Unpacking ICEDID | Elastic

elastic.co/security-labs/unpacking-icedid

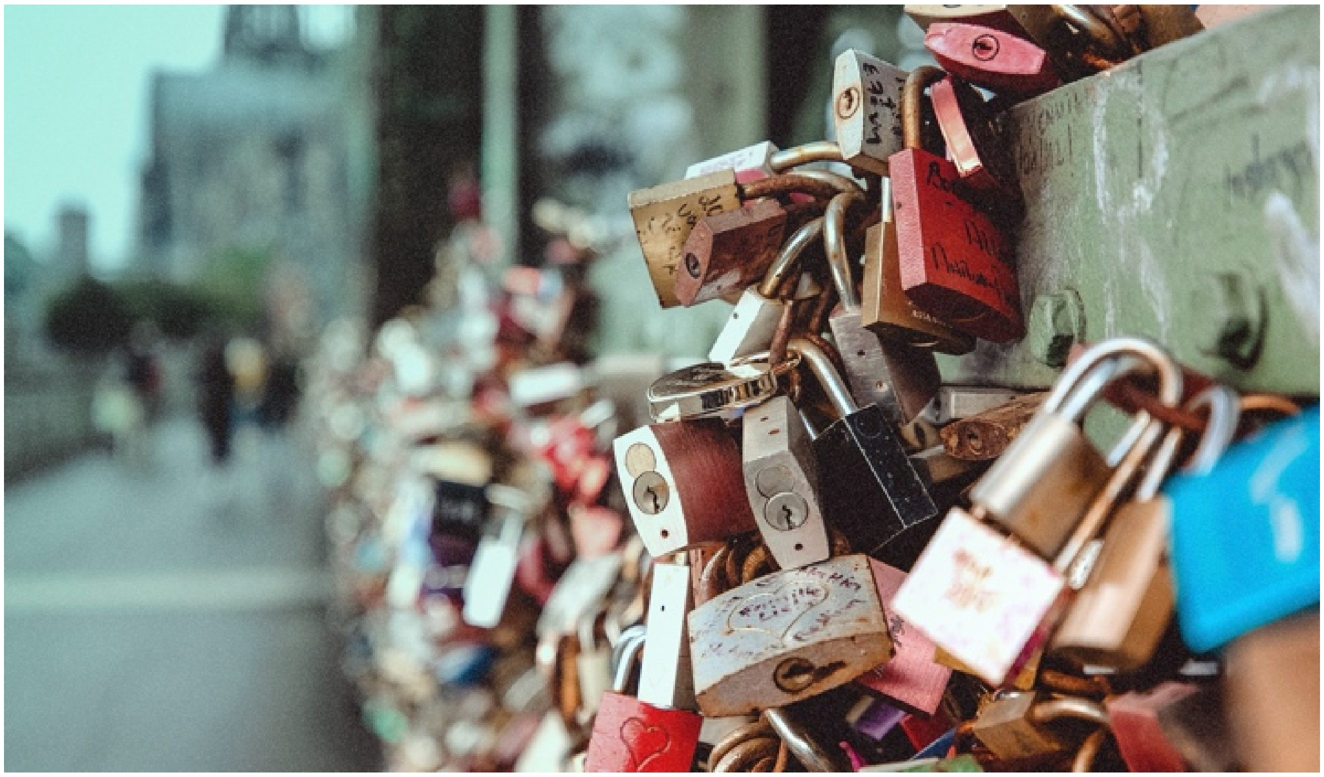
Unpacking ICEDID

A comprehensive tutorial with Elastic Security Labs open source tools

By

Cyril François

04 May 2023
English



Preamble

ICEDID is a malware family discovered in 2017 by IBM X-force researchers and is associated with the theft of login credentials, banking information, and other personal information. ICEDID has always been a prevalent family but achieved even more growth since EMOTET's temporary disruption in early 2021. ICEDID has been linked to the distribution of several distinct malware families including DarkVNC and COBALT STRIKE. Regular industry reporting, including research publications like this one, help mitigate this threat.

ICEDID is known to pack its payloads using custom file formats and a custom encryption scheme. Following our latest ICEDID research that covers the GZip variant execution chain.

In this tutorial, we will introduce these tools by unpacking a recent ICEDID sample starting with downloading a copy of the fake GZip binary:

Analyzing malware can be dangerous to systems and should only be attempted by experienced professionals in a controlled environment, like an isolated virtual machine or analysis sandbox. Malware can be designed to evade detection and infect other systems, so it's important to take all necessary precautions and use specialized tools to protect yourself and your systems.

54d064799115f302a66220b3d0920c1158608a5ba76277666c4ac532b53e855f

Environment setup

For this tutorial, we're using Windows 10 and Python 3.10.

Elastic Security Labs is releasing a set of tools to automate the unpacking process and help analysts and the community respond to ICEDID.

Script	Description	Compatibility
decrypt_file.py	Decrypt ICEDID encrypted file	Windows and others (not tested)
gzip_variant/extract_gzip.py	Extract payloads from ICEDID fake GZip file	Windows and others (not tested)
gzip_variant/extract_payload_from_core.py	Extract and decrypt payloads from the rebuilt ICEDID core binary	Windows and others (not tested)
gzip_variant/load_core.py	Load and execute core custom PE binary	Windows only
gzip_variant/read_configuration.py	Read ICEDID configuration file contained in the fake GZip	Windows and others (not tested)
rebuild_pe.py	Rebuild a PE from ICEDID custom PE file	Windows and others (not tested)

In order to use the tools, clone the Elastic Security Lab release repository and install the nightMARE module.

