tfzbwlyzljylawhzzdvyk' to decode the password.

Encoded Password: tfzbwlyzljylawhzzdvyk
Decoded Password: mysupersecretpassword

Decompiled PYTHON2.pyc

```
root@kali:~/Desktop/2018 NCL Gym# uncompyle6 PYTHON2.pyc
# uncompyle6 version 3.2.3
# Python bytecode 2.7 (62211)
# Decompiled from: Python 2.7.15 (default, Jul 28 2018, 11:29:29)
# [GCC 8.1.0]
# Embedded file name: NCL-2015-Python2.py
# Compiled at: 2015-11-12 17:43:01
import sys

def main():
   if len(sys.argv) != 2:
      print 'Invalid args'
      return
   password = sys.argv[1]
   counter = 0
```

```
    vals = list('tfzbwlyzljylawhzzdvyk')
    if len(password) != len(vals):
        print 'incorrect'
        return
    while counter < len(password):
        x = ord(password[counter]) + 7
        if x > ord('z'):
            x -= 26
        if chr(x) != vals[counter]:
            print 'incorrect'
            return
        counter += 1

    print 'correct'

if __name__ == '__main__':
    main()
# okay decompiling PYTHON2.pyc
```

3) PYTHON3.PYC   Answer: Password => NAAAAAAA777

What you know from the code:

- Password length = 11
- First character in the password is ASCII numeric value 78 or ‘N’

How to:

- Must decompile .pyc file using uncompyle6 to get Python source code
- Use unicode-table.com or any ASCII table to get the ASCII numeric value for the letters and digits.
- Review the Python program and you will see that it gives the first letter of the password as 78 which is the letter 'N', the length of the password is 11 letters or digits long, and the (hard part) password’s ASCII numeric code total must match the encoded total ASCII numeric value of builder, which is 1234538.
- Do the math backwards on builder value of 1234538. I wrote a Java program to do the bitwise XOR, shifts, and complements operations backwards on the builder. Using BlueJay, here is my Java program which prints out the steps backwards from PYTHON3.PY.
- The character total is 698 so you subtract the 78 which is the first character in your password and you're left with 620 as a character total which must be spread across the remaining 10 letters/digits. At this point, you can play with the ASCII table to get any combination of letters/digits to add up to 620. I came up with 7 'A's (65) and 3 '7's (55).
- When you put your password together, remember that the 1st letter in your password must be 'N' then any combination of 10 letter and digits afterwards.

Decompiled PYTHON3.pyc

```
root@kali:~/Desktop/2018 NCL Gym# uncompyle6 PYTHON3.pyc
# uncompyle6 version 3.2.3
# Python bytecode 2.7 (62211)
# Decompiled from: Python 2.7.15 (default, Jul 28 2018, 11:29:29)
# [GCC 8.1.0]
# Embedded file name: NCL-2015-Python3.py
# Compiled at: 2015-11-12 20:09:05
import sys

def main():
   if len(sys.argv) != 2:
      print 'Invalid args'
      return
   password = sys.argv[1]
   builder = 0
   for c in password:
      builder += ord(c)
```

```
    builder = builder << 2
    builder = ~builder
    builder = builder ^ 12648430
    builder = ~builder
    if builder == 12645638 and ord(password[0]) == 78 and len(password) == 11:
        print 'correct'
    else:
        print 'incorrect'

if __name__ == '__main__':
    main()
# okay decompiling PYTHON3.pyc
```

<u>Reverse calculations in PYTHON3.PY</u>

```
public class Python3
{
    public Python3() {}

    public static void main(String[] args)
    {
        int builder;
        builder = 12645638;
        System.out.println("Step 1: builder = " + builder);
        builder = ~builder;                    // bitwise complement
        System.out.println("Step 2: ~builder = " + builder);
        builder = builder ^ 12648430;          // bitwise XOR with 12648430
        System.out.println("Step 3: ^builder = " + builder);
        builder = ~builder;                    // bitwise complement
        System.out.println("Step 4: ~builder = " + builder);
        builder = builder >> 2;                // bitwise shift right
        System.out.println("Step 5: >>builder = " + builder);
    }
}
```

