
LAYER ZERO
UNLV CYBER SECURITY CLUB

Enumeration

Note

- All examples are included in the Layer-Zero 2019-2020 git repository under week-5/enumeration/examples

What is enumeration?

- Analyzing software and services for weakness and vulnerabilities that can later be exploited
- In the scope of NCL, this usually means finding a hole in some program that will enable you to find the flag

Types of Programming Languages

- Compiled - Languages which must be first turned into a binary using a compiler before running
  - Examples: C++, Go, Rust
- Interpreted - Languages which are executed straight from source code
  - Examples: Python, Ruby
- Byte-code based - Languages in which source code is translated into an intermediate byte-code. This byte-code is then compiled into assembly right when it is to be run.
  - Examples: Java, C#, (Python .pyc files)

Tools for Analysis

- Interpreted & Byte-code Languages:
  - Because of the nature of these languages, programs exist that can take the byte-code and translate it back to source code -> EASY analysis
  - Examples: “CFR Java Decompiler”, “Uncompyle Python Decompiler”
- Compiled Languages:
  - Much harder to analyze due to only have the binary available
  - Requires familiarity with assembly language and the API for the specific OS
    - Linux / Windows assembly calling conventions are different!
  - Binary Analysis tools:
    - IDA64 (Industry standard binary analysis tool) - EXPENSIVE :(
    - GDB - standard binary debugger - FREE
    - Radare2 - amazing, open-source tool like IDA64 - FREE

Types of Analysis

Static Analysis

- Analysis of programs without running them
- Includes:
  - Finding strings within programs
  - Identifying functions
  - Analysis of program flow
  - OS, Language-type
- Can only get you so far

Dynamic Analysis

- Actually running code
- Involves:
  - Analysis of function behavior
  - Analysis of network traffic
  - Memory usage
  - Much more

IMPORTANT

Running random programs is extremely dangerous and should only be done within an isolated sandbox. While NCL programs are not malicious, you should never get into the habit of running unknown programs without precaution. Running a malicious program can allow your entire system to be owned immediately

Python Examples

Python1

- Most easy challenges simply require you to analyze the source code and figure out a solution
- To run a Python program, enter “python3 <file_name>”
- Python is great to experiment with if you’re not familiar with it because a command line interpreter exists by running “python3”. This will allow you run code snippets on the spot.

```python
"""
Python Example 1
"""


def checkPassword(password):
    if len(password) < 8:
        return False

    if sum([ord(x) for x in password]) != 960:
        return False

    if ord(password[5]) != 91 or sum([ord(x) for x in password[:3]]) != 260:
        return False

    return True


if __name__ == '__main__':
    userInput = input("What's the password?: ")
    if checkPassword(userInput):
        print('Yup')
    else:
        print('Nope. Try again')
```

Solution to Python1

1. Look at the source code
2. Understand the program flow
    a. Retrieves a password from the user
    b. Checks if the password is good via “checkPassword”
3. In “checkPassword” some conditions must be met:
    a. A password with length >= 8
    b. The ascii sum of all characters must equal 960
    c. The 6th character must be of ascii value 91
    d. The ascii sum of the first 3 characters must equal 260
4. Once all these conditions are met, the program will accept the input

