# API-Hashing in the Sodinokibi/REvil Ransomware – Why and How? – nullteilerfrei

By born

Published: 2019-11-09 · Archived: 2026-04-05 13:59:01 UTC

This post is written for aspiring reverse engineers and will talk about a technique called *API hashing*. The technique is used by malware authors to hinder reverse engineering. We will first discuss the reasons a malware author may even consider using API hashing. Then we will cover the necessary technical details around resolving dynamic imports at load-time and at runtime and finally, will described API hashing and show a Python script that emulates the hashing method used in Sodinokibi/REvil ransomware.

## Top Down vs. Bottom Up

One way to think about different approaches of reverse engineering is *Top Down* vs. *Bottom Up*. Top Down means that you start with the entry point(s) of a binary and follow the execution flow. Bottom Up means that you start at an *interesting location* of the binary and try to understand, if and how the execution flow will reach it. Such a location may for example be an interesting string embedded in the binary ( `http://example.com` or `s3cr3t` for example). So-called *imports* are also a good entry point for a Bottom Up analysis.

One strength of the Bottom Up approach is, that you can *focus* your analysis efforts with very little understanding of the malware or - to phrase it differently - at a very early point in time: You want to know where the juice crypto stuff is happening? Look for calls to the Windows API function `BCryptOpenAlgorithmProvider`. Want to know, where the payload, later dropped to `%TEMP%\evil.exe`, comes from? Look for reference to the string `\evil.exe`. Since this approach is so powerful, malware authors are highly motivated to make it harder for the reverse engineer to use it. One of these counter-measures is called "dynamic API resolutions with name hashing" and described in this blog post.

## API Resolution at Load-Time

Under Windows, imports are listed in the import address table (IAT) of the Portable Executable (PE) file. The table references functions that can be used during execution but are located in an external module. If a program wants to call `PathCanonicalize` in `SHLWAPI.dll` for example, the IAT of your executable contains an entry referencing that function. Say, you are looking for the command & control (C2) server of the malware, then you may want to look at the the functions referencing `WININET.dll` and find the `InternetConnect` entry. A string containing the C2 may be referenced in the the vacinity of calls to this function.

## API Resolution at Runtime

An obvious counter-measure against this reverse engineering approach is, to not use the function pointer from the IAT but resolve it at runtime. For this, the Windows API offers the functions `LoadLibrary` and `GetProcAddress`.

The `LoadLibrary` function accepts the name of a DLL and returns a handle to it. This handle can then be passed to `GetProcAddress` together with a function name to get a pointer to the corresponding function:

```
 #include <stdio.h>
#include <windows.h>

typedef BOOL(*funcType)(LPSTR, LPCSTR);

int main() {
    char buf[MAX_PATH];
    HMODULE hModule = LoadLibrary("SHLWAPI.dll");
    funcType PathCanonicalizeA = (funcType)GetProcAddress(hModule, "PathCanonicalizeA");
    if (! PathCanonicalizeA(buf, "A:\\path\\.\\somewhere\\..\\file.dat")) {
        return 1;
    }
    printf("%s", buf);
    FreeLibrary(hModule);
    return 0;
}
```

Looking at references to `PathCanonicalizeA` in the IAT will not lead to the call seen in the above code listing, defeating the technique described in *Static Imports*. For convenience reasons, one can now store the addresses retrieved by `GetProcAddress` in global variables: If these global variables are named like the actual API functions, one wouldn't even need to change old code.

Switching back to the perspective of a reverse engineer, one can look for the *string* `PathCanonicalizeA` now. It needs to be passed to the `GetProcAddress` function at some point. This, of course, can then be defeated by obfuscating the strings in the binary. But we will explore a different avenue in this post: The below-described method avoids inclusion of function names altogether. Instead, only a fix-sized hash of the function name will be present which has the additional benefit of saving on storage space. This is especially interesting when developing shellcode, where size restrictions may be tough and real.