<u>Output is:</u>

```
Step 1: builder = 12645638
Step 2: ~builder = -12645639
Step 3: ^builder = -2793
Step 4: ~builder = 2792
Step 5: >>builder = 698
```

4) RE1_64bit 8532 => Flag is NCL-EZOF-2046

a) Used the hint to learn about using gdb (GNU debugger) for disassembly. Bring up a terminal window and type 'gdb RE1_64bit', then inside gdb type 'disassemble main'. The hint guide said you should deduce that the ‘gets’ call is vulnerable such that a buffer overflow can be triggered if you figure out the input buffer length from the disassembled code below. The guide says it is set to 30 so an overflow should happen at 31 characters but I found that the overflow happens at 45 characters.

b) To run the code, open a terminal window and type './RE1_64bit 8532'. The program will ask you for your password. The program will handle a length of 44 or less characters/digits normally. If you enter 45 or more characters/digits, it will cause a buffer overflow and the key will be revealed => NCL-EZOF-2046

Using gdb:

```
root@kali:~/Downloads# gdb RE1_64bit
GNU gdb (Debian 8.1-4) 8.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from RE1_64bit...(no debugging symbols found)...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000040084e <+0>:     push   %rbp
   0x000000000040084f <+1>:     mov    %rsp,%rbp
   0x0000000000400852 <+4>:     sub    $0x40,%rsp
   0x0000000000400856 <+8>:     mov    %edi,-0x34(%rbp)
   0x0000000000400859 <+11>:    mov    %rsi,-0x40(%rbp)
   0x000000000040085d <+15>:    cmpl   $0x2,-0x34(%rbp)
   0x0000000000400861 <+19>:    je     0x400886 <main+56>
   0x0000000000400863 <+21>:    mov    -0x40(%rbp),%rax
   0x0000000000400867 <+25>:    mov    (%rax),%rax
   0x000000000040086a <+28>:    mov    %rax,%rsi
   0x000000000040086d <+31>:    mov    $0x4009a2,%edi
   0x0000000000400872 <+36>:   mov    $0x0,%eax
   0x0000000000400877 <+41>:   callq  0x400580 <printf@plt>
   0x000000000040087c <+46>:   mov    $0x1,%edi
   0x0000000000400881 <+51>:   callq  0x4005e0 <exit@plt>
   0x0000000000400886 <+56>:   mov    -0x40(%rbp),%rax
   0x000000000040088a <+60>:   add    $0x8,%rax
   0x000000000040088e <+64>:   mov    (%rax),%rax
   0x0000000000400891 <+67>:   mov    %rax,%rdi
   0x0000000000400894 <+70>:   callq  0x400570 <strlen@plt>
   0x0000000000400899 <+75>:   cmp    $0x4,%rax
   0x000000000040089d <+79>:   je     0x4008c2 <main+116>
   0x000000000040089f <+81>:   mov    -0x40(%rbp),%rax
   0x00000000004008a3 <+85>:   mov    (%rax),%rax
   0x00000000004008a6 <+88>:   mov    %rax,%rsi
   0x00000000004008a9 <+91>:   mov    $0x4009a2,%edi
   0x00000000004008ae <+96>:   mov    $0x0,%eax
   0x00000000004008b3 <+101>:  callq  0x400580 <printf@plt>
   0x00000000004008b8 <+106>:  mov    $0x1,%edi
   0x00000000004008bd <+111>:  callq  0x4005e0 <exit@plt>
   0x00000000004008c2 <+116>:  movl   $0x0,-0x4(%rbp)
   0x00000000004008c9 <+123>:  mov    $0x4009b3,%edi
   0x00000000004008ce <+128>:  mov    $0x0,%eax
   0x00000000004008d3 <+133>:  callq  0x400580 <printf@plt>
   0x00000000004008d8 <+138>:  lea    -0x30(%rbp),%rax
   0x00000000004008dc <+142>:  mov    %rax,%rdi
   0x00000000004008df <+145>:  callq  0x4005d0 <gets@plt>
   0x00000000004008e4 <+150>:  cmpl   $0x0,-0x4(%rbp)
   0x00000000004008e8 <+154>:  je     0x4008f8 <main+170>
   0x00000000004008ea <+156>:  mov    -0x40(%rbp),%rax
   0x00000000004008ee <+160>:  mov    %rax,%rdi
   0x00000000004008f1 <+163>:  callq  0x4006dd <fg>
   0x00000000004008f6 <+168>:  jmp    0x400902 <main+180>
   0x00000000004008f8 <+170>:  mov    $0x4009cd,%edi
   0x00000000004008fd <+175>:  callq  0x400560 <puts@plt>
   0x0000000000400902 <+180>:  mov    $0x0,%eax
   0x0000000000400907 <+185>:  leaveq
   0x0000000000400908 <+186>:  retq
End of assembler dump.
```

