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

As our project we participate in the 2019 edition of the Nixu Challenge. The Nixu Challenge is a yearly Capture-The-Flag (CTF) event organized by the cyber security company Nixu Corporation. In a CTF event the participants try to complete various challenges with the objective of retrieving specific tokens. The Nixu Challenge consists of a variety of challenges related to subjects like web security, memory analysis, cryptography and reverse engineering. The difficulty level range from simple problems solvable in a couple of minutes to complex challenges.

Our goal is not to solve every single challenge but to beat as many as possible and report how we solved them and what we've learned. We also want to complete diverse challenges of all difficulties for the report to touch on as many subject as possible.

II. Challenges

A. AIMLES - staging

We are given a network capture in the form of a `.pcap` and a url:port pair to a server running SSH. Trying to SSH into the server gives an error about no matching ciphers. Therefore, we decide to look at the network capture which consists of SSH traffic. Looking around in the packets, we can see that the shell session is not encrypted. We use `strings` on the pcap file to extract the readable text, which gives us some commands that were run on the server and a email about the security audit, which contains 4 hints. The fourth hint in the email confirms our doubt that there is no encryption cipher offered by the SSH server. We need to compile the openssh client with a small modification to allow us to connect to the SSH server using the `none` cipher. Once we try to connect, we need to authenticate using a key. An other hint from the email tell us that the encryption key used by the two employees to connect to the server may have a weakness. After having extracted the public keys from the network capture, we used RsaCtfTool to performs an attack against the two public keys to find a common factor and recover the private keys. We are now able to go one step further, but the server asks for a time-based one-time password (TOTP). Using the other hints in the email, we know the validity of the TOTP is 5 minutes. Also, the user ran a command on the file containing the TOTP secret that gives us what number appears in the secret and that there is only consonants and numbers. The user also ran `ls`, which gives us the length (8 chars) of the secret and he ran `md5sum`, which gives us the MD5 hash of the secret. Using hashcat and the hints we have about the secret, we brute forced the MD5 hash to find the value of the TOTP secret. With the secret, we are able to generate a TOTP that is valid for 5 minutes and finaly connect to the SSH server to retreive the flag!

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

In this part of the forensics challenge, we need to look for a lost file. To do this we need to search for files present in the main memory dump and more importantly files that have been deleted or moved to the recycle bin. There are multiple commands available in volatility to search for files such as `filescan`, `dumpfiles` and `mftparser` . We had success with `mftparser`. Using the following command `mftparser --dump-dir=output --output-file=badmem_mft.body --output=body`, we get a list of extracted files in `badmem_mft.body` and the extracted files in the `output` folder. Knowing we are looking for a lost file, we search for the recycle bin like this`cat badmem_mft.body | grep -i "recycle"` which gives us about 10 results. We try after to display the files in the output and we finaly find one that is interesting (`cat output/file.0x286f8400.data0.dmp`). This content is Base64 encoded which once converted becomes a string of 0 and 1 that can be convert to a ASCII string that is the flag.

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

This is the first challenge in a serie of three challenges based on reversing programs from an old computer architecture. We are given a manual of how the architecture works and what are the instructions and opcodes. We started by dividing the bytes manually into the appropriate sections and translating progressively the instructions. Soon enough, we can understand what the program does and find a pattern in the instructions. The program push two numbers to the stack and substract them, which gives an ASCII char and by doing a few of the substrations manually, we can see the string as the format of the flag (NIXU...). We wrote a small Python script to read the binary, find the substraction instruction and do in operation on the numbers, which allowed us to recover the full flag.

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}
```

S. Stowaway

III. Conclusion

IV. References