19:12 Inside the Emulator of Windows Defender

by Alexei Bulazel

Antivirus emulators are for used dynamic analysis of unknown potentially malicious binaries on endpoint computer systems. As modern malware is often packed, obfuscated, or otherwise transformed to make signature-based classification difficult, emulation is an essential part of any modern antivirus (AV). During emulation, binaries are loaded and run in an emulator which emulates a CPU, an operating system, and a computer environment (settings, files, etc.), among other facilities. Runtime instrumentation allows antivirus software to make heuristic or signature-based determinations about the potential malware it is emulating - the binary may use certain operating system APIs that heuristically indicate malicious intent, or it may unpack or drop a known signed binary. Unfortunately, while AV use of emulators for dynamic analysis is well known, few researchers have published analysis of their inner workings. As it brings together all the challenges and excitement of understanding instruction set architectures, operating system internals, malware behavior, and antivirus itself, emulator analysis is a fascinating topic in reverse engineering.

In this article, I'll share three tricks and anecdotes from my research into Windows Defender Antivirus' emulator. While the term Defender now seems to refer to any security tool or mitigation built into Windows, we'll be looking specifically at the Antivirus product, the first to bear to the Defender name, and a default free install on Windows. The tricks I'll be sharing are Defender specific, but the astute hacker will be able to generalize them to other AVs.

We'll take a look at the mechanisms Defender uses to implement native OS API function emulation, and then present three related reverse engineering tricks: 1) how reverse engineers can establish an output channel to help them observe emulator state from outside of the emulator; 2) how we can bypass Microsoft's attempted mitigations against abuse of the emulator's custom `apicall` instruction; and 3) writing IDA tooling to help us load Defender VDLL binaries that use the `apicall` instruction.

Background

The core of the Windows Defender Antivirus is an enormous 45 thousand function, eleven megabyte library, `mpengine.dll`. Deep within this huge DLL, a proprietary emulator provides facilities for dynamic analysis of potentially malicious Windows PE binaries on the endpoint.

Many AVs are difficult to analyze due to practical hurdles to reverse engineering such as anti-debugging, GUI-only interfaces, custom non-standard binary formats, and enormous disassembler-breaking functions. These challenges are all surmountable (kernel debuggers, custom harnesses, bespoke IDA / Binary Ninja loaders, and additional RAM), but they can be a major impediment to analysis. Joxean Koret has done some tremendous and under-appreciated work on addressing these challenges, interested readers are referred to the Antivirus Hacker's Handbook.

Fortunately, Defender is one of the easiest AVs to analyze that I have encountered - it does run as a Windows Protected Process (so it cannot be debugged by another usermode program), and its binary is massive, but otherwise it is fairly easy to work with. Microsoft's publication of `mpengine.dll` PDBs is also a tremendous help in reverse engineering efforts.

The fact that emulators generally do not provide output other than malware identification makes it difficult to follow their execution without actually debugging them. While previous work on AVLeak from Jeremy Blackthorne, I, and several other collaborators at RPI showed the potential for exploiting malware identification as a side channel to exfiltrate data from within emulators, this technique is slow (generally less than 10 bytes per-second) and only effective for exfiltration of artifacts from within emulators that remains static from execution to execution.[64]

Debugging emulators and setting breakpoints on functions of interest can allow for tracing of program flow. (E.g., is the malware actually getting emulated? Is execution stopping after a particular API call?) Breakpoint-based debugging can get confusing when emulators have complex initialization

[64] For example, some AVs may randomize certain traits of the execution environment with each run. If only a single byte can be extracted with each run, researchers can't extract multi-byte traits.

and teardown routines that invoke functions of interest unrelated to actual malware execution, as is the case with Windows Defender. I would note that I've found code coverage exploration tools, such as a customized version of Markus Gaasedelen's Lighthouse to be extremely helpful in understanding the big picture of emulator execution.[65]

While Defender supports other architectures and binary formats, this article will focus solely on emulator support for 32-bit Windows PE executables. Readers interested in other dynamic analysis facilities in Defender can check out my REcon Brussels 2018 presentation on Defender's JavaScript engine.

On Emulator Architecture

AV emulators are generally constructed from three key components - CPU emulation, operating system emulation, and a virtual environment. Due to performance and legal licensing concerns, CPU and OS emulation are usually wholly proprietary and built on AV-industry developed tooling, not open source projects like QEMU or WINE.

CPU emulators implement a particular instruction set architecture in software, so that binary code can be executed in the emulator. OS emulation is software-based emulation of operating system facilities - allowing malware to make OS API calls as it runs. Finally, emulators must emulate a virtual environment with observable traits such as usernames, files on disk, and registry entries, among many other traits. Other than a handful of traits that are accessible from within a processes actual memory space (e.g., OS build information on the Windows PEB), most of the virtual execution environment can only be observed through OS API calls. (Querying for a username, statting a directory, reading a registry key, etc.) As a result, OS emulation is often tightly coupled with virtual environment emulation.

