Chunky
Malware
100 Points

Both challenges in the Malware category were able to be solved using tools and steps in what is commonly referred to as the “Basic Analysis” phase. This means that no debugger, disassembler, or decompiler is necessary to solve these challenges.

All the tools required to solve this challenge are installed by default with FlareVM, one of the best malware analysis VMs out there, made by Google’s Mandiant: https://github.com/mandiant/flare-vm.

This challenge could be solved in two ways: statically or dynamically.

Statically, the first step is to run the program through strings, or since it’s an executable, Floss:

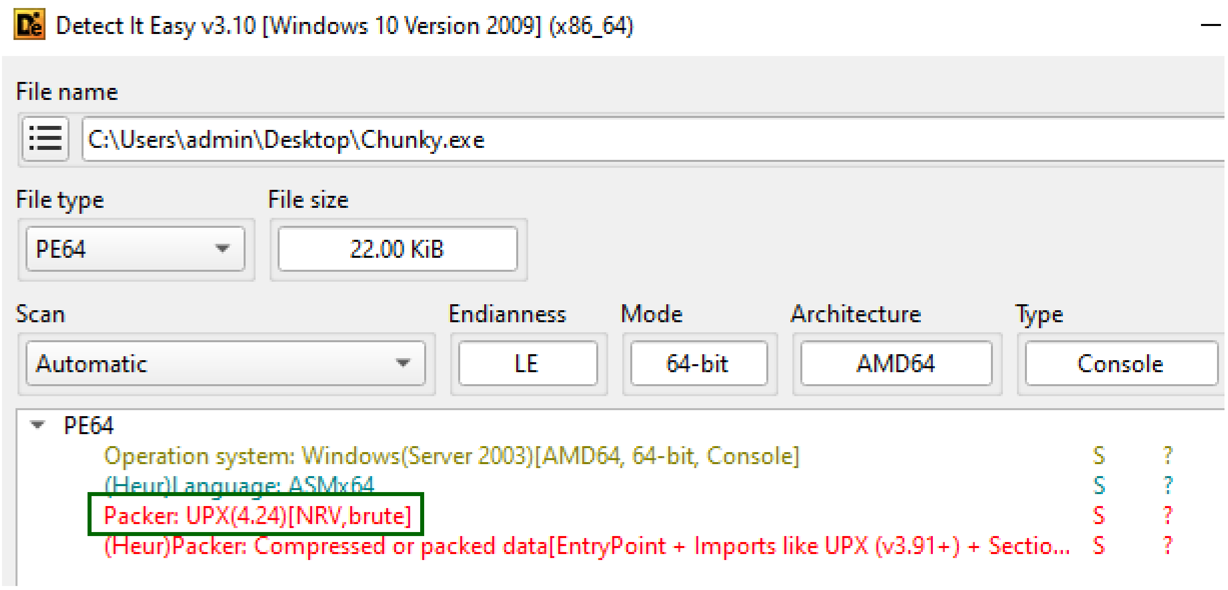
```
C:\Users\admin\Desktop
λ floss -n 5 "Chunky.exe" > floss.txt
INFO: floss: extracting static strings
finding decoding function features: 100%|                    | 3/3 [00:00<?,
INFO: floss.stackstrings: extracting stackstrings from 2 functions
extracting stackstrings: 100%|
INFO: floss.tightstrings: extracting tightstrings from 1 functions...
extracting tightstrings from function 0x140012132: 100%|
INFO: floss.string_decoder: decoding strings
emulating function 0x140012132 (call 1/1): 100%|
INFO: floss: finished execution after 2.72 seconds
INFO: floss: rendering results
```

Floss is another tool made by Mandiant that has several string decryption techniques built into it that the regular strings utility doesn’t have, but it only works on PE executables for Windows.