## API Hashing

Let's again put ourselfs into the shoes of a malware author and let us further assume, we have a list of all exported function names for a given DLL. We can now calculate a hash for each function name. The hash doesn't need be cryptographically secure or even evenly distributed on the codomain. It only needs to be relatively collision-free on the set of all exported function names. It may become clearer soon, what that means exactly.

The following hashing function - which is used in a recent sample of the Sodinokibi/REvil ransomware - initializes a state with `0x2b` and then uses each character of the function name as argument for an arithmetic operation. It performs an arbitrary arithmetic operation in the end and finally returns the resulting state.

```
 #!/usr/bin/env python3
def calc_hash(function_name):
```

```
    ret = 0x2b
    for c in function_name:
        ret = ret * 0x10f + ord(c)
    return (ret ^ 0xafb9) & 0x1fffff
```

The table below lists the resulting hash for a few functions exported from `WINHTTP.dll`. One already may notice that all the hashes are different. In fact, they are pairwise-different for all functions exported from `WINHTTP.dll`, so if we would only consider exports of that DLL and only want to use these eleven functions, the hash can be used to uniquly identify the API function.

| API function name | Hash |
|---|---|
| `WinHttpSendRequest` | `0x1ce2a7` |
| `WinHttpSetOption` | `0x1a5e67` |
| `WinHttpReadData` | `0x064450` |
| `WinHttpCloseHandle` | `0x0fd1fe` |
| `WinHttpOpen` | `0x0a459a` |
| `WinHttpQueryDataAvailable` | `0x0b8299` |
| `WinHttpReceiveResponse` | `0x128d32` |
| `WinHttpConnect` | `0x105ed8` |
| `WinHttpQueryHeaders` | `0x09d98e` |
| `WinHttpOpenRequest` | `0x066635` |
| `WinHttpCrackUrl` | `0x0c17e7` |

You might have guessed, what the overall goal now is: somehow get a list of all API function names together with their addresses, calculate the hashes for each of their names and only use the hash when referring to them. This way, we would avoid inclusion of any string - obfuscated or not - that refers to the API function by name. This works as long as for each function we want to use, the hashing method is collision-free over all considered function names.

## Retrieving API function names

But how do we get a list of all exported functions, potentially for all kinds of DLLs? As already described in the *Static Imports* section, every Portable Executable (PE) file, including DLLs, contain a table with all exported functions. And since all the data from a PE file is loaded into memory, these table is present in memory too. To be precise, the table is the first entry of the `DataDirectory` array of the *Optional Header* of the PE. The Optional Header is a structure of type `IMAGE_OPTIONAL_HEADER` and part of another structure called *PE Header*, which is located at offset `0x18`. The PE Header is of type `IMAGE_NT_HEADERS` and can be found when following the

address at the very end of the *DOS Header* (that is at offset `0x3c`). The DOS Header is of type `IMAGE_DOS_HEADER`. To summarize this rambling the other way around:

- The *DOS Header* is located at the beginning of a PE file.
- The *PE Header* is referenced at position `0x3c` of the *DOS header*.
- The *Optional Header* is part of the PE header at offset `0x18`.
- All *Data Directories* are stored at offset `0x60` of the Optional Header.
- The *Export Directory* is the first Data Directory.

If you prefere code over words, the following pseudo-C-snippet shows, how to retrieve the export directory starting from a pointer to a PE in memory:

```
 IMAGE_DOS_HEADER pe_file = get_ptr_to_memory_containing_pe();
IMAGE_NT_HEADERS pe_header = *(pe_file + 0x3c);
IMAGE_OPTIONAL_HEADER optional_header = pe_header + 0x18;
DATA_DIRECTORY data_directories[16] = optional_header + 0x60;
IMAGE_EXPORT_DIRECTORY export_directory = data_directories[0];
```

**#lifehack**: offsets `0x3c` and `0x78` (which is the sum of the two offsets `0x18 + 0x60` from the listing above) appearing in assembly or decompiled code indicate that PE parsing is going on with the goal to retrieve exports from a PE file.