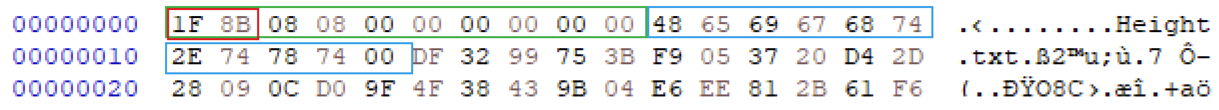
```
git clone https://github.com/elastic/labs-releases
cd labs-release
pip install .\nightMARE\
```

The nightMARE module

All tools in this tutorial use the **nightMARE** module, this library implements different algorithms we need for unpacking the various payloads embedded within ICEDID. We’re releasing nightMARE because it is required for this ICEDID analysis, but stay tuned - more to come as we continue to develop and mature this framework.

Unpacking the fake GZip

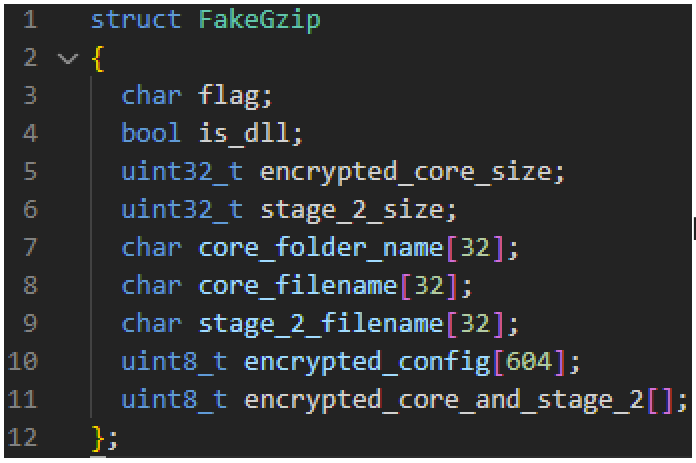
The ICEDID fake GZip is a file that masquerades as a valid GZip file formatted by encapsulating the real data with a GZip header and footer.



```
00000000  1F 8B 08 08 00 00 00 00 00 00 48 65 69 67 68 74  .‹........Height
00000010  2E 74 78 74 00 DF 32 99 75 3B F9 05 37 20 D4 2D  .txt.ß2™u;ù.7 Ô-
00000020  28 09 0C D0 9F 4F 38 43 9B 04 E6 EE 81 2B 61 F6  (..ÐŸO8C›.æî.+aö
```

GZip header and footer
GZip magic bytes appear in red.
The GZip header is rendered in green.
The dummy filename value is blue.

After the GZip header is the true data structure, which we describe below.



```
struct FakeGzip
{
    char flag;
    bool is_dll;
    uint32_t encrypted_core_size;
    uint32_t stage_2_size;
    char core_folder_name[32];
    char core_filename[32];
    char stage_2_filename[32];
    uint8_t encrypted_config[604];
    uint8_t encrypted_core_and_stage_2[];
};
```

FakeGzip data structure

We will use the **labs-releases\tools\icedid\gzip-variant\extract_gzip.py** script to unpack this fraudulent GZip.

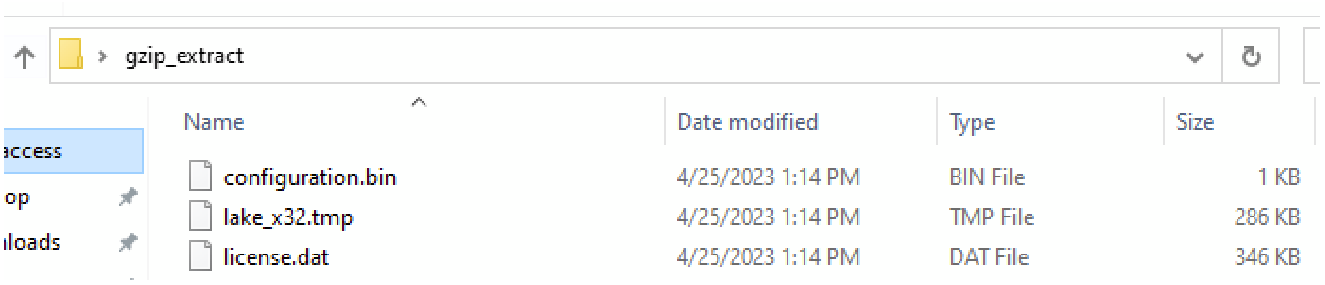
```
usage: extract_gzip.py [--help] input output

positional arguments:
  input       Input file
  output      Output directory

options:
  -h, --help  show this help message and exit
```

We'll use extract_gzip.py on the ICEDID sample linked above and store the contents into a folder we created called “**extract**” (you can use any existing output folder).

```
python extract_gzip.py
54d064799115f302a66220b3d0920c1158608a5ba76277666c4ac532b53e855f extract

============================================================
Fake Gzip
============================================================
is_dll: True
core: UponBetter/license.dat (354282 bytes)
stage_2: lake_x32.tmp (292352 bytes)

extract\configuration.bin
extract\license.dat
extract\lake_x32.tmpRead more
```

This script returns three individual files consisting of:

- The encrypted configuration file: **configuration.bin**
- The encrypted core binary: **license.dat**
- The persistence loader: **lake_x32.tmp**



Name	Date modified	Type	Size
configuration.bin	4/25/2023 1:14 PM	BIN File	1 KB
lake_x32.tmp	4/25/2023 1:14 PM	TMP File	286 KB
license.dat	4/25/2023 1:14 PM	DAT File	346 KB

