IcedID – Technical Analysis of an IcedID Lightweight x64 DLL - 0x0d4y Malware Research

By 0x0d4y

Published: 2024-04-08 · Archived: 2026-04-05 23:33:15 UTC

My first *public malware research* was regarding an *x32 PE stager* (*exe*) from the **IcedID** family. In this research I analyzed three samples from different years, with the aim of identifying code reuse, and developing a **Yara** signature capable of detecting any **IcedID** sample, based on fixed code patterns persistent over the years.

> So you may be asking me: "*why make another one?*".

> Well, I would answer: "*Because my friend **techevo** posted a sample on **MalwareBazaar** of an IcedID **x64 DLL**, and that sample didn't match the signature I developed in my research!*"

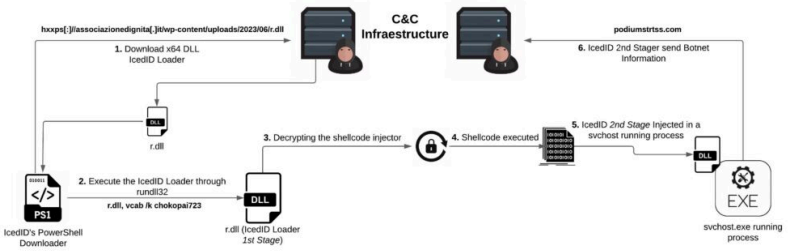
Well, because of that, I previously analyzed the sample, to understand what was different between the samples I analyzed previously, and this one. And it was so fun, that I decided to publish another article showing another face of *IcedID*.

Therefore, in this article I will focus on the following aspects of analysis:

- *Loader Reverse Engineering*, to understand how this *x64 DLL* version of *IcedID* is loaded into memory;
- Reverse Engineering of the *x64 IcedID DLL*;
- Understand what symptoms infected systems may present, so that *Threat Hunters* and *Incident Handlers* can know how to identify such behavior;
- Develop ***Yara*** and ***Sigma*** Detection Rules (if possible).

Execution Flow Summary: From Loader to IcedID

Below is an illustrated and summarized way of how this *IcedID* sample infects the victim system.



The great ***techevo***, posted a *trilogy of research*, in which he has been analyzing the infection chain of this version of the *IcedID* campaign. This research is about the *2nd Stage DLL* that is injected into a *svchost.exe* process running. In this research we will analyze what it does, how it does it, and how we can detect it from the execution of the loader (*r.dll*).

Reverse Engineering: IcedID DLL version of x64

This *x64* version of *IcedID* (*unlike the IcedID I reviewed previously*) is very lightweight and contains few instructions. So light and small that it allows us to take a printout of its entire function call tree. Therefore, we will go through each function, but detailing only those that contain important features.

The body of the main function allows us to have a general idea of the features implemented in this version of *IcedID*. Below you can see a *decryption function*, followed by functions relating to configuring communication with the *C&C servers*, and a possible drop feature from a possible *3rd Stager*.

> NOTE
>
> *It is worth remembering that most of the variable and function names were renamed by me, with the aim of improving understanding of the analysis.*

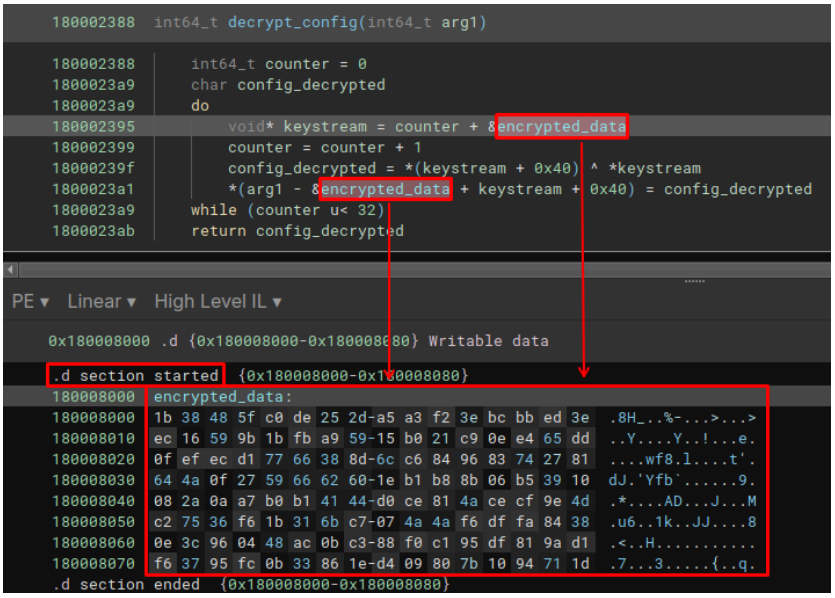
```
int64_t main()

    int64_t rbx = 0
    int64_t comparator = 4
    int64_t counter
    do
        void tsc
        int16_t temp0_1
        temp0_1, temp0_1 = _rdtsc(tsc)
        int32_t temp1
        rbx = rbx << 0x10 | zx.q(temp0_1 | (zx.q(temp1) << 0x20).w)
        Sleep(dwMilliseconds: rbx.d & 0xf)
        counter = comparator
        comparator = comparator - 1
    while (counter != 1)
    void _%016IX
    wsprintfW(param0: &_%016IX, param1: u"%016IX", rbx)
    void config_decrypted
    decrypt_config(&config_decrypted)
    int32_t var_48
    PWSTR botnet_info = botnet_info_struct(var_48, (comparator + 1).d, &_%016IX)
    void var_44
    if (botnet_info != 0 && conditionals_to_c2connection(&var_44, botnet_info, &lpMem_1, &arg_10) != 0)
        int64_t rdx_5 = arg_10
        if (rdx_5 u>= 0x400)
            int32_t error_code = create_write_file_mem_alloc(lpMem_1, rdx_5)
            void* lpMem = lpMem_1
            if (lpMem != 0)
                HeapFree(hHeap: GetProcessHeap(), dwFlags: HEAP_NONE, lpMem)
            int64_t rax_5 = func_char_manipulation(&_%016IX, error_code)
            if (rax_5 != 0)
                wrapper_wrapper_http_connection(&var_44, rax_5)
    return 0
```

The first function we will look at will be a decryption function. This function is essential for the *Yara* signature development process, and for the development of the configuration extractor. Let's analyze it.