Python2

- Although Python is an interpreted language, the Python virtual machine will sometimes compile the source code to Python byte-code for faster loading. These files are denoted by the .pyc file extension.
- It is possible to decompile these files back to source code using a decompiler.
- “Uncompyle Decompiler”
  - Install using “pip3 install uncompyle”
    - Python3 and pip3 must be installed first
  - Decompile using “uncompyle6 <.pyc file>”
    - The output will be the source code

```
mother@DAUGHTERSHIP:~/Documents/college/layer_zero/training-sessions/2019-2020/week-5/enumeration/examples/python$ uncompyle6 python2.pyc
# uncompyle6 version 3.2.6
# Python bytecode 3.6 (3379)
# Decompiled from: Python 3.6.7 (default, Oct 22 2018, 11:32:17)
# [GCC 8.2.0]
# Embedded file name: /home/mother/Documents/college/layer_zero/training-sessions/2019-2020/week-5/enumeration/examples/python/python2.py
# Compiled at: 2019-04-05 09:04:08
# Size of source mod 2**32: 336 bytes


def checkPassword(password):
    if password == "Sike that's the wrong number":
        return True
    else:
        return False


if __name__ == '__main__':
    userInput = input('Give me the formuoli: ')
    if checkPassword(userInput):
        print('Thats the correct formuoli')
    else:
        print('That aint the formuoli')
# okay decompiling python2.pyc
mother@DAUGHTERSHIP:~/Documents/college/layer_zero/training-sessions/2019-2020/week-5/enumeration/examples/python$
```

Binary Analysis

Binary Analysis

- Unlike the previous examples, we cannot get the original source code from binary files, only assembly language instructions
- Many of the hard style problems require digging through and understanding the assembly in the binary
- HOWEVER many problems exist where serious knowledge of it is not required :)
- TOOLS:
  - Radare2 - “apt install radare2”
  - unix-tools

binSamp

- The first thing to do when receiving a binary is discover its main characteristics:
  - What language?
  - Target OS?
  - Which architecture? - x86? ARM?

Binary Characteristics with R2

- Radare2 is an amazing tool for binary analysis
- Loading a file: “r2 <binary_file>”
- Radare has its own command line
  - i - give binary info

```
mother@DAUGHTERSHIP:~/Documents/college/layer_zero/training-sessions/2019-2020/week-5/enu
meration/examples/binaries$ r2 binSamp
[0x00000d10]> i
blksz    0x0
block    0x100
fd       3
file     binSamp
format   elf64
iorw     false
mode     -r-x
size     0x38e8
humansz  14.2K
type     DYN (Shared object file)
arch     x86
binsz    12578
bintype  elf
bits     64
canary   false
class    ELF64
crypto   false
endian   little
havecode true
intrp    /lib64/ld-linux-x86-64.so.2
lang     cxx
linenum  true
lsyms    true
machine  AMD x86-64 architecture
maxopsz  16
minopsz  1
nx       true
os       linux
pcalign  0
pic      true
relocs   true
relro    partial
```

Assembly Format (Linux)

C++

x86

Useful Radare Commands

- i - info
- iz - list all strings within binary (Extremely useful for hardcoded strings)
- aaa - analyze binary symbols and functions
- afl - list all functions
- s <function or address> - makes a marker at the function / address
- pdf @<function_name> - “print / display function”
    - NOTE: MUST analyze using aaa, before functions become available to use as symbols

For this problem, when we see which strings are in the file, we get these. Are any a possible password??

```
[0x00000d10]> iz
000 0x000010f5 0x000010f5  20  21 (.rodata) ascii Enter the password:
001 0x0000110a 0x0000110a  10  11 (.rodata) ascii lAYer-z3R0
002 0x00001115 0x00001115   8   9 (.rodata) ascii Correct!
003 0x0000111e 0x0000111e  10  11 (.rodata) ascii Incorrect!
```

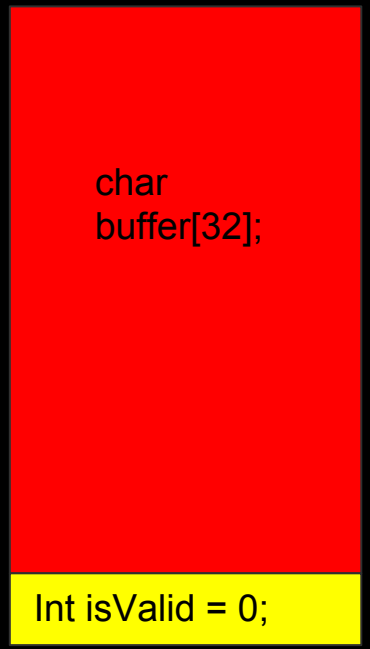
Buffer Overflow

- Buffer - a finite section of memory reserved for storing data
  - Since this section is finite, if the program does check for overfilling of this section, memory surrounding the buffer can be overwritten, causing problems
  - Often this can cause a segmentation fault
  - IF done carefully, this overwritten data can have unintended consequences