Files extracted from the fake GZip

Decrypting the core binary and configuration files

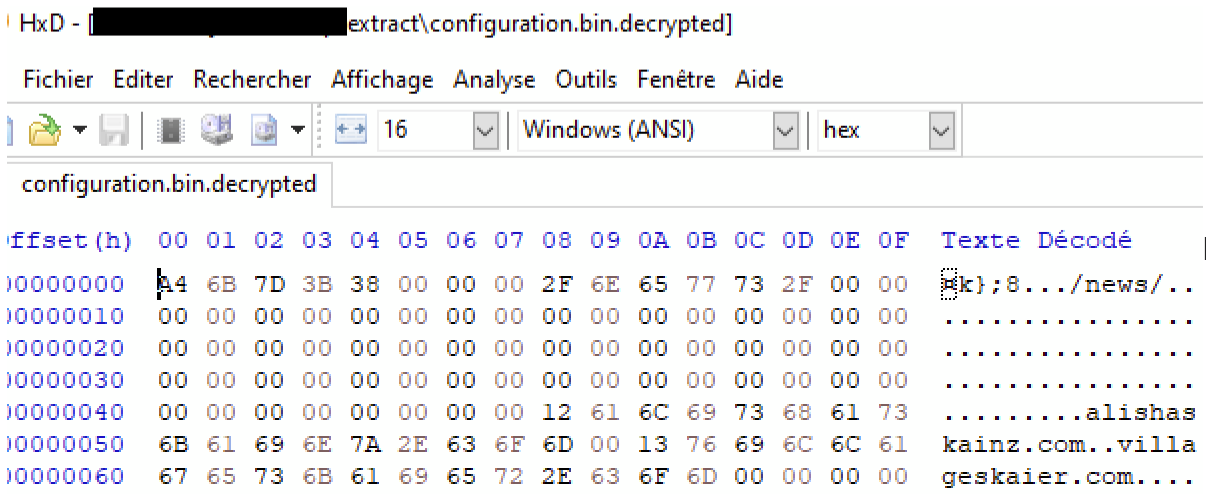
The configuration and the core binary we extracted are encrypted using ICEDID’s custom encryption scheme. We can decrypt them with the **labs-releases\tools\icedid\decrypt_file.py** script.

```
usage: decompress_file.py [--help] input output

positional arguments:
  input       Input file
  output      Output file

options:
  -h, --help  show this help message and exit
```

As depicted here (note that decrypted files can be written to any valid destination):

```
python .\decrypt_file.py .\extract\license.dat .\extract\license.dat.decrypted

python .\decrypt_file.py .\extract\configuration.bin
.\extract\configuration.bin.decrypted
```

The core binary and the configuration are now ready to be processed by additional tools. See the data from the decrypted configuration presented in the following screenshot:



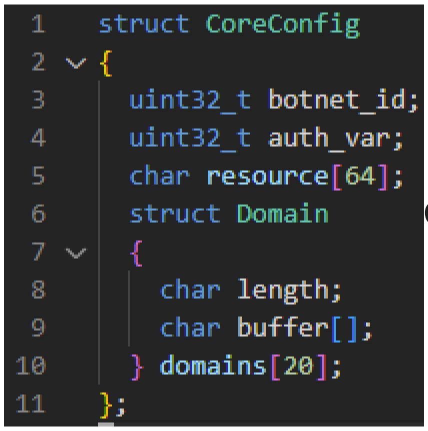
HxD - [[redacted]extract\configuration.bin.decrypted]
Fichier Editer Rechercher Affichage Analyse Outils Fenêtre Aide
16
Windows (ANSI)
hex
configuration.bin.decrypted

```
ffset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Texte Décodé
0000000  A4 6B 7D 3B 38 00 00 00 2F 6E 65 77 73 2F 00 00  ¤k};8.../news/..
0000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000040  00 00 00 00 00 00 00 00 12 61 6C 69 73 68 61 73  .........alishas
0000050  6B 61 69 6E 7A 2E 63 6F 6D 00 13 76 69 6C 6C 61  kainz.com..villa
0000060  67 65 73 6B 61 69 65 72 2E 63 6F 6D 00 00 00 00  geskaier.com....
```

Hex view of the decrypted configuration file

Reading the configuration

The configuration file format is presented below.



```
struct CoreConfig
{
    uint32_t botnet_id;
    uint32_t auth_var;
    char resource[64];
    struct Domain
    {
        char length;
        char buffer[];
    } domains[20];
};
```

Configuration file

The configuration can be read using the **labs-releases\tools\icedid\gzip-variant\read_configuration.py** script.

```
usage: read_configuration.py [--help] input

positional arguments:
  input       Input file

options:
  -h, --help  show this help message and exit
```

We'll use the **read_configuration.py** script to read the **configuration.bin.decrypted** file we collected in the previous step.

```
python .\gzip-variant\read_configuration.py .\extract\configuration.bin.decrypted

============================================================
Configuration
============================================================
botnet_id: 0x3B7D6BA4
auth_var: 0x00000038
uri: /news/
domains:
        alishaskainz.com
        villageskaier.comRead more
```

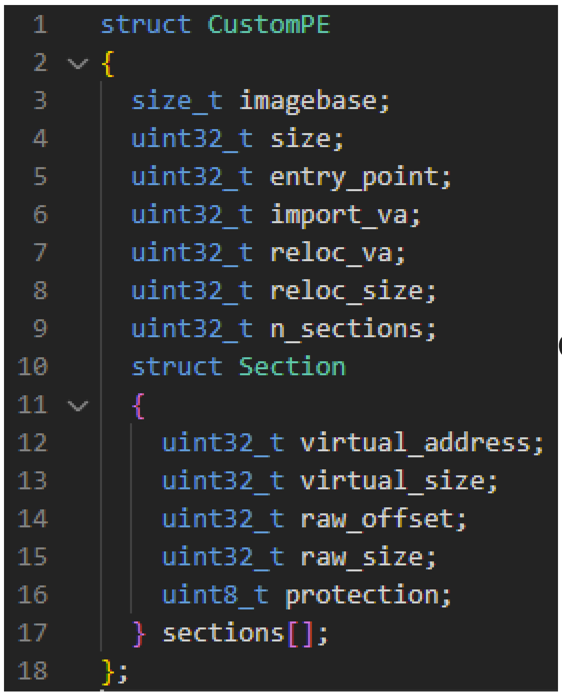
This configuration contains two C2 domains:

- alishaskainz[.]com
- villageskaier[.]com

For this sample, the beaconing URI that ICEDID uses is "**/news/**".