Configuration Decryption

It is interesting to see, that *IcedID* has changed its way of decrypting configuration. In my previous research, *IcedID* implemented the **RC4** algorithm, to decrypt the configuration in which the key and data were found, in the ***.data*** section.

In the image below, you can see that the data location section was changed to the .d section, in addition to a relatively simple custom decryption algorithm being implemented.



Simply put, the algorithm is executed *32 times* in a *do while* loop. While this *do while* loop executes, the ***counter*** will be incremented and its value will be used to indicate the position where the ***keystream*** should concatenate a slice of data, with the entire block of data.

The **XOR** operation will then select the first byte of the *keystream* and the byte at the address **keystream[0x40]**, an **XOR** operation is performed on these two bytes, where the result of this operation will be the *decrypted byte*.

The decrypted byte will be placed in a specific position. Thus, in the end, all necessary bytes will be decrypted within the *while* loop, *32 times*.

Below is my Python implementation of the algorithm identified above.

```
def decrypt(data, key):

 counter = 0
 output = bytearray(len(data))
 while counter < 0x20:
   keystream = data[counter:] + key
   byte_decrypted = keystream[0x40] ^ keystream[0]
   output[counter + 0x40 - len(key)] = byte_decrypted
   counter += 1
 return output

data = bytearray.fromhex("1b38485fc0de252da5a3f23ebcbbed3eec16599b1bfba95915b021c90ee465dd0fefecd17766388d6cc6849683742781
key = bytearray.fromhex("1b38485fc0de252da5a3f23ebcbbed3eec16599b1bfba95915b021c90ee465dd0fefecd17766388d6cc6849683742781

decrypted_data = decrypt(data, key)

print("\nConfig Decrypted in Hex:", decrypted_data.hex())
print("Config Decrypted in ASCII:", decrypted_data.decode("ascii", errors="ignore"))
```

And below, the output generated was the **C&C** hostname of this *IcedID* campaign.

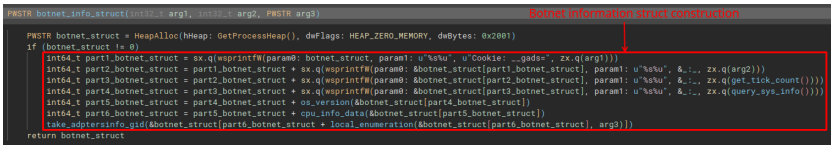
```
 Config Decrypted in Hex: 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
Config Decrypted in ASCII: Bpodiumstrtss.comk?
```

Let's move on to the next function to be executed in the *main* function, **botnet_info_struct**.

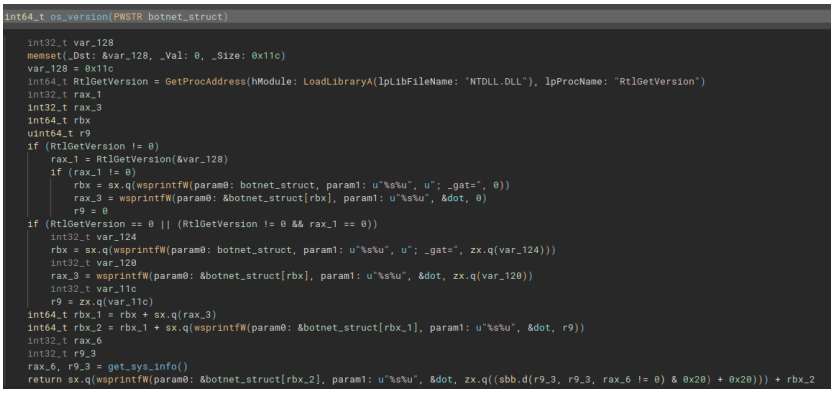
IcedID's Botnet Information Struct

This function is interesting, as it involves collecting information from the user and the infected system, to build the *Cookies* header, which will be sent to the **C&C** server. Below we can see its code structure in *Assembly*.

Through pseudo code, we can have a better understanding of the position of each information, starting with the **__gads** parameter. This parameter will contain the *campaign ID* and previous information about the system.



Analyzing the **os_version** function, we are presented with the collection of the version and build of the infected Windows separated by a dot, which will be concatenated to the **_gat** parameter.



In the following function, we are presented with the **take_cpu_info** function. This function collects information from the infected system's CPU, and concatenates it into the **_ga** parameter. Information is collected by executing the **take_cpu_info** function, which implements the execution of **_cpuid** and **_rdtsc**. Both pieces of information are separated by a period.

```
int64_t cpu_info_data(PWSTR arg1)

    int32_t _cpuid_rdtsc_return
    take_cpu_info(&_cpuid_rdtsc_return)
    int64_t part1_cpu_info = sx.q(wsprintfW(param0: arg1, param1: u"%s%u", u"; _ga=", zx.q(_cpuid_rdtsc_return)))
    int32_t var_14
    int64_t part2_cpu_info = part1_cpu_info + sx.q(wsprintfW(param0: &arg1[part1_cpu_info], param1: u"%s%u", &dot, zx.q(var_14)))
    int32_t var_c
    int64_t cpu_info = part2_cpu_info + sx.q(wsprintfW(param0: &arg1[part2_cpu_info], param1: u"%s%u", &dot, zx.q(var_c)))
    int32_t var_10
    return sx.q(wsprintfW(param0: &arg1[cpu_info], param1: u"%s%u", &dot, zx.q(var_10))) + cpu_info
```