## Listing exports of DLLs

Let's get our hands dirty: In this section, we will briefly explain, how to easily collect API function names from Windows DLLs. This is useful independently of the malware family one analysis. In the section after, we will look at a concret sample of the Sodinokibi/REvil ransomware. With the help of a Python script, we will calculate API hashes for the collected function names and compare the resulting hashes with the content of a buffer embedded in the malware.

The following Python script accepts a list of DLLs on the command line. Each line of its output is the name of the DLL followed by a space and the name of an export of the DLL.

```
 #!/usr/bin/env python3
import sys
import pefile
import glob

for arg in sys.argv[1:]:
    for file_name in glob.glob(arg):
        try:
            with open(file_name, 'rb') as fp:
                pe = pefile.PE(data=fp.read())
        except pefile.PEFormatError:
            continue
```

```
        if not hasattr(pe, 'DIRECTORY_ENTRY_EXPORT'):
            continue
        export = pe.DIRECTORY_ENTRY_EXPORT
        dll_name = pe.get_string_at_rva(export.struct.Name)
        if not dll_name:
            continue
        if len(export.symbols) == 0:
            continue
        for pe_export in export.symbols:
            if not pe_export.name:
                continue
            print(dll_name.decode('utf-8'), pe_export.name.decode('utf-8'))
```

You can use the script to accumulate exports for as many DLLs as possible. We will collect the result in a file called `exports.txt`.

## API Hashing in Sodinokibi/REvil Ransomware

Let's consider the sample with SHA-256 hash `5f56d5748940e4039053f85978074bde16d64bd5ba97f6f0026ba8172cb29e93`. It belongs to the Sodinokibi/REvil ransomware family and contains a build timestamp of 2019-06-10 15:29:32. Reverse engineering with Ghidra yields the buffer shown below in hex-encoded format.

```
ae91d3b60313b7aca7e29c9ff53a4b505bf85fc37c42ad39da426c2851aa6caeaad50b9f84573f3d142cbcc1c2dbbffc1a8fa8e55e0acfa
```

Right after startup, the malware interprets it as an array of length `0x230` storing a `DWORD` in each entry. Each `DWORD` corresponds to an API function and Sodinokibi/REvil uses API hashing to resolve the corresponding addresses. The Python script listed in the *Appendix* emulates the behaviour of the malware: it reads all exports from `exports.txt`, calculate all hashes for all exports, and list each `DWORD` of the buffer stored in `buffer.bin` together with their API function name:

| DLL Name | API Hash | Function Name |
|---|---|---|
| ADVAPI32.dll | `0x2b7d106b` | CheckTokenMembership |
| ADVAPI32.dll | `0x40b57bbe` | FreeSid |
| ADVAPI32.dll | `0x431d781e` | IsValidSid |
| ADVAPI32.dll | `0x43e878e7` | GetTokenInformation |
| ADVAPI32.dll | `0x45357e2f` | GetUserNameW |
| ADVAPI32.dll | `0x49d572d6` | OpenProcessToken |