Rebuilding the core binary for static analysis

ICEDID uses a custom PE format to obfuscate its payloads thus defeating static or dynamic analysis tools that expect to deal with a normal Windows executable. The custom PE file format is described below.


```
struct CustomPE
{
    size_t imagebase;
    uint32_t size;
    uint32_t entry_point;
    uint32_t import_va;
    uint32_t reloc_va;
    uint32_t reloc_size;
    uint32_t n_sections;
    struct Section
    {
        uint32_t virtual_address;
        uint32_t virtual_size;
        uint32_t raw_offset;
        uint32_t raw_size;
        uint8_t protection;
    } sections[];
};
```

Custom PE file format

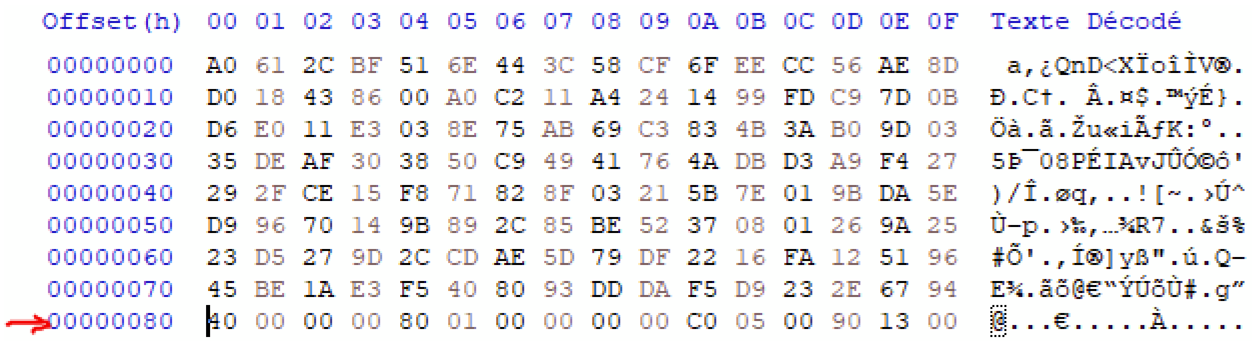
If we want to analyze the core binary, for example with IDA Pro, we need to rebuild it into a valid PE. We use the **labs-releases\tools\icedid\rebuild_pe.py** script.

```
usage: rebuild_pe.py [--help] [-o OFFSET] input output

positional arguments:
  input                 Input file
  output                Output reconstructed PE

options:
  -h, --help            show this help message and exit
  -o OFFSET, --offset OFFSET
                        Offset to real data, skip possible garbage
```

However, when attempting to use **rebuild_pe.py** on the decrypted core binary, **license.dat.decrypted**, we receive the following error message:

```
python .\rebuild_pe.py .\extract\license.dat.decrypted .\extract\core.bin
Traceback (most recent call last):
  File "rebuild_pe.py", line 32, in <module>
    main()
  File "rebuild_pe.py", line 28, in main
    custom_pe.CustomPE(data).to_pe().write(args.output)
  File "nightmare\malware\icedid\custom_pe.py", line 86, in __init__
    raise RuntimeError("Failed to parse custom pe")
RuntimeError: Failed to parse custom pe
```

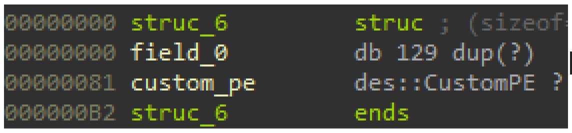
The subtlety here is that the custom PE data doesn't always start at the beginning of the file. In this case, for example, if we open the file in a hexadecimal editor like HxD we can observe a certain amount of garbage bytes before the actual data.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Texte Décodé