Moving on to the next function, we arrive at the **local_enumeration** function. In which, it collects information from the user and the infected device. The **_u** parameter will contain *username* and *hostname* information. In the **__io** parameter, it will contain the user's *SID*.

```
int64_t local_enumeration(int64_t arg1, PWSTR arg2)

    size = 256
    int32_t lpBuffer
    if (GetComputerNameExW(NameType: ComputerNameNetBIOS, lpBuffer: &lpBuffer, nSize: &size) == 0)
        lpBuffer = 120
    int64_t header_u= = char_manipulation(arg1, u"; _u=", &lpBuffer)
    size = 256
    if (GetUserNameW(lpBuffer: &lpBuffer, pcbBuffer: &size) == 0)
        lpBuffer = 120
    int64_t computer_name = header_u= + char_manipulation(arg1 + (header_u= << 1), &_:_, &lpBuffer)
    int64_t user_name = computer_name + char_manipulation(arg1 + (computer_name << 1), &_:_, arg2)
    int32_t str_buffer
    int32_t account_information = get_computername_lookup_accountname(5, &str_buffer)
    PWSTR param0 = arg1 + (user_name << 1)
    int64_t user_SID
    if (account_information == 0)
        user_SID = user_name + sx.q(wsprintfW(param0, param1: u"%s%u", u"; __io=", 0))
    else
        int64_t counter = 1
        user_SID = user_name + sx.q(wsprintfW(param0, param1: u"%s%u", u"; __io=", zx.q(str_buffer)))
        for (; counter u< zx.q(account_information); counter = counter + 1)
            user_SID = user_SID + sx.q(wsprintfW(param0: arg1 + (user_SID << 1), param1: u"%s%u", &_, zx.q((&str_buffer)[counter])))
    return user_SID
```

botnet struct construction, with user/pc information

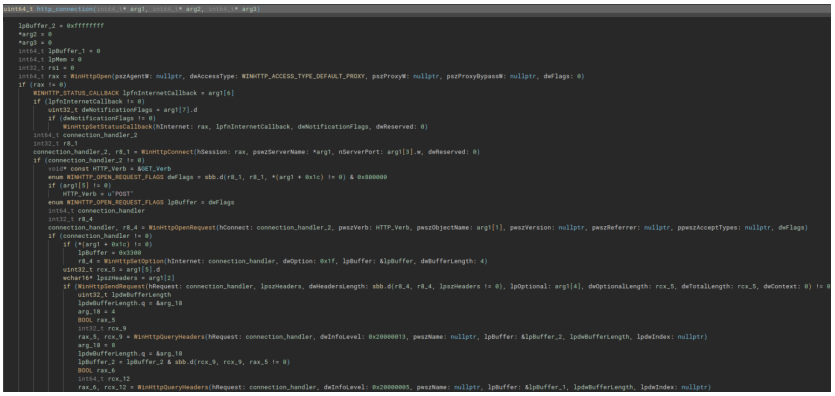
And finally, **take_adptersinfo_gid** is executed, the last function to build the botnet's information structure. This function will collect information from the infected system's network adapter, and will perform manipulation of the *MAC* address information, and concatenate the output into the **_gid** parameter.

C&C Communication Setup

The next function is the configuration and execution of the connection with the **C&C** servers, the **conditionals_to_c2connection**. This function calls a wrapper function that will actually take the code flow to the **WinHTTP** standard API execution function, to execute a connection through the **HTTP** protocol. The return of this function will be conditioned and depending on its content, the code may execute the **process_xor_memory_region** function. In which, it performs exactly what the name indicates.

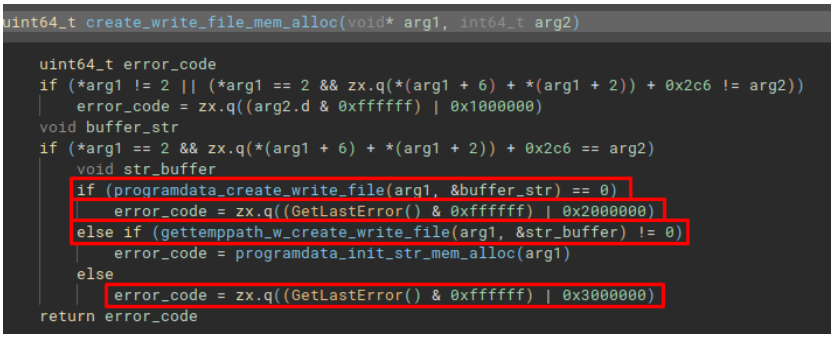
```
int64_t conditionals_to_c2connection(int64_t arg1, int64_t arg2, int64_t* arg3, int64_t* arg4)

    void param0
    wsprintfW(param0: &param0, param1: &_%S, arg1)
    int64_t return_code
    while (true)
        int32_t connection_status_code = wrapper_http_connection(&param0, arg2, arg3, arg4)
        char* lpMem = *arg3
        if (connection_status_code == 404)
            if (lpMem != 0)
                HeapFree(hHeap: GetProcessHeap(), dwFlags: HEAP_NONE, lpMem)
            return_code = 0
            break
        if (connection_status_code == 200 && lpMem != 0 && *arg4 u>= 1024 && *lpMem == 31 && lpMem[1] == 139 && process_xor_memory_region(arg3, arg4) != 0)
            return_code = 1
            break
        if ((connection_status_code != 200 || (connection_status_code == 200 && lpMem != 0)) && lpMem != 0)
            HeapFree(hHeap: GetProcessHeap(), dwFlags: HEAP_NONE, lpMem)
        // 60000 milliseconds == 6 seconds
        Sleep(dwMilliseconds: 60000)
    return return_code
```

The main function has no particularity (**http_connection** function), as its intentions are very clear, just reading the full use of **WinHTTP** functions is enough.

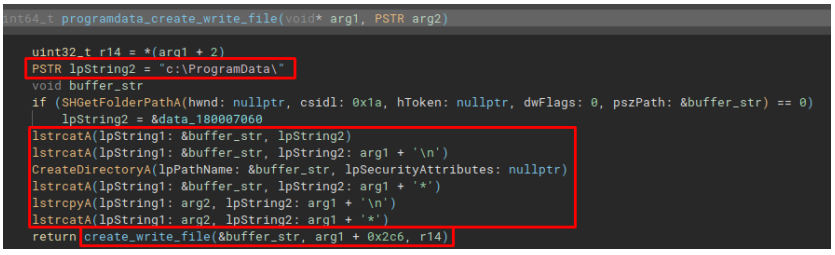


Possible Dropped File's Location – 3nd Stage

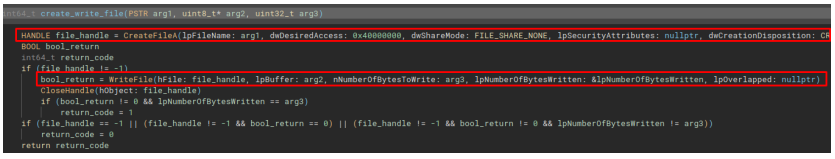
After communicating with the **C&C** servers, **IcedID** demonstrates the ability to drop a possible *3rd Stage* onto disk. It is possible to observe this capacity by analyzing the **create_write_file_mem_alloc** function.