<u>To run the binary:</u>

```
root@kali:~/Downloads# ./RE1_64bit 8532
Please enter a password: 1223124233465337987394729834987923749792787394 79
your tid: 8532
NCL-EZOF-2046
```

5) **RE2_64bit => Flag is NCL-FYOF-6840**

a) Used the hint to learn about using gdb for function information, setting a breakpoint in main, and calling the validation function by TID in gdb. Bring up a terminal window and type 'gdb RE2_64bit', then inside gdb type 'info functions'. The hint guide said to notice the function 'getflagbytid'.

b) Set a breakpoint in main by typing 'break main', then run the program by typing 'r'. The program will break at the top of main.

c) The hard part about this problem is that if you type in 'call getflagbytid(7026)' it will give you the following error " 'getflagbytid' has unknown return type; cast the call to its declared return type" and exit. You must figure out what the return data type is. What I deduced from looking at the disassemble code is the NCL flag is typically a string and the program appears to be a C program. Given this information, the data type String is not a recognized data type in C. C uses character arrays to store character strings so to pass a string in or out of a C function, you usually reference a character array using a char pointer or (char *).

d) To make this call work, you should type in 'call (char *) getflagbytid(7026)' and the flag will be revealed for the TID of 7026 which is given to you in the problem description. The flag => NCL-EZOF-2046

e) One thing I noticed about the main program....it never calls the function 'getflagbytid' which means the only way to solve this problem is to use gdb or a debugger to manually call this function with the TID.

<u>To run the binary:</u>

```
root@kali:~/Downloads# ./RE2_64bit
bash: ./RE2_64bit: Permission denied
root@kali:~/Downloads# chmod 777 RE2_64bit
root@kali:~/Downloads# ./RE2_64bit
usage: ./RE2_64bit <tid>
root@kali:~/Downloads# ./RE2_64bit 7026
Please enter a password: 1234567890123456789012345678901234567890
Getting the flag from the program.... password incorrect
root@kali:~/Downloads#
```

<u>Using gdb:</u>

```
root@kali:~/Downloads# gdb RE2_64bit
GNU gdb (Debian 8.1-4) 8.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