00000000  A0 61 2C BF 51 6E 44 3C 58 CF 6F EE CC 56 AE 8D   a,¿QnD<XÏoîÌV®.
00000010  D0 18 43 86 00 A0 C2 11 A4 24 14 99 FD C9 7D 0B  Ð.C†. Â.¤$.™ýÉ}.
00000020  D6 E0 11 E3 03 8E 75 AB 69 C3 83 4B 3A B0 9D 03  Öà.ã.Žu«iÃƒK:°..
00000030  35 DE AF 30 38 50 C9 49 41 76 4A DB D3 A9 F4 27  5Þ¯08PÉIAvJÛÓ©ô'
00000040  29 2F CE 15 F8 71 82 8F 03 21 5B 7E 01 9B DA 5E  )/Î.øq‚..![~.›Ú^
00000050  D9 96 70 14 9B 89 2C 85 BE 52 37 08 01 26 9A 25  Ù–p.›‰,…¾R7..&š%
00000060  23 D5 27 9D 2C CD AE 5D 79 DF 22 16 FA 12 51 96  #Õ'.,Í®]yß".ú.Q–
00000070  45 BE 1A E3 F5 40 80 93 DD DA F5 D9 23 2E 67 94  E¾.ãõ@€"ÝÚõÙ#.g"
00000080  40 00 00 00 80 01 00 00 00 00 C0 05 00 90 13 00  @...€.....À.....
```

Prepended garbage bytes

We know from our research that the size of the garbage is **129** bytes.



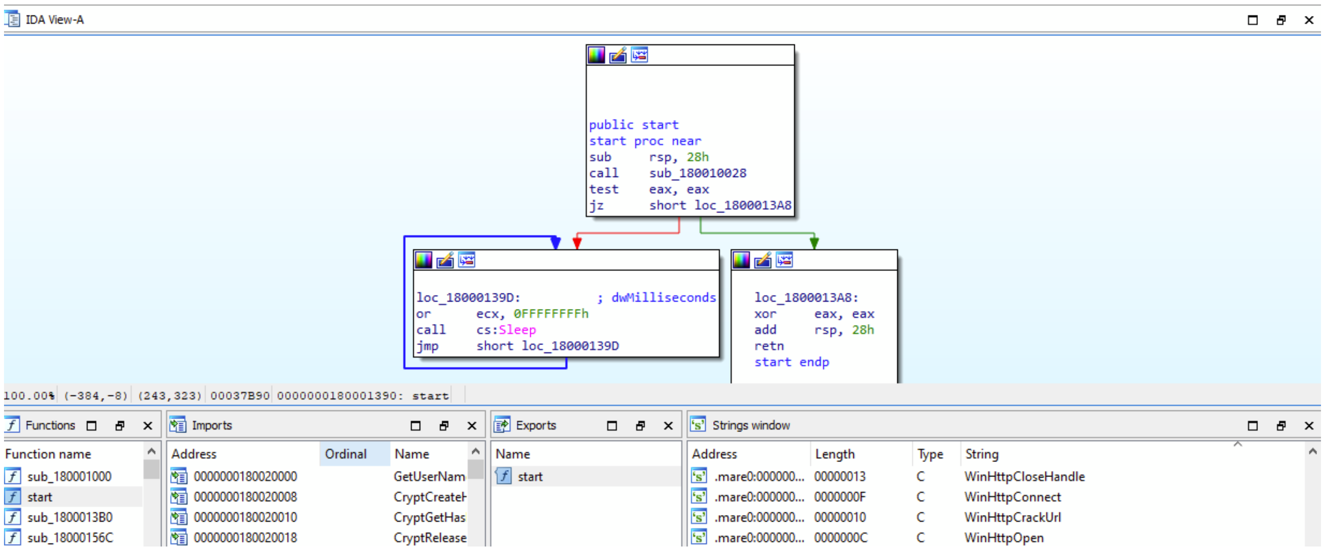
```
00000000 struc_6         struc ; (sizeof
00000000 field_0         db 129 dup(?)
00000081 custom_pe       des::CustomPE ?
000000B2 struc_6         ends
```

Identifying garbage size

With that in mind, we can skip over the garbage bytes and rebuild the core binary using the **rebuild_pe.py** script using the **"-o 129"** parameter. This time we, fortunately, receive no error message. **core.bin** will be saved to the output directory, **extract** in our example.

```
python .\rebuild_pe.py .\extract\license.dat.decrypted .\extract\core.bin -o 129
```

The rebuilt PE object is **not** directly executable but you can statically analyze it using your disassembler of choice.



```
public start
start proc near
sub     rsp, 28h
call    sub_180010028
test    eax, eax
jz      short loc_1800013A8

loc_18000139D:                    ; dwMilliseconds
or      ecx, 0FFFFFFFFh
call    cs:Sleep
jmp     short loc_18000139D

loc_1800013A8:
xor     eax, eax
add     rsp, 28h
retn
start endp

100.00% (-384,-8) (243,323) 00037B90 0000000180001390: start
```

Function name
sub_180001000
start
sub_1800013B0
sub_18000156C

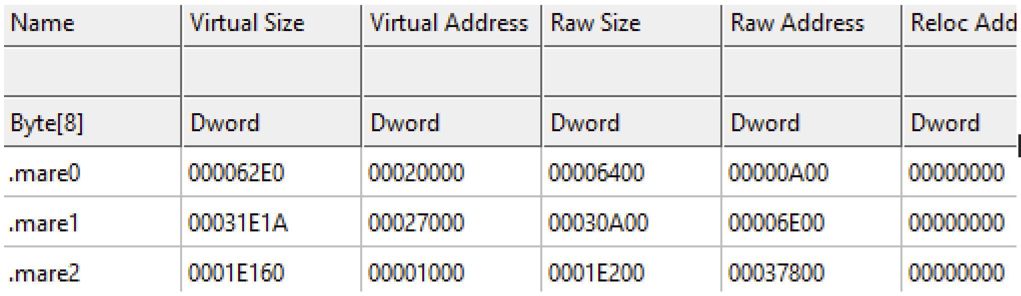
Address	Ordinal	Name
0000000180020000		GetUserNam
0000000180020008		CryptCreateH
0000000180020010		CryptGetHas
0000000180020018		CryptRelease

Name
start

Address	Length	Type	String
.mare0:000000...	00000013	C	WinHttpCloseHandle
.mare0:000000...	0000000F	C	WinHttpConnect
.mare0:000000...	00000010	C	WinHttpCrackUrl
.mare0:000000...	0000000C	C	WinHttpOpen

IDA view of core.bin

We assigned custom names to the rebuilt binary sections (**.mare{0,1,2,...}**).



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Add
Byte[8]	Dword	Dword	Dword	Dword	Dword
.mare0	000062E0	00020000	00006400	00000A00	00000000
.mare1	00031E1A	00027000	00030A00	00006E00	00000000
.mare2	0001E160	00001000	0001E200	00037800	00000000

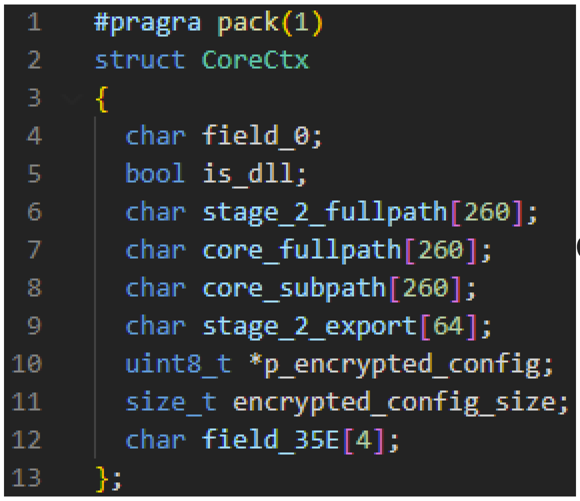
Rebuilt binary section names

We want to credit and thank Hasherezade's work from which we took inspiration to build this tool.

Executing the core binary (Windows only)

The core binary can't be executed without a custom loader that understands ICEDID's custom PE format as well as the entry point function prototype.

From our research, we know that the entry point expects a structure we refer to as the context structure, which contains ICEDID core and persistence loader paths with its encrypted configuration. The context structure is described below.

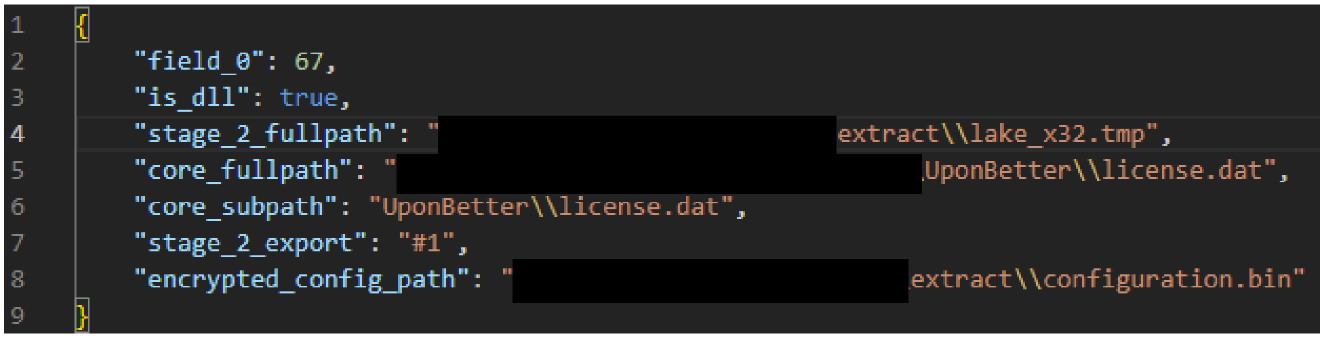