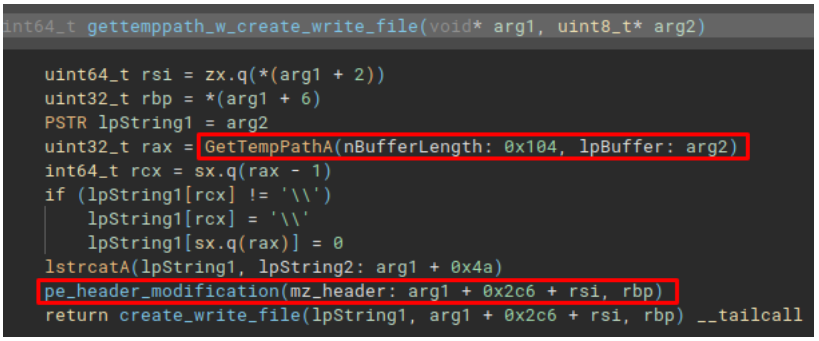
Analyzing the first function to be executed within **create_write_file_mem_alloc**, **programdata_create_write_file** function, we are able to observe that the directory in which the *3rd stage* would be saved would be in the **C:\ProgramData** directory.



Next, the **programdata_create_write_file** function will execute the **create_write_file** function. This is the function that actually creates and writes the *3rd stage* to disk.



Depending on the conditional to be followed in the **create_write_file_mem_alloc** function, the code can follow the flow and execute the **gettemppath_w_create_write_file** function. This function collects the system's default temporary directory via the *GetTempPathA* API and then executes an interesting function.



This function is **pe_header_verification**. This function appears to check whether the *3rd stage* is in fact a ***PE file***, through some calculations the code hopes to obtain the location and mapping of the **MZ** and **PE headers** (mapping occurs in the **mapping_pe** function). Where, if an **MZ** and **PE header** is not identified, the return code will be **0** (*failure*), if the headers can be identified, the return code will be **1** (*success*).

```
int64_t pe_header_verification(int16_t* mz_header, int32_t arg2)

    int64_t result_code
    void* pe_header
    if (*mz_header == 'MZ')
        pe_header = sx.q(*(mz_header + 0x3c)) + mz_header
        if (*pe_header == 'PE')
            mapping_pe(mz_header, pe_header)
            void tsc
            uint32_t temp0_1
            temp0_1, temp0_1 = _rdtsc(tsc)
            int16_t* ptr_mz_header = mz_header
            uint32_t rdx_4 = 0
            int32_t temp1
            *(pe_header + 8) = ((zx.q(temp0_1) | zx.q(temp1) << 0x20) u% 0x1e13380).d + 0x54a48e00
            uint32_t i_1 = (arg2 + 1) u>> 1
            if (i_1 != 0)
                uint32_t i
                do
                    uint32_t rax_3 = zx.d(*ptr_mz_header)
                    ptr_mz_header = &ptr_mz_header[1]
                    uint32_t rdx_5 = rdx_4 + rax_3
                    rdx_4 = zx.d(rdx_5.w) + (rdx_5 u>> 0x10)
                    i = i_1
                    i_1 = i_1 - 1
                while (i != 1)
            uint32_t rdx_7 = zx.d((rdx_4 u>> 0x10).w + rdx_4.w)
            int32_t rcx_8 = *(sx.q(*(mz_header + 0x3c)) + mz_header + 0x58)
            uint32_t rax_8 = zx.d(rcx_8.w)
            uint32_t rcx_9 = rcx_8 u>> 0x10
            int32_t rdx_8 = rdx_7 - adc.d(rax_8, 0, rdx_7 u< rax_8)
            int16_t r10_1
            r10_1.b = rdx_8 u< rcx_9
            rdx_8.w = rdx_8.w - r10_1
            rdx_8.w = rdx_8.w - rcx_9.w
            *(pe_header + 0x58) = zx.d(rdx_8.w) + arg2
            result_code = 1
    if (*mz_header != 'MZ' || (*mz_header == 'MZ' && *pe_header != 'PE'))
        result_code = 0
    return result_code
```

Infected System Behavior Analysis: Threat Hunting and Incident Handler

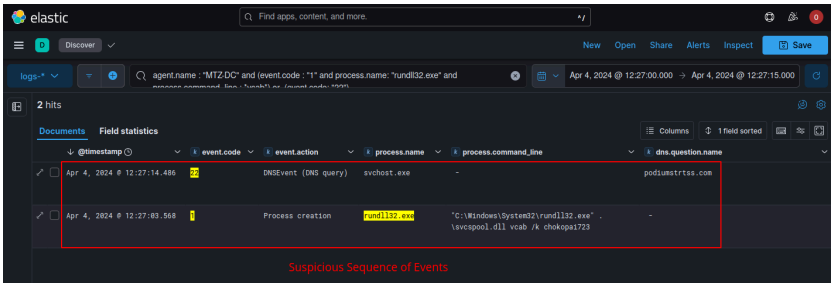
Now that we have analyzed and know the capabilities of this lightweight *x64 DLL* version of **IcedID**, in this section we will analyze what patterns are produced when running it in an environment controlled and monitored by us.

This analysis will produce rich intelligence for *Threat Hunters* and *Incident Handlers*, who will be able to identify the behavior produced by running this version of **IcedID** throughout their operations, in infrastructures monitored by a **SIEM** and receiving logs from **Sysmon**.