| DLL Name | API Hash | Function Name |
|---|---|---|
| ADVAPI32.dll | `0x5b2a6022` | CryptAcquireContextW |
| ADVAPI32.dll | `0x8012bb13` | ImpersonateLoggedOnUser |
| ADVAPI32.dll | `0x8c2cb729` | RegCloseKey |
| ADVAPI32.dll | `0x8f6ab47f` | RevertToSelf |
| ADVAPI32.dll | `0x9c12a701` | RegOpenKeyExW |
| ADVAPI32.dll | `0x9d3ca625` | RegSetValueExW |
| ADVAPI32.dll | `0xc35ff85b` | RegQueryValueExW |
| ADVAPI32.dll | `0xd48aef93` | RegCreateKeyExW |
| ADVAPI32.dll | `0xda1fe106` | AllocateAndInitializeSid |
| ADVAPI32.dll | `0xe46bdf69` | CryptGenRandom |
| combase.dll | `0x39ad427c` | CreateStreamOnHGlobal |
| CRTDLL.dll | `0x958c2a38` | _snwprintf |
| CRYPT32.dll | `0x2dc78a5e` | CryptStringToBinaryW |
| CRYPT32.dll | `0xadcf0a5e` | CryptBinaryToStringW |
| GDI32.dll | `0x3371decc` | DeleteObject |
| GDI32.dll | `0x346dd9cc` | DeleteDC |
| GDI32.dll | `0x4bc6a666` | SetTextColor |
| GDI32.dll | `0x5829b59a` | SetBkColor |
| GDI32.dll | `0x6e5983e6` | SetPixel |
| GDI32.dll | `0x842f699b` | GetDeviceCaps |
| GDI32.dll | `0x8c936138` | CreateFontW |
| GDI32.dll | `0xab3e468f` | GetDIBits |
| GDI32.dll | `0xc1bc2c14` | GetObjectW |
| GDI32.dll | `0xd0803d2a` | SetBkMode |
| GDI32.dll | `0xeb6406c3` | CreateCompatibleBitmap |
| GDI32.dll | `0xec0601b3` | GetStockObject |

| DLL Name | API Hash | Function Name |
|---|---|---|
| GDI32.dll | `0xedc4007f` | SelectObject |
| GDI32.dll | `0xf7bc1a03` | CreateCompatibleDC |
| KERNEL32.dll | `0x08c76270` | MapViewOfFile |
| KERNEL32.dll | `0x0924639c` | DeleteFileW |
| KERNEL32.dll | `0x0b6061c9` | WaitForSingleObject |
| KERNEL32.dll | `0x0f5c65e6` | GetNativeSystemInfo |
| KERNEL32.dll | `0x13dbba0b` | timeBeginPeriod |
| KERNEL32.dll | `0x15fc7f40` | GetFileAttributesW |
| KERNEL32.dll | `0x1b4f71f8` | Process32NextW |
| KERNEL32.dll | `0x1ce07649` | GetVolumeInformationW |
| KERNEL32.dll | `0x286c42da` | CreateToolhelp32Snapshot |
| KERNEL32.dll | `0x2f324589` | UnmapViewOfFile |
| KERNEL32.dll | `0x2ff4455d` | FindClose |
| KERNEL32.dll | `0x32bf580a` | GetCommandLineW |
| KERNEL32.dll | `0x35195fa1` | GetFileSize |
| KERNEL32.dll | `0x3c89563a` | HeapCreate |
| KERNEL32.dll | `0x3d3f5784` | OpenMutexW |
| KERNEL32.dll | `0x40d32a7d` | SetErrorMode |
| KERNEL32.dll | `0x427728cd` | FindNextFileW |
| KERNEL32.dll | `0x43f52945` | CreateFileMappingW |
| KERNEL32.dll | `0x490d23af` | ExitProcess |
| KERNEL32.dll | `0x4d1e27a2` | SystemTimeToFileTime |
| KERNEL32.dll | `0x4f3d2599` | WriteFile |
| KERNEL32.dll | `0x504b3af5` | PostQueuedCompletionStatus |
| KERNEL32.dll | `0x50733aca` | CompareFileTime |
| KERNEL32.dll | `0x588e3220` | GetModuleFileNameW |