```
#pragra pack(1)
struct CoreCtx
{
    char field_0;
    bool is_dll;
    char stage_2_fullpath[260];
    char core_fullpath[260];
    char core_subpath[260];
    char stage_2_export[64];
    uint8_t *p_encrypted_config;
    size_t encrypted_config_size;
    char field_35E[4];
};
```

Context structure

To natively execute the core binary we use the **labs-releases\tools\icedid\gzip-variant\load_core.py** script, but before using it we need to create the **context.json** file that'll contain all the information needed by this script to build this structure.

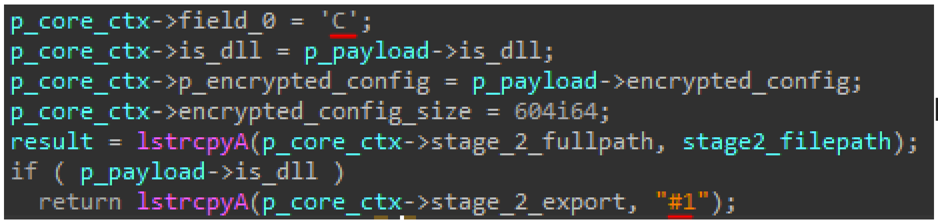
For this sample, we copy the information contained in the fake gzip and we use the path to the encrypted configuration file. We've included an example at **gzip_variant/context.json.example**.



```
{
    "field_0": 67,
    "is_dll": true,
    "stage_2_fullpath": "[redacted]extract\\lake_x32.tmp",
    "core_fullpath": "[redacted]UponBetter\\license.dat",
    "core_subpath": "UponBetter\\license.dat",
    "stage_2_export": "#1",
    "encrypted_config_path": "[redacted]extract\\configuration.bin"
}
```

Example configuration file

Please note that **“field_0”** and **“stage_2_export”** values have to be found while reversing the sample.

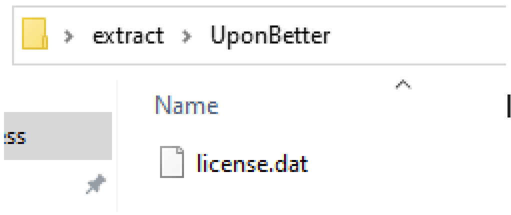


```
p_core_ctx->field_0 = 'C';
p_core_ctx->is_dll = p_payload->is_dll;
p_core_ctx->p_encrypted_config = p_payload->encrypted_config;
p_core_ctx->encrypted_config_size = 604i64;
result = lstrcpyA(p_core_ctx->stage_2_fullpath, stage2_filepath);
if ( p_payload->is_dll )
  return lstrcpyA(p_core_ctx->stage_2_export, "#1");
```

Populating values from previous research

Here we use values from our previous research as placeholders but we have no guarantee that the sample will work 100%. For example, in this sample, we don’t know if the **#1** ordinal export is the actual entry point of the persistence loader.

We also reproduce the first stage behavior by creating the **UponBetter** directory and moving the **license.dat** file into it.



license.dat in the UponBetter directory

We execute the **labs-releases\tools\icedid\gzip_variant\load_core.py** script using the **decrypted core** binary: **license.dat.decrypted**, the **context.json** file.

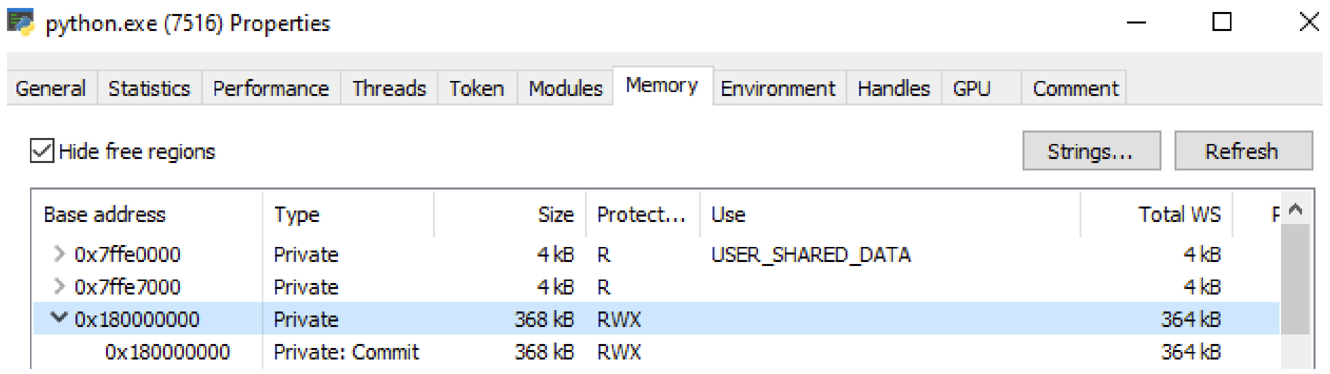
WARNING: The binary is going to be loaded/executed natively by this script, Elastic Security Labs does not take responsibility for any damage to your system. Please execute only within a safe environment.