C Code Example

```
Int main() {
        char buffer[32];
        Int isValid = 0;
        gets(buffer);          // get input
        If (checkPassword(buffer) == 1) printf(“You did it!”);
        Else print(“You failed! Yay.”);
}
```

Memory Layout



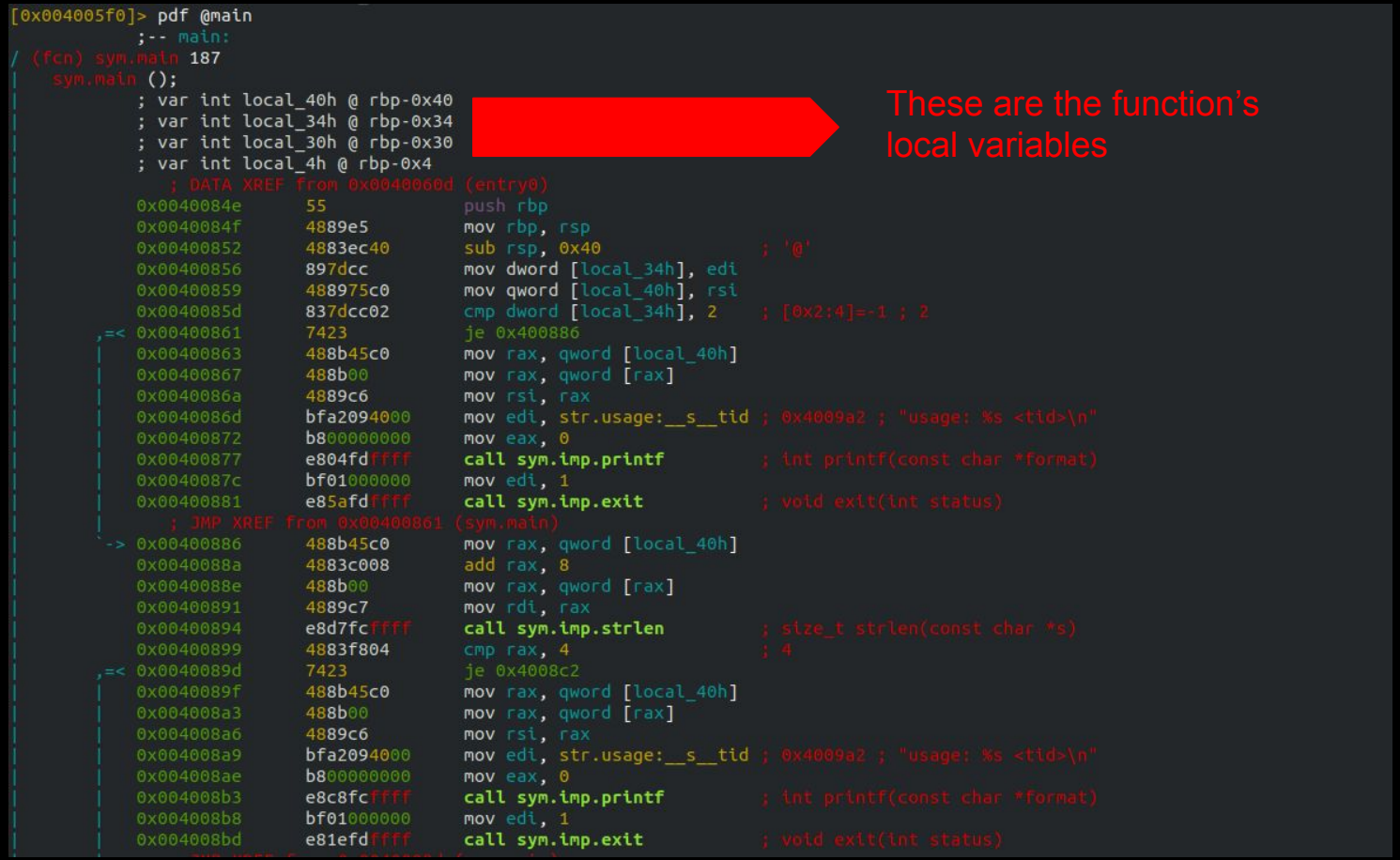
What happens if the buffer gets more than 32 values?