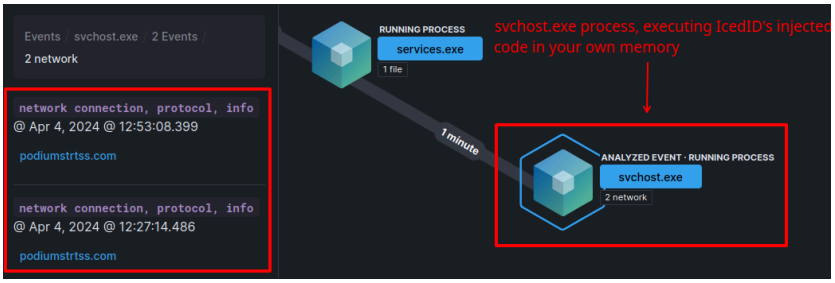
For this analysis to be effective, it is necessary to run the *DLL Loader* (which ***techevo*** analyzed in *part 3 of its triology*) through **rundll32**, executing the **vcab** function with the **/k** argument and the password that is characteristic of each campaign.

When running the *loader*, below it is possible to observe its execution (*with function names and arguments characteristic of this version of **IcedID***) through the **Sysmon Event ID 1** process creation audit log, followed by a DNS resolution to the C&C hostname of this sample (detected by **Sysmon Event ID 22**) through a *running **svchost** process*.

> NOTE
>
> *The next event can be **Sysmon Event ID 22** or **Sysmon Event ID 3**, depending on whether the hostname is still responding to connections.*

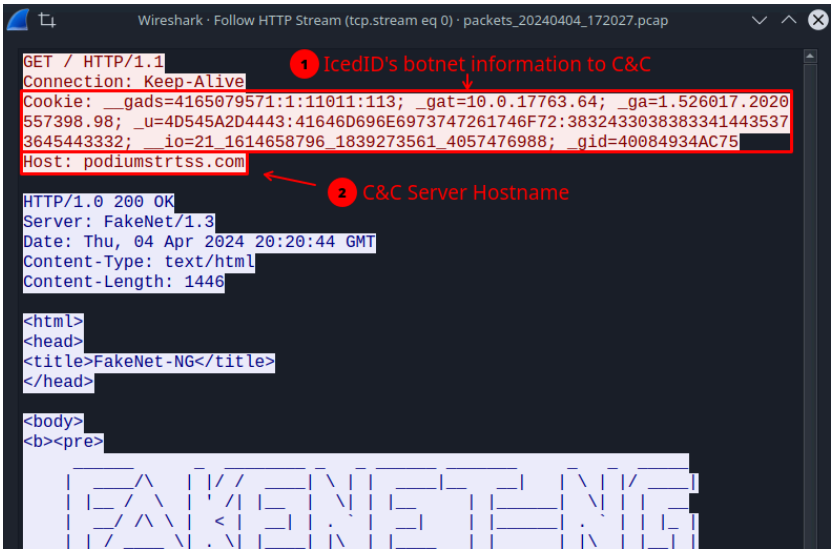


As we know, the loader will decrypt the **IcedID** *DLL* version in memory, and inject it into a running **svchost** process. Therefore, when looking for this behavior in systems through a **SIEM**, the connection to the **C&C** servers will be executed by a benign process (but with malicious code injected) from **svchost**.



If Threat Hunters or Incident Handlers have the opportunity to analyze the execution window traffic, they can also validate the malware family through the characteristics of the communication process with the IcedID C&C.

As we discussed in the reverse engineering section, **IcedID** will collect user and host information, and send it through specific parameters in the *Cookie header*.



To better understand what is being sent to the C&C, below is the list of parameters and assigned information.

```
{

   "__gads" : "botnet_campaing_info",
   "_gat" : "windows_version_build",
   "_ga" : "cpu_info",
   "_u" : "username_hostname_info",
   "__io" : "user_sid",
   "_gid" : "mac_info"

}
```

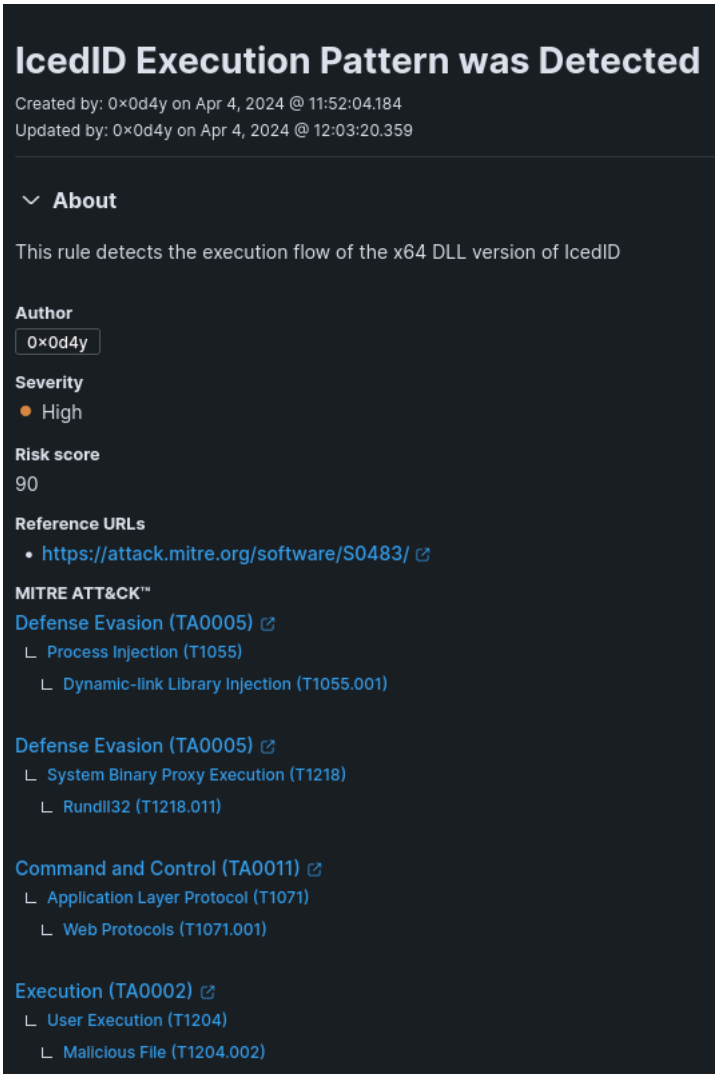
Detection Engineering: SIEM Detection Rule (ELK)

Now that we better understand the symptoms produced by an infection from this strain of **IcedID**, we will be able to produce a detection rule for **SIEM**, with the aim of detecting possible infections originating from this strain of **IcedID**.