```
usage: load_core.py [--help] [-o OFFSET] core_path ctx_path

positional arguments:
  core_path             Core custom PE
  ctx_path              Path to json file defining core's context

options:
  -h, --help            show this help message and exit
  -o OFFSET, --offset OFFSET
                        Offset to real data, skip possible garbage
```

Because we have the same garbage bytes problem as stated in the previous section, we use the **"-o 129"** parameter to skip over the garbage bytes.

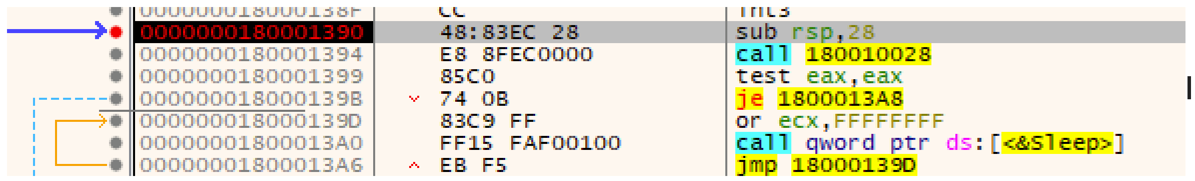
```
python .\gzip-variant\load_core.py .\extract\license.dat.decrypted .\gzip-
variant\context.example.json -o 129

============================================================
Core Loader
============================================================
Base address: 0x180000000
Entrypoint: 0x180001390

Press a key to call entrypoint...
```

When launched, the script will wait for user input before calling the entry point. We can easily attach a debugger to the Python process and set a breakpoint on the ICEDID core entry point (in this example **0x180001390**).



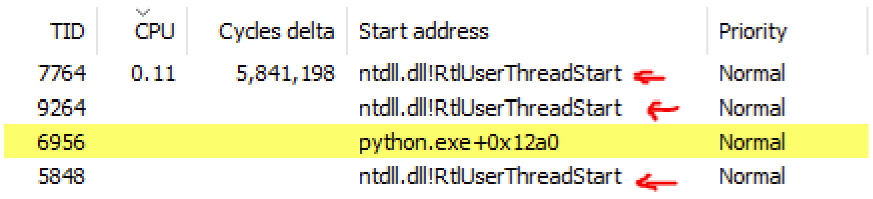
Base address	Type	Size	Protect...	Use	Total WS
0x7ffe0000	Private	4 kB	R	USER_SHARED_DATA	4 kB
0x7ffe7000	Private	4 kB	R		4 kB
0x180000000	Private	368 kB	RWX		364 kB
0x180000000	Private: Commit	368 kB	RWX		364 kB

Breakpoint set on the ICEDID core entry point
Once the key is pressed, we reach the entry point.



```
000000018000138F   CC          int3
0000000180001390   48:83EC 28  sub rsp,28
0000000180001394   E8 8FEC0000 call 180010028
0000000180001399   85C0        test eax,eax
000000018000139B   74 0B       je 1800013A8
000000018000139D   83C9 FF     or ecx,FFFFFFFF
00000001800013A0   FF15 FAF00100 call qword ptr ds:[<&Sleep>]
00000001800013A6   EB F5       jmp 18000139D
```

ICEDID entry point
If we let the binary execute, we see ICEDID threads being created (indicated in the following screenshot).



```
TID    CPU    Cycles delta  Start address                Priority
7764   0.11   5,841,198     ntdll.dll!RtlUserThreadStart Normal
9264                        ntdll.dll!RtlUserThreadStart Normal
6956                        python.exe+0x12a0            Normal
5848                        ntdll.dll!RtlUserThreadStart Normal
```

ICEDID threads being created

Unpacking and rebuilding payloads from the rebuilt core binary

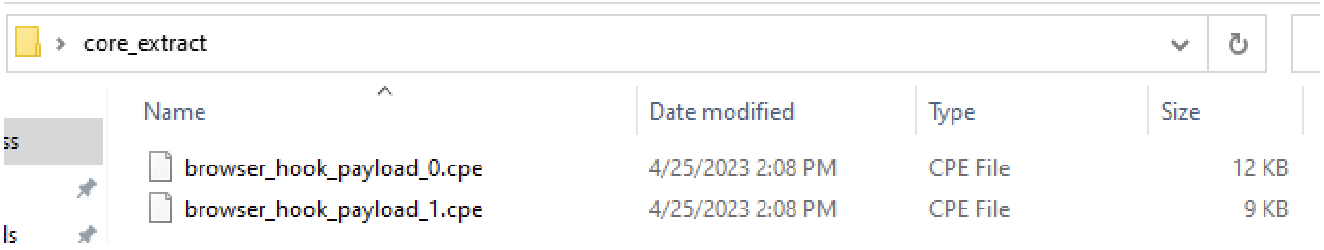
For extracting any of the payloads that are embedded inside the core binary, we will use the **labs-releases\tools\icedid\gzip-variant\extract_payloads_from_core.py** script

```
usage: extract_payloads_from_core.py [--help] input output

positional arguments:
  input       Input file
  output      Output directory

options:
  -h, --help  show this help message and exit
```

We'll use this script on the rebuiltcore binary.