```
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from RE2_64bit...(no debugging symbols found)...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000400990 <+0>:     push   %rbp
   0x0000000000400991 <+1>:     mov    %rsp,%rbp
   0x0000000000400994 <+4>:     sub    $0x50,%rsp
   0x0000000000400998 <+8>:     mov    %edi,-0x44(%rbp)
   0x000000000040099b <+11>:    mov    %rsi,-0x50(%rbp)
   0x000000000040099f <+15>:    cmpl   $0x2,-0x44(%rbp)
   0x00000000004009a3 <+19>:    je     0x4009c8 <main+56>
   0x00000000004009a5 <+21>:    mov    -0x50(%rbp),%rax
   0x00000000004009a9 <+25>:    mov    (%rax),%rax
   0x00000000004009ac <+28>:    mov    %rax,%rsi
   0x00000000004009af <+31>:    mov    $0x400b94,%edi
   0x00000000004009b4 <+36>:    mov    $0x0,%eax
   0x00000000004009b9 <+41>:    callq  0x4006a0 <printf@plt>
   0x00000000004009be <+46>:    mov    $0x1,%edi
   0x00000000004009c3 <+51>:    callq  0x400710 <exit@plt>
   0x00000000004009c8 <+56>:    mov    -0x50(%rbp),%rax
   0x00000000004009cc <+60>:    add    $0x8,%rax
   0x00000000004009d0 <+64>:    mov    (%rax),%rax
   0x00000000004009d3 <+67>:    mov    %rax,%rdi
   0x00000000004009d6 <+70>:    callq  0x400690 <strlen@plt>
   0x00000000004009db <+75>:    cmp    $0x4,%rax
   0x00000000004009df <+79>:    je     0x400a04 <main+116>
   0x00000000004009e1 <+81>:    mov    -0x50(%rbp),%rax
   0x00000000004009e5 <+85>:    mov    (%rax),%rax
   0x00000000004009e8 <+88>:    mov    %rax,%rsi
   0x00000000004009eb <+91>:    mov    $0x400b94,%edi
   0x00000000004009f0 <+96>:    mov    $0x0,%eax
   0x00000000004009f5 <+101>:   callq  0x4006a0 <printf@plt>
   0x00000000004009fa <+106>:   mov    $0x1,%edi
   0x00000000004009ff <+111>:   callq  0x400710 <exit@plt>
   0x0000000000400a04 <+116>: mov    -0x50(%rbp),%rax
   0x0000000000400a08 <+120>: add    $0x8,%rax
   0x0000000000400a0c <+124>: mov    (%rax),%rax
   0x0000000000400a0f <+127>: mov    $0xa,%edx
   0x0000000000400a14 <+132>: mov    $0x0,%esi
   0x0000000000400a19 <+137>: mov    %rax,%rdi
   0x0000000000400a1c <+140>: callq  0x4006e0 <strtol@plt>
   0x0000000000400a21 <+145>: mov    %eax,-0x4(%rbp)
   0x0000000000400a24 <+148>: mov    $0x400ba5,%edi
   0x0000000000400a29 <+153>: mov    $0x0,%eax
   0x0000000000400a2e <+158>: callq  0x4006a0 <printf@plt>
   0x0000000000400a33 <+163>: lea    -0x40(%rbp),%rax
   0x0000000000400a37 <+167>: mov    %rax,%rsi
   0x0000000000400a3a <+170>: mov    $0x400bbf,%edi
   0x0000000000400a3f <+175>: mov    $0x0,%eax
   0x0000000000400a44 <+180>: callq  0x400700 <__isoc99_scanf@plt>
   0x0000000000400a49 <+185>: mov    $0x400bc8,%edi
   0x0000000000400a4e <+190>: mov    $0x0,%eax
   0x0000000000400a53 <+195>: callq  0x4006a0 <printf@plt>
   0x0000000000400a58 <+200>: mov    0x200681(%rip),%rax        # 0x6010e0
<stdout@@GLIBC_2.2.5>
   0x0000000000400a5f <+207>: mov    %rax,%rdi
   0x0000000000400a62 <+210>: callq  0x4006f0 <fflush@plt>
   0x0000000000400a67 <+215>: mov    $0x1,%edi