Below is a correlation rule in **EQL** syntax, which detects the sequence of observed events characteristic of this version of **IcedID**.

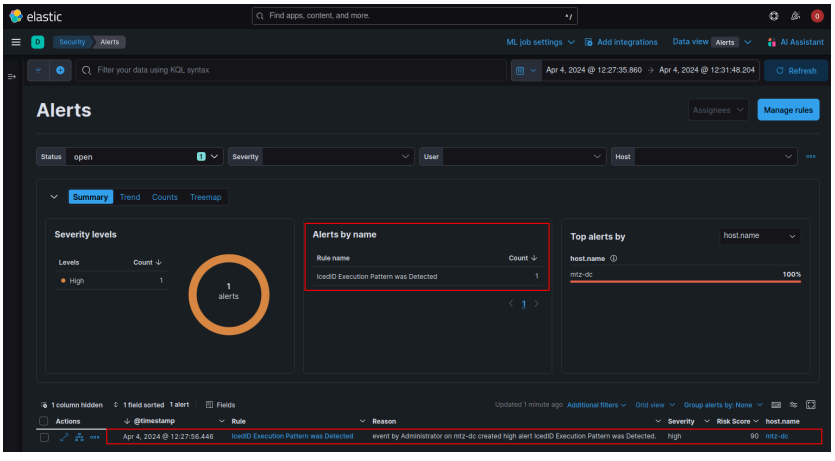
```
 sequence by host.name with maxspan=60s
[any where (event.code : "1" or event.code: "4688") and process.name : "rundll32.exe" and (process.command_line : "*vcab*'
[any where (event.code : "22" or event.code: "3") and process.name: "svchost.exe"]
```

Below is the fully configured detection rule.

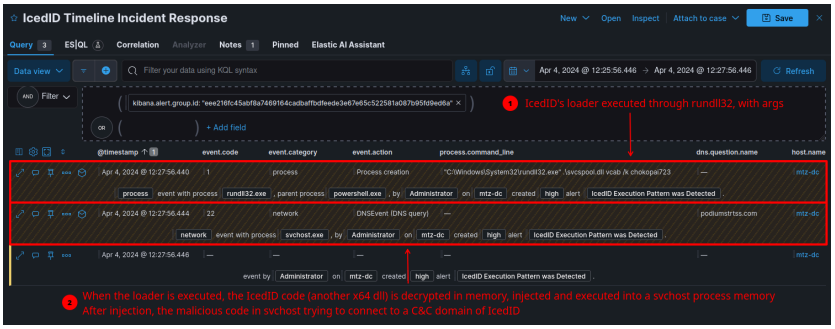


Detection Engineering: SIEM Detection Rule Match (Elastic)

And in order to validate the functionality of this rule, I ran the **IcedID** loader again in my controlled and monitored environment. And as you can see in the image below, the detection rule was effective in detecting the execution of this version of **IcedID**.



Below, we can see in more detail the flow of events that led to the detection rule matching. Having exactly the same pattern of behavior observed previously.



Detection Engineering: Yara Rule

Now that we have analyzed how this version of **IcedID** performs its capabilities, and identified its particularities, we must develop a *Yara* rule to help *Threat Hunters* and *Incident Handlers* detect possible samples of **IcedID** in their infrastructures. *Yara* rules also allow us to identify possible new versions with small modifications, but in which our *Yara* rule still consists of detecting them. If the malware family produces a sample that your *Yara* rule cannot detect, it is because there have been significant changes in its development. And so, you can track changes in malware families over time.

As we analyzed, there are two functions that are very characteristic of this version of **IcedID**, they are:

- The function of the ***Configuration Decryption Algorithm***;
- The function of ***Building the Structure of Botnet Information*** to be sent to C&C.

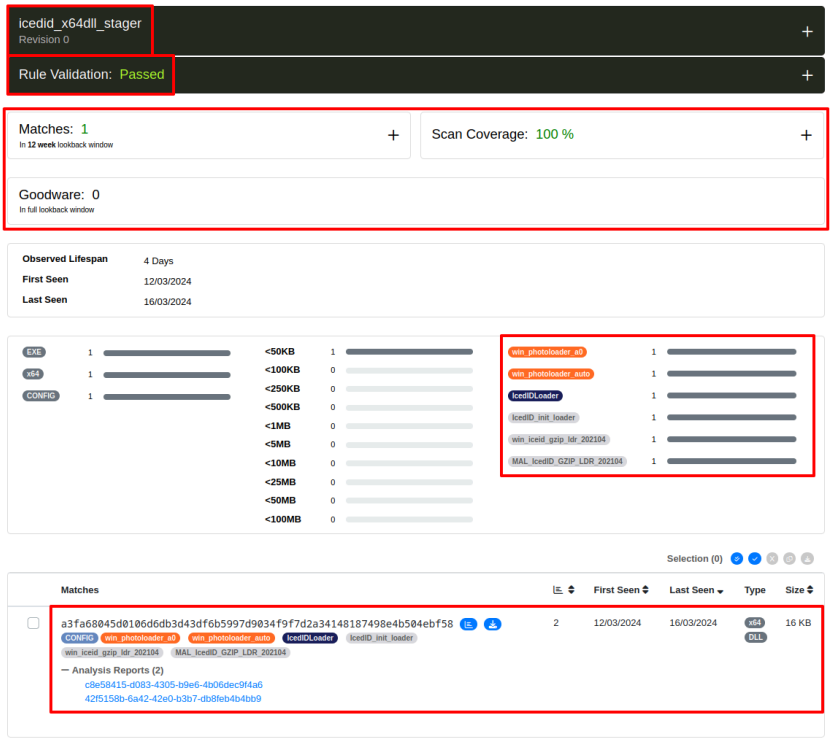
Therefore, Yara rule was based on these two characteristic functions.

```
rule icedid_x64dll_stager {
 meta:
     author = "0x0d4y"
     description = "This rule detects samples from the IcedID family unpacked in memory, identifying code reuse of new co
     date = "2024-04-08"
     score = 100
     reference = "https://0x0d4y.blog/icedid-technical-analysis-of-x64-dll-version/"
     yarahub_reference_md5 = "06cc2fdfd408c15a1e16adfb46e8bb38"
     yarahub_uuid = "5e3bb39f-f9c8-4eb5-8cfd-6812bb27b74a"
     yarahub_license = "CC BY 4.0"
     yarahub_rule_matching_tlp = "TLP:WHITE"
     yarahub_rule_sharing_tlp = "TLP:WHITE"
     malpedia_family = "win.icedid"
   strings:
   $conf_decrypt_algorithm = {
       45 33 C0 ?? ?? ?? ?? ?? ?? ?? 49 2B C9 4B 8D 14 08 49 FF C0 8A 42 40 32 02 88 44 11 40 49 83 F8 20
       }
   $botnet_info_struct_build = {
       44 8B CB 4C 8D 05 ?? ?? ?? ?? 48 8D ?? ?? ?? ?? ?? 48 8B CF FF 15 ?? ?? ?? ?? 48 63 D8 44 8B CD 48 8D 15 ?? ?? ??
   }
   condition:
       uint16(0) == 0x5a4d and
       ($conf_decrypt_algorithm or $botnet_info_struct_build)
}
```