The three tricks addressed here will all touch upon "VDLLs" (presumably "virtual DLLs") within the Defender emulator. VDLLs emulate the functionality of real Windows DLLs (dynamic-link libraries) in the Defender emulator, providing emulation of the operating system API, including presenting the virtual execution environment. These VDLLs are real Windows PE files, and using them is just like using real Windows DLLs - they are loaded into the memory space of binaries under emulation, they are present in the emulated file system in the right directories, they can be loaded with LoadLibrary, etc. Like real DLLs, they are compiled x86 code, and they run at the same privilege level, with the same stack, registers, and other facilities as the code invoking them - it just happens that this is going on within a virtualized emulated process running on an emulated CPU.

On a real Windows system, some DLL functions may ultimately resolve to triggering system calls where interaction with the kernel is necessary (e.g., when writing a file to disk, opening a network socket, putting the process to sleep, etc.), while others may stay in usermode and simply set return values or transform input. (E.g., grabbing the `IsDebuggerPresent` flag off the PEB, translating a string to uppercase, or performing a `memcpy`.) Similarly, Defender's VDLLs may trap into special natively implemented emulation routines akin to performing system calls, or they may stay executing solely within emulator memory while setting return values or manipulating input.

Lets take a look at the simpler form of VDLL emulated functions - those which stay executing in emulator memory without trapping out to a special kernel syscall-like emulation routine implemented in native code. Figure 5 shows Defender's `kernel32.dll` VDLL emulation of `kernel32!GetComputerNameW`. When a malware binary calls `GetComputerNameW`, this code provides emulation of the function with x86 code that simply runs on the virtual CPU. As we can observe, this routine is hardcoded to return the string "`HAL9TH`" - evidently the developer who wrote this emulation was a fan of Arthur C. Clarke. This particular trait could be used by malware to evade the Defender emulator, e.g., malware seeing the computer name "`HAL9TH`" could choose not to run, knowing that it is likely being emulated by Defender.

Having looked at simple, in-emulator, VDLL routines, we can now look at more complex routines that require invoking native emulation. These routines are akin to those OS API functions which require syscalling in to the kernel. Just like in the kernel, these routines are used to handle more complex operations, such as interacting with the file system, creating threads, or interacting with mutexes or events.

Whereas on a real system the int or syscall instruction and specific register values are used to alert the kernel that it must service some usermode re-

[65] `git clone https://github.com/gaasedelen/lighthouse`

```
.text:7C82D0EA ; ═══════════════ S U B R O U T I N E ═══════════════════════════════════════
.text:7C82D0EA
.text:7C82D0EA ; Attributes: bp-based frame
.text:7C82D0EA
.text:7C82D0EA ; BOOL __stdcall GetComputerNameW(LPWSTR lpBuffer, LPDWORD nSize)
.text:7C82D0EA                 public GetComputerNameW
.text:7C82D0EA GetComputerNameW proc near               ; DATA XREF: .text:off_7C8547D8
.text:7C82D0EA
.text:7C82D0EA lpBuffer        = dword ptr  8
.text:7C82D0EA nSize           = dword ptr  0Ch
.text:7C82D0EA
.text:7C82D0EA                 push    ebp
.text:7C82D0EB                 mov     ebp, esp
.text:7C82D0ED                 mov     eax, [ebp+nSize]
.text:7C82D0F0                 push    edi
.text:7C82D0F1                 test    eax, eax
.text:7C82D0F3                 jz      short loc_7C82D119
.text:7C82D0F5                 mov     edi, [ebp+lpBuffer]
.text:7C82D0F8                 test    edi, edi
.text:7C82D0FA                 jz      short loc_7C82D119
.text:7C82D0FC                 cmp     eax, 1000h
.text:7C82D101                 jbe     short loc_7C82D119
.text:7C82D103                 push    8
.text:7C82D105                 pop     ecx
.text:7C82D106                 cmp     [eax], ecx
.text:7C82D108                 jnb     short loc_7C82D120
.text:7C82D10A                 mov     [eax], ecx
.text:7C82D10C                 mov     eax, large fs:18h
.text:7C82D112                 mov     dword ptr [eax+34h], 6Fh
.text:7C82D119
.text:7C82D119 loc_7C82D119:                           ; CODE XREF: GetComputerNameW+9
.text:7C82D119                                         ; GetComputerNameW+10 ...
.text:7C82D119                 xor     eax, eax
.text:7C82D11B
.text:7C82D11B loc_7C82D11B:                           ; CODE XREF: GetComputerNameW+4B
.text:7C82D11B                 pop     edi
.text:7C82D11C                 pop     ebp
.text:7C82D11D                 retn    8
.text:7C82D120 ; ──────────────────────────────────────────────────────────────────────────
.text:7C82D120
.text:7C82D120 loc_7C82D120:                           ; CODE XREF: GetComputerNameW+1E
.text:7C82D120                 push    esi
.text:7C82D121                 mov     esi, offset aHal9th_0 ; "HAL9TH"
.text:7C82D126                 movsd
.text:7C82D127                 movsd
.text:7C82D128                 movsd
.text:7C82D129                 movsw
.text:7C82D12B                 mov     dword ptr [eax], 7
.text:7C82D131                 xor     eax, eax