```
python .\gzip-variant\extract_payloads_from_core.py .\extract\core.bin core_extract

core_extract\browser_hook_payload_0.cpe
core_extract\browser_hook_payload_1.cpe
```

From here, we output two binaries corresponding to ICEDID's payloads for web browser hooking capabilities, however, they are still in their custom PE format.

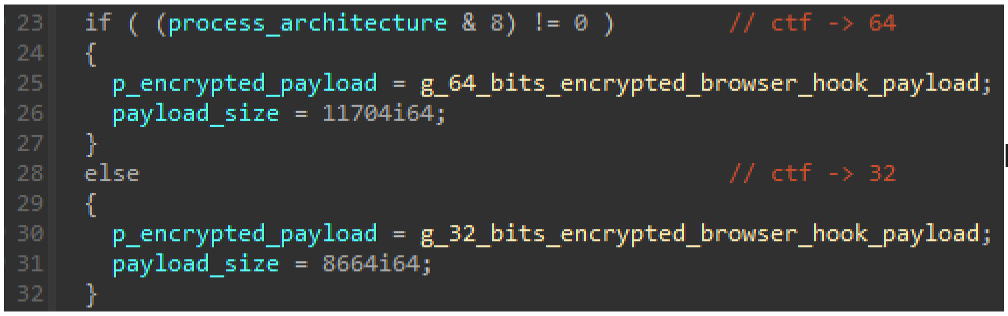


core_extract

Name	Date modified	Type	Size
browser_hook_payload_0.cpe	4/25/2023 2:08 PM	CPE File	12 KB
browser_hook_payload_1.cpe	4/25/2023 2:08 PM	CPE File	9 KB

ICEDID payloads

Based on our research, we know that **browser_hook_payload_0.cpe** is the x64 version of the browser hook payload and **browser_hook_payload_1.cpe** is the x86 version.



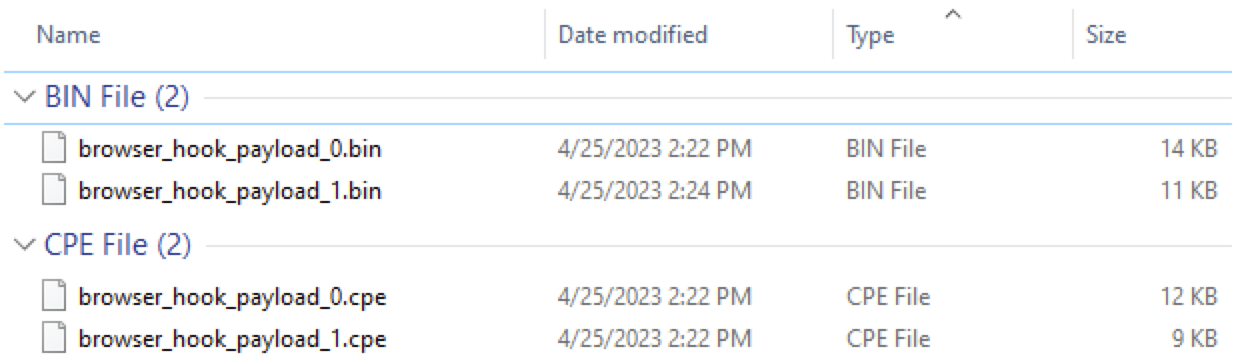
```
if ( (process_architecture & 8) != 0 )        // ctf -> 64
{
  p_encrypted_payload = g_64_bits_encrypted_browser_hook_payload;
  payload_size = 11704i64;
}
else                                          // ctf -> 32
{
  p_encrypted_payload = g_32_bits_encrypted_browser_hook_payload;
  payload_size = 8664i64;
}
```

Browser hook payload architectures

In order to rebuild them, we use the **rebuild_pe.py** script again, this time there are no garbage bytes to skip over.

```
python .\rebuild_pe.py .\core_extract\browser_hook_payload_0.cpe
.\core_extract\browser_hook_payload_0.bin

python .\rebuild_pe.py .\core_extract\browser_hook_payload_1.cpe
.\core_extract\browser_hook_payload_1.bin
```

Now we have two PE binaries (**browser_hook_payload_0.bin** and **browser_hook_payload_1.bin**) we can further analyze.



Name	Date modified	Type	Size
BIN File (2)			
browser_hook_payload_0.bin	4/25/2023 2:22 PM	BIN File	14 KB
browser_hook_payload_1.bin	4/25/2023 2:24 PM	BIN File	11 KB
CPE File (2)			
browser_hook_payload_0.cpe	4/25/2023 2:22 PM	CPE File	12 KB
browser_hook_payload_1.cpe	4/25/2023 2:22 PM	CPE File	9 KB

Payloads for further analysis

Attentive readers may observe that we have skipped the **VNC server** unpacking from the core binary, a decision we made intentionally. We will release it along with other tools in upcoming research, so stay tuned!

Conclusion

In this tutorial we covered ICEDID GZip variant unpacking, starting with the extraction of the fake GZip binary, followed by the reconstruction of the core binary and unpacking its payloads.

ICEDID is constantly evolving, and we are going to continue to monitor major changes and update our tooling along with our research. Feel free to open an issue or send us a message if something is broken or doesn't work as expected.

Elastic Security Labs is a team of dedicated researchers and security engineers focused on disrupting adversaries through the publication of detailed detection logic, protections, and applied threat research.

Follow us on @elasticseclabs and visit our research portal for more resources and research.