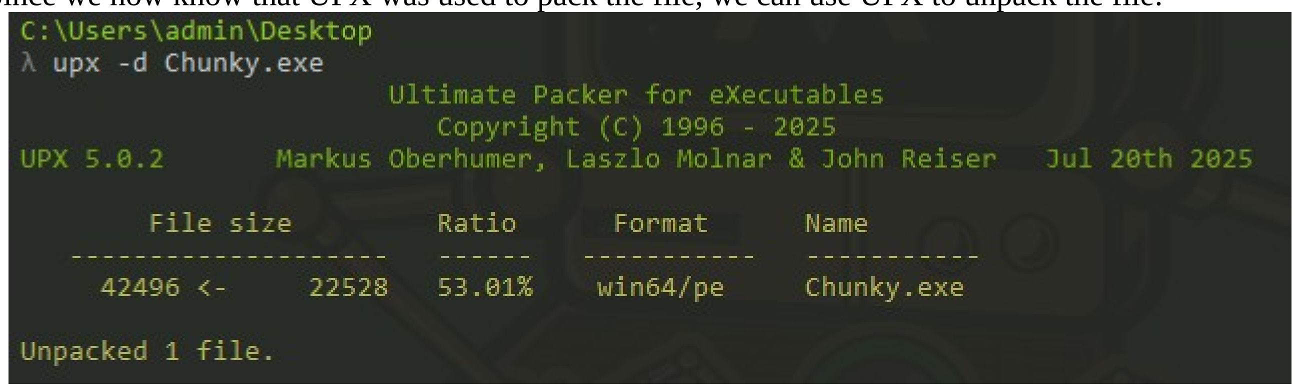
Looking through the results of the floss.txt file, nothing of interest stands out, which is a strong indicator that the executable is packed. To confirm, we can use a few different tools, one of which being Capa:

```
C:\Users\admin\Desktop
λ capa Chunky.exe
WARNING  capa.capabilities.common:
         ------------------------------------------------------------
WARNING  capa.capabilities.common:  This sample appears to be packed.
WARNING  capa.capabilities.common:
```

Capa confirms it is packed, but doesn’t tell us what packer was used. Another tool we can use is Detect it Easy (DiE). Opening DiE and putting Chunky.exe into it reveals the type of packer used:



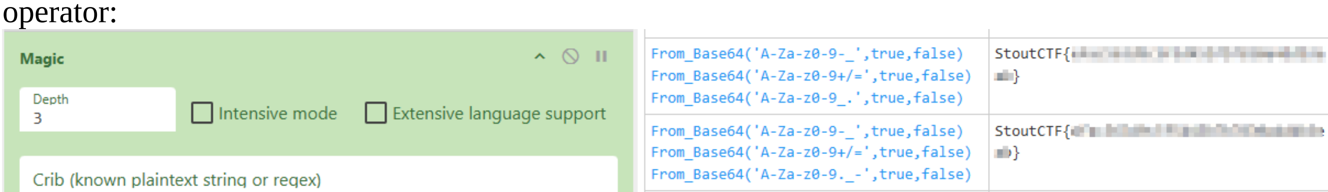
Since we now know that UPX was used to pack the file, we can use UPX to unpack the file:



Now that the file is unpacked, we can run Floss on it again to see what it finds. Looking through the results, there is one long strings ending in ==:

```
[^_]A\A]A^A_
127.0.0.1 %s
%s.local
C:\Windows\System32\drivers\etc\hosts
V1ROU2RtUllVa1JXUlZvM1dsZGFhRmw2VlRKTk1sRTFXWHBXYlU1WFVUQk9WMGt6V1dwak5FMTZRbWhPUjFKcldXcE9iRmxYU2prPQ==
127.0.0.1
POST /submit HTTP/1.1
Host: %s
Content-Length: %zu
```

This can be decrypted with CyberChef (https://gchq.github.io/CyberChef) using either the Magic operator:

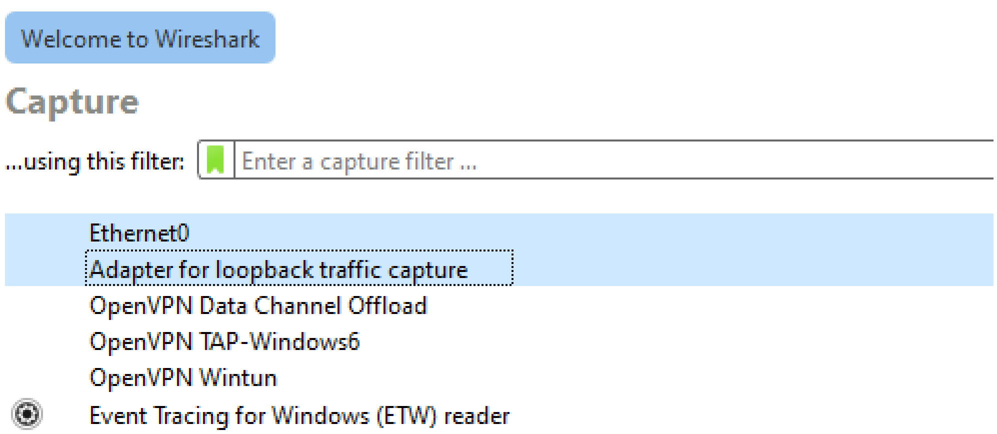


Or three Base64 operators:

Dynamically, this challenge can be solved with just two tools: Wireshark and CyberChef.

As this is simulating malware, it should be run in a virtual machine with no internet connection.

Wireshark is one of the most useful tools when analyzing malware, as it can discover any external domains the file connects to, or any local transactions it makes. To start, highlight both the Ethernet adapter and the Loopback adapter, and click the blue fin in the top left:



Once started, double click the file to run it. Since the VM has no internet connection, the traffic generated should only come from the challenge file. The file generates a bit of traffic, a lot of TCP requests and a few HTTP requests. If we sort by just the HTTP requests by typing ‘http’ in the filter field, we can filter down to the details:

http

No.	Time	Source	Destination	Protocol	Lengtl	Info
4	0.000462	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
12	0.151837	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
21	0.308618	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
30	0.464320	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
39	0.620831	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
47	0.777133	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
56	0.933048	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
64	1.089299	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
73	1.245735	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
81	1.401689	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
90	1.558486	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
99	1.715178	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1
108	1.870893	127.0.0.1	127.0.0.1	HTTP	297	POST /submit HTTP/1.1

If we click on one of these, we can see that there's 8 bytes of data:

```
> Frame 4: 297 bytes on wire (2376 bits), 297 bytes captured (2376 bits) on interface \Devic
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 49676, Dst Port: 8080, Seq: 1, Ack: 1, Len: 253
> Hypertext Transfer Protocol
v Data (8 bytes)
    Data: 566c524f55325274
    [Length: 8]
```

This data is in hexadecimal format. If we click on the data, it will highlight it on the right side, with both the hexadecimal and ASCII text types, where we can see the value of the data:

```
0a 43 6f 6e 74 65 6e 74  2d 4c 65 6e 67 74 68 3a   ·Content -Length:
20 38 0d 0a 43 6f 6e 74  65 6e 74 2d 54 79 70 65    8··Cont ent-Type
3a 20 61 70 70 6c 69 63  61 74 69 6f 6e 2f 6f 63   : applic ation/oc
74 65 74 2d 73 74 72 65  61 6d 0d 0a 43 6f 6e 6e   tet-stre am··Conn
65 63 74 69 6f 6e 3a 20  63 6c 6f 73 65 0d 0a 0d   ection:  close···
0a 56 6c 52 4f 55 32 52  74                        ·VlROU2R t
```

Each one of the HTTP requests has additional data, which when appended to each other, will generate the full string shown earlier in the Floss output. And just like with the static analysis, we can use CyberChef to decode the data into the flag.

If you want to learn more about malware analysis, HuskyHacks has a great YouTube course on it: https://www.youtube.com/watch?v=qA0YcYMRWyI