isValid changes its value from 0 to something else, which completely changes the behavior of the program

Analysis

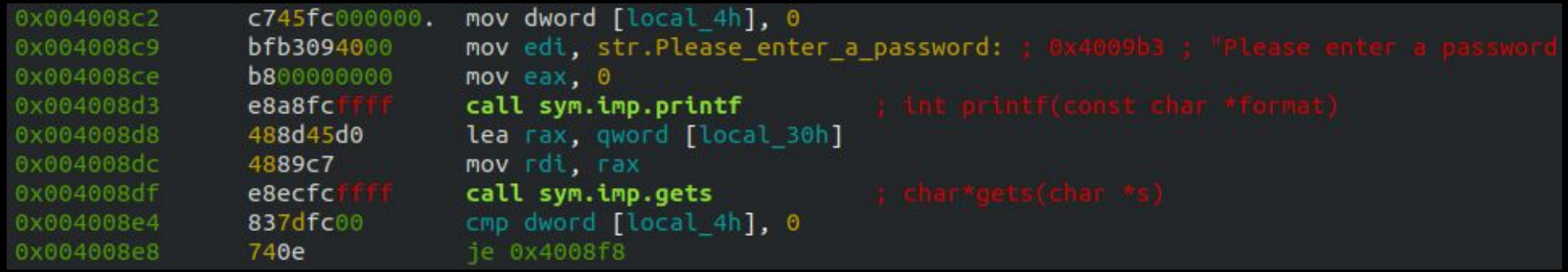
1. Load in r2 - “r2 harder”
2. Analyze symbols - “aaa”
3. Set a marker at main - “s main”
4. Let’s use R2’s visual mode
   a. VV
   b. Able to navigate using “hjkl”
   c. Shows a flow-chart view of program execution

“Harder” sample

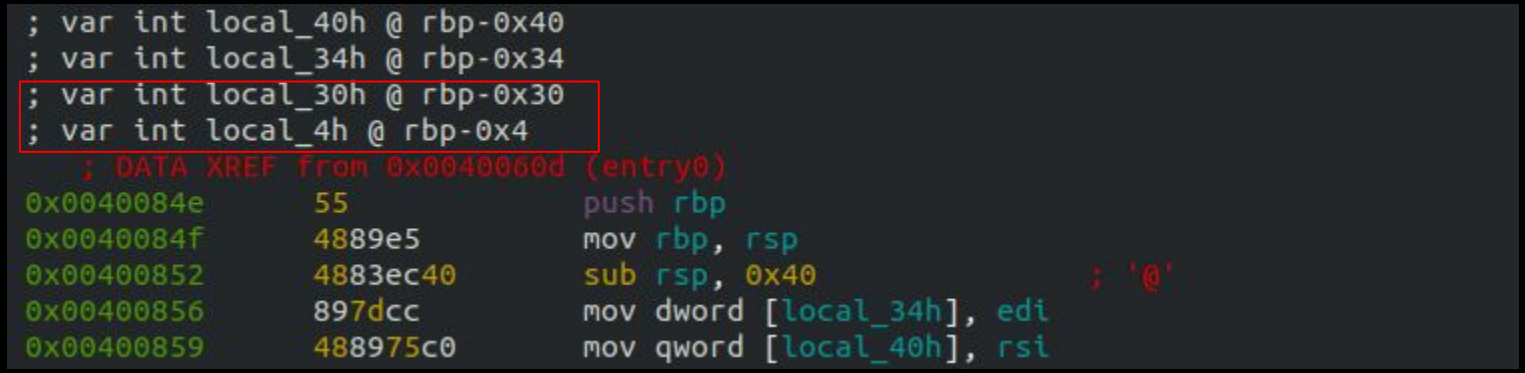


```
; var int local_40h @ rbp-0x40
; var int local_34h @ rbp-0x34
; var int local_30h @ rbp-0x30
; var int local_4h @ rbp-0x4
   ; DATA XREF from 0x0040060d (entry0)
0x0040084e      55              push rbp
0x0040084f      4889e5          mov rbp, rsp
0x00400852      4883ec40        sub rsp, 0x40                ; '@'
0x00400856      897dcc          mov dword [local_34h], edi
0x00400859      488975c0        mov qword [local_40h], rsi
```