Detection Engineering: Yara Rule Matches

In order to validate the good functionality of the *Yara* rule, I submitted the rule on two platforms that allow scanning your Yara rule on a large set of malware samples (and benign software, to test for false positives). **Unpac.me** and the **Yara Scan Service** that sends the results in *json* format to your email.

Below, we can see the result on the *unpac.me* platform.



And below, we can see the output sent to me, of all the samples that matched my *Yara* rule, through the *Yara Scan Service*.


```
[
  {
    "rule": "icedid_x64dll_stager",
    "malware": "IcedID",
    "sha256": "96e2e63b51500cff2f266e0e4894a667b11d47245837bcb797b8899e04b16b8a",
    "mime_type": "application\/x-msdownload",
    "virustotal_link": "https:\/\/www.virustotal.com\/gui\/file\/96e2e63b51500cff2f266e0e4894a667b11d47245837bcb797b88
    "malwarebazaar_link": "https:\/\/bazaar.abuse.ch\/sample\/96e2e63b51500cff2f266e0e4894a667b11d47245837bcb797b8899e
    "tags": []
  },
  {
    "rule": "icedid_x64dll_stager",
    "malware": "IcedID",
    "sha256": "63770070208c532df8a7d41a391faff7c5280814bebd13b0b935f0fa80fc8e27",
    "mime_type": "application\/x-msdownload",
    "virustotal_link": "https:\/\/www.virustotal.com\/gui\/file\/63770070208c532df8a7d41a391faff7c5280814bebd13b0b935f
    "malwarebazaar_link": "https:\/\/bazaar.abuse.ch\/sample\/63770070208c532df8a7d41a391faff7c5280814bebd13b0b935f0fa
    "tags": []
  },
  {
    "rule": "icedid_x64dll_stager",
    "malware": "IcedID",
    "sha256": "2fa82bdf1836fc7f61d09637580ca0ea26f3aed7e59acad9cd7f793148368214",
    "mime_type": "application\/x-msdownload",
    "virustotal_link": "https:\/\/www.virustotal.com\/gui\/file\/2fa82bdf1836fc7f61d09637580ca0ea26f3aed7e59acad9cd7f7
    "malwarebazaar_link": "https:\/\/bazaar.abuse.ch\/sample\/2fa82bdf1836fc7f61d09637580ca0ea26f3aed7e59acad9cd7f793
    "tags": []
  },
  {
    "rule": "icedid_x64dll_stager",
    "malware": "IcedID",
    "sha256": "42605a1640895ba3d64833f8fc077c074710b142fcb0332607af8560feb64a24",
    "mime_type": "application\/x-msdownload",
    "virustotal_link": "https:\/\/www.virustotal.com\/gui\/file\/42605a1640895ba3d64833f8fc077c074710b142fcb0332607af8
    "malwarebazaar_link": "https:\/\/bazaar.abuse.ch\/sample\/42605a1640895ba3d64833f8fc077c074710b142fcb0332607af856
    "tags": []
  },
  {
    "rule": "icedid_x64dll_stager",
    "malware": "IcedID",
```

```
        "sha256": "8cbd6dee1613f15d998328021a90ecf13b092ea0312555ae4b5627e8f758fe97",
        "mime_type": "application\/x-msdownload",
        "virustotal_link": "https:\/\/www.virustotal.com\/gui\/file\/8cbd6dee1613f15d998328021a90ecf13b092ea0312555ae4b562
        "malwarebazaar_link": "https:\/\/bazaar.abuse.ch\/sample\/8cbd6dee1613f15d998328021a90ecf13b092ea0312555ae4b5627e8
        "tags": []
    },
    {
        "rule": "icedid_x64dll_stager",
        "malware": "IcedID",
        "sha256": "b3063a902d1acc5bdafb98a7976974ea2430b8d62d8aeb414cc3f2fab190dafa",
        "mime_type": "application\/x-msdownload",
        "virustotal_link": "https:\/\/www.virustotal.com\/gui\/file\/b3063a902d1acc5bdafb98a7976974ea2430b8d62d8aeb414cc3f
        "malwarebazaar_link": "https:\/\/bazaar.abuse.ch\/sample\/b3063a902d1acc5bdafb98a7976974ea2430b8d62d8aeb414cc3f2fa
        "tags": []
    },
    {
        "rule": "icedid_x64dll_stager",
        "malware": "UNKNOWN",
        "sha256": "376074f492525537909adb586df6454950e8424665ef9ece63c9ea90979bb238",
        "mime_type": "application\/x-msdownload",
        "virustotal_link": "https:\/\/www.virustotal.com\/gui\/file\/376074f492525537909adb586df6454950e8424665ef9ece63c9e
        "malwarebazaar_link": "https:\/\/bazaar.abuse.ch\/sample\/376074f492525537909adb586df6454950e8424665ef9ece63c9ea90
        "tags": []
    }
]
```


You can go to each malwarebaazar link and check the samples.