| DLL Name | API Hash | Function Name |
|---|---|---|
| KERNEL32.dll | `0x5c6436d6` | CreateMutexW |
| KERNEL32.dll | `0x5fcd3573` | OpenProcess |
| KERNEL32.dll | `0x66d30c7b` | GetDiskFreeSpaceExW |
| KERNEL32.dll | `0x6a0800be` | GetUserDefaultUILanguage |
| KERNEL32.dll | `0x719e1b29` | GetProcessHeap |
| KERNEL32.dll | `0x7f5b15e7` | GetDriveTypeW |
| KERNEL32.dll | `0x8aabe016` | FindFirstFileW |
| KERNEL32.dll | `0x8cabe614` | SetFileAttributesW |
| KERNEL32.dll | `0x8cdbe673` | MultiByteToWideChar |
| KERNEL32.dll | `0x90c6fa75` | Sleep |
| KERNEL32.dll | `0x91f2fb5a` | ReleaseMutex |
| KERNEL32.dll | `0x93b3f91f` | GetComputerNameW |
| KERNEL32.dll | `0x9763fdd3` | Process32FirstW |
| KERNEL32.dll | `0x9a9bf02c` | LocalAlloc |
| KERNEL32.dll | `0x9ad6f07d` | CreateFileW |
| KERNEL32.dll | `0x9c60f6ca` | GetSystemDefaultUILanguage |
| KERNEL32.dll | `0x9e07f4be` | GlobalAlloc |
| KERNEL32.dll | `0xa185cb2c` | CloseHandle |
| KERNEL32.dll | `0xa468cec4` | SetFilePointerEx |
| KERNEL32.dll | `0xaf7fc5dd` | GetSystemDirectoryW |
| KERNEL32.dll | `0xb0b6da10` | TerminateProcess |
| KERNEL32.dll | `0xb780dd38` | GetCurrentProcess |
| KERNEL32.dll | `0xbf26d591` | Wow64RevertWow64FsRedirection |
| KERNEL32.dll | `0xc1ddab7a` | GetProcAddress |
| KERNEL32.dll | `0xc610aca5` | GetQueuedCompletionStatus |
| KERNEL32.dll | `0xc97fa3d5` | LocalFree |

| DLL Name | API Hash | Function Name |
|---|---|---|
| KERNEL32.dll | `0xca02a0b5` | GetCurrentProcessId |
| KERNEL32.dll | `0xca0863c2` | timeGetTime |
| KERNEL32.dll | `0xcb37a181` | MulDiv |
| KERNEL32.dll | `0xcbe9a151` | Wow64DisableWow64FsRedirection |
| KERNEL32.dll | `0xcc3aa698` | CreateThread |
| KERNEL32.dll | `0xcc49a6fa` | GetTempPathW |
| KERNEL32.dll | `0xd0cdba63` | GlobalFree |
| KERNEL32.dll | `0xdb88b122` | GetFileSizeEx |
| KERNEL32.dll | `0xdd54b7ec` | VirtualAlloc |
| KERNEL32.dll | `0xe2e48854` | ReadFile |
| KERNEL32.dll | `0xe36b89db` | WideCharToMultiByte |
| KERNEL32.dll | `0xe5a88f1a` | HeapDestroy |
| KERNEL32.dll | `0xe6c88c71` | GetSystemInfo |
| KERNEL32.dll | `0xe96d83df` | GetFileAttributesExW |
| KERNEL32.dll | `0xeb7281db` | GetWindowsDirectoryW |
| KERNEL32.dll | `0xee8d8436` | MoveFileW |
| KERNEL32.dll | `0xf1989b33` | CreateIoCompletionPort |
| KERNELBASE.dll | `0x380572b8` | PathFindExtensionW |
| MPR.dll | `0x7518713e` | WNetEnumResourceW |
| MPR.dll | `0xae6caa51` | WNetCloseEnum |
| MPR.dll | `0xc258c662` | WNetOpenEnumW |
| ntdll.dll | `0x3822879e` | RtlFreeHeap |
| ntdll.dll | `0x7697c934` | RtlTimeToTimeFields |
| ntdll.dll | `0x8fe93045` | RtlDeleteCriticalSection |
| ntdll.dll | `0x95e62a4e` | NtOpenFile |
| ntdll.dll | `0xacb71303` | RtlGetLastWin32Error |