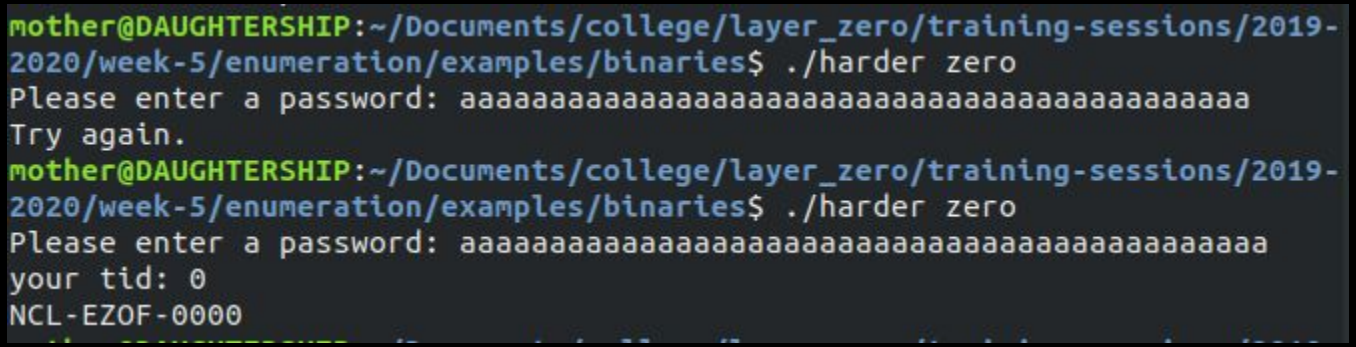
- Int main (argc, argv) {}
- Local_34 - argc
- Local_40 - argv
- What are local_30h and local_4h?
  - The numbers of the locals describe what byte the item is on the stack
  - Local_4 is 4 bytes long (4 - 0) -> int?
  - Local_30 is 44 bytes long (48 - 4) -> some array?
  - Local_34 is 4 bytes (52 - 48) -> int == argc
  - Local_40 is 12 bytes (64 - 52) -> array of pointers == argv



- At the point of getting a password, the program changes the value of local_4h to 0. -> this indicates either the start of a count or a bool value
- The program then uses the array, local_30h as the parameter to gets(), which retrieves user input
- After getting the input, it checks if local_4h is still 0. If it is, the program says to try again. If not, the program calls another function. How does local_4h change? Lets try an overflow



- Like we saw before, the size of our array, local_30h, is (0x30 - 0x04 == 44).
- This means that its total capacity is 44. Lets try a few inputs



- Trying an input of 44 a’s results in an invalid entry
- Trying an input of 45 a’s results in an VALID entry. We have successfully overflowed the buffer!