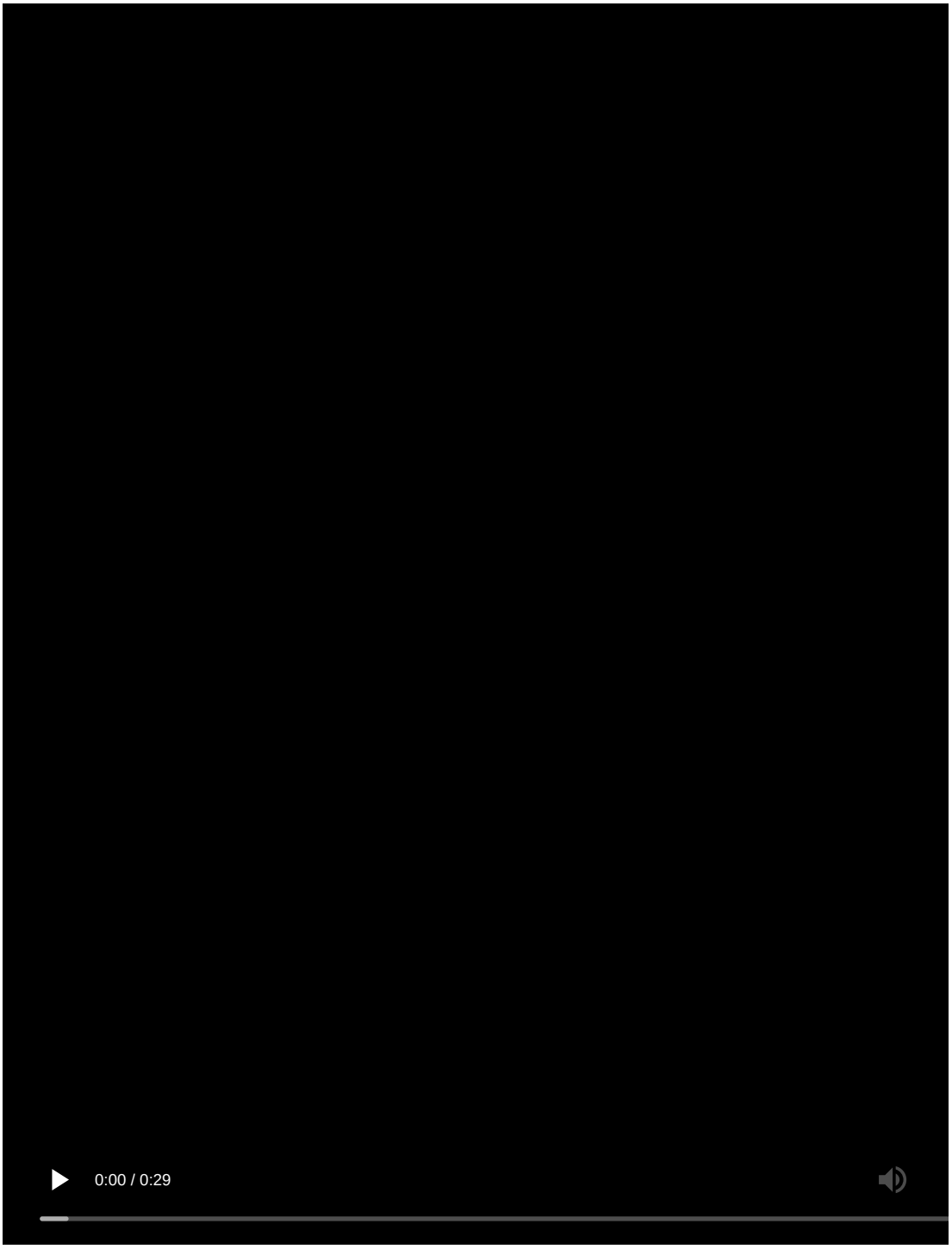
Detection Engineering: Suricata Rule

Because we also analyze the traffic produced by running this version of **IcedID**, we are also able to produce a detection for the pattern of this malicious traffic. Below is the *Suricata* rule that I developed.

```
alert http any any -> any any (msg:"IcedID x64 DLL Traffic was Detected";http.cookie;content:"__gads=";startswith;classty
```

Detection Engineering: Suricata Rule Validation

In order to validate the detection capacity of this signature, below is a *PoC* of the infection process and detection of traffic produced by **IcedID**.



x64 DLL IcedID's Configuration Extractor

And the last output that we can generate to extract as much intelligence as possible from the analysis of this sample, let's move on to creating the *Configuration Extractor*.

We have already analyzed the configuration decryption algorithm for this **IcedID** sample, and we just need to automate the configuration extraction process for the other samples that can be tested. I used my personal project, ***OCEK***, to reuse a Python code that collects the raw data in a certain *PE* section.

Below is the source code of my configuration extractor, for this version of **IcedID**.

```
 import sys
sys.path.append('/home/researcher/Projects/OCEK') # <- My project to automate some config extractor's code
from helpers.get_pe_section import get_pe_section
```

```
def decrypt(data, key):

  counter = 0
  output = bytearray(len(data))
  while counter < 0x20:
    keystream = data[counter:] + key
    byte_decrypted = keystream[0x40] ^ keystream[0]
    output[counter + 0x40 - len(key)] = byte_decrypted
    counter += 1
  return output

filepath = input("\nPut the IcedID x64 DLL filepath: ")
data = input("Put the PE section: ")

payload = get_pe_section(filepath, data)

decrypted_data = decrypt(payload, payload)

print("\nHex Input Decrypted:", decrypted_data.hex())
print("ASCII Output Decrypted:", decrypted_data.decode("ascii", errors="ignore"))
print("")
```

Conclusion

When we reached the end of this research, we were able to analyze the sample statically, run it in a monitored and controlled laboratory, and through this analysis we were able to extract intelligence in different formats:

- Detection Rule for SIEM;
- Yara Detection Rule;
- Suricata Detection Rule.

I hope you, the reader, had fun and/or learned something new. To the next!

References

1. **Technical Evolution's Blog**
2. **Suricata's Blog**
3. **Suricata Rule**