   0x0000000000400a6c <+220>: mov    $0x0,%eax
   0x0000000000400a71 <+225>: callq  0x400720 <sleep@plt>
   0x0000000000400a76 <+230>: mov    $0x2e,%edi
   0x0000000000400a7b <+235>: callq  0x400670 <putchar@plt>
   0x0000000000400a80 <+240>: mov    0x200659(%rip),%rax        # 0x6010e0
<stdout@@GLIBC_2.2.5>
   0x0000000000400a87 <+247>: mov    %rax,%rdi
   0x0000000000400a8a <+250>: callq  0x4006f0 <fflush@plt>
   0x0000000000400a8f <+255>: mov    $0x1,%edi
   0x0000000000400a94 <+260>: mov    $0x0,%eax
   0x0000000000400a99 <+265>: callq  0x400720 <sleep@plt>
   0x0000000000400a9e <+270>: mov    $0x2e,%edi
   0x0000000000400aa3 <+275>: callq  0x400670 <putchar@plt>
   0x0000000000400aa8 <+280>: mov    0x200631(%rip),%rax        # 0x6010e0
<stdout@@GLIBC_2.2.5>
   0x0000000000400aaf <+287>: mov    %rax,%rdi
   0x0000000000400ab2 <+290>: callq  0x4006f0 <fflush@plt>
   0x0000000000400ab7 <+295>: mov    $0x1,%edi
   0x0000000000400abc <+300>: mov    $0x0,%eax
   0x0000000000400ac1 <+305>: callq  0x400720 <sleep@plt>
   0x0000000000400ac6 <+310>:  mov    $0x2e,%edi
   0x0000000000400acb <+315>:  callq  0x400670 <putchar@plt>
   0x0000000000400ad0 <+320>: mov    0x200609(%rip),%rax        # 0x6010e0
<stdout@@GLIBC_2.2.5>
   0x0000000000400ad7 <+327>: mov    %rax,%rdi
   0x0000000000400ada <+330>: callq  0x4006f0 <fflush@plt>
   0x0000000000400adf <+335>:  mov    $0x1,%edi
   0x0000000000400ae4 <+340>: mov    $0x0,%eax
   0x0000000000400ae9 <+345>: callq  0x400720 <sleep@plt>
   0x0000000000400aee <+350>: mov    $0x400beb,%edi
   0x0000000000400af3 <+355>:  callq  0x400680 <puts@plt>
   0x0000000000400af8 <+360>: mov    $0x0,%eax
   0x0000000000400afd <+365>:  leaveq
   0x0000000000400afe <+366>:  retq
End of assembler dump.
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0000000000400640  _init
0x0000000000400670  putchar@plt
0x0000000000400680  puts@plt
0x0000000000400690  strlen@plt
0x00000000004006a0  printf@plt
0x00000000004006b0  memset@plt
0x00000000004006c0  __libc_start_main@plt
0x00000000004006d0  __gmon_start__@plt
0x00000000004006e0  strtol@plt
0x00000000004006f0  fflush@plt
0x0000000000400700  __isoc99_scanf@plt
0x0000000000400710  exit@plt
0x0000000000400720  sleep@plt
0x0000000000400730  _start
0x0000000000400760  deregister_tm_clones
0x0000000000400790  register_tm_clones
0x00000000004007d0  __do_global_dtors_aux
0x00000000004007f0  frame_dummy
0x000000000040081d  getflagbytid
0x0000000000400990  main
0x0000000000400b00  __libc_csu_init
0x0000000000400b70  __libc_csu_fini
0x0000000000400b74  _fini
(gdb) break main
Breakpoint 1 at 0x400994
```

```
(gdb) r
Starting program: /root/Downloads/RE2_64bit

Breakpoint 1, 0x0000000000400994 in main ()
(gdb) call getflagbytid(7026)
'getflagbytid' has unknown return type; cast the call to its declared return type
(gdb) call (char *) getflagbytid(7026)
NCL-FYOF-6840
$1 = 0xe <error: Cannot access memory at address 0xe>
(gdb)
```