| DLL Name | API Hash | Function Name |
|---|---|---|
| ntdll.dll | `0xb86307ce` | RtlInitializeCriticalSection |
| ntdll.dll | `0xc09d7f3e` | NtClose |
| ntdll.dll | `0xc97676c4` | RtlEnterCriticalSection |
| ntdll.dll | `0xd2ae6d17` | RtlLeaveCriticalSection |
| ntdll.dll | `0xd69d6931` | RtlAllocateHeap |
| ntdll.dll | `0xe4135ba8` | NtQueryInformationFile |
| ntdll.dll | `0xfac34566` | RtlInitUnicodeString |
| rtm.dll | `0x6acc17e7` | EnumOverTable |
| SHCORE.dll | `0x2b1ca591` | CommandLineToArgvW |
| SHCORE.dll | `0x64472ee8` | SHDeleteValueW |
| SHCORE.dll | `0x9f0bd5aa` | SHDeleteKeyW |
| SHELL32.dll | `0x9cbc123d` | ShellExecuteExW |
| USER32.dll | `0x368a11e7` | GetForegroundWindow |
| USER32.dll | `0x9359b433` | GetDC |
| USER32.dll | `0xb6d391ae` | wsprintfW |
| USER32.dll | `0xbda29ac3` | GetKeyboardLayoutList |
| USER32.dll | `0xd228f54c` | SystemParametersInfoW |
| USER32.dll | `0xec21cb5b` | FillRect |
| USER32.dll | `0xfb28dc52` | DrawTextW |
| USER32.dll | `0xfcbfdbc2` | ReleaseDC |
| WINHTTP.dll | `0x1b146635` | WinHttpOpenRequest |
| WINHTTP.dll | `0x235a5e67` | WinHttpSetOption |
| WINHTTP.dll | `0x23ef5ed8` | WinHttpConnect |
| WINHTTP.dll | `0x38b7459a` | WinHttpOpen |
| WINHTTP.dll | `0x39714450` | WinHttpReadData |
| WINHTTP.dll | `0x9f9ce2a7` | WinHttpSendRequest |

| DLL Name | API Hash | Function Name |
|---|---|---|
| WINHTTP.dll | `0xa4a0d98e` | WinHttpQueryHeaders |
| WINHTTP.dll | `0xacd6d1fe` | WinHttpCloseHandle |
| WINHTTP.dll | `0xf0078d32` | WinHttpReceiveResponse |
| WINHTTP.dll | `0xffb58299` | WinHttpQueryDataAvailable |

This data can now be used to deduce which functions are called and enable a Buttom Up approach again. Looking at the only references to `WinHttpConnect` for example will probably lead to a C2 server.

## Appendix

```
 #!/usr/bin/env python3.7
from revil import calc_hash


def chunks(l, n):
    for i in range(0, len(l), n):
        yield l[i:i + n]


def main():
    exports = []
    with open('exports.txt', 'r') as fp:
        for line in fp:
            sp = line.strip().split(' ')
            if len(sp) != 2:
                continue
            exports.append(sp)

    with open('buffer.bin', 'rb') as fp:
        hash_buffer = fp.read()

    resolutions = {}
    for chunk in chunks(hash_buffer, 4):
        api_hash = int.from_bytes(chunk, byteorder='little')
        for dll_name, export_name in exports:
            calculated_hash = calc_hash(export_name)
            if calculated_hash == ( ( api_hash ^ 0x76c7 ) << 0x10 ^ api_hash ) & 0x1fffff:
                if dll_name not in resolutions.keys():
                    resolutions[dll_name] = []
                resolutions[dll_name].append((api_hash, export_name))
                break
```

```
    for dll_name, pairs in resolutions.items():
        for api_hash, export_name in pairs:
            print(F'{dll_name}\t0x{api_hash:08x}\t{export_name}')


if __name__ == '__main__':
    main()
```

Update (2019-11-22): No mention of the term "static import" now because it doesn't make sense. Instead of "dynamic" vs. "static" imports, the post now talks about imports resolved at "load-time" vs. those resolved at "runtime".

---

Source: https://blag.nullteilerfrei.de/2019/11/09/api-hashing-why-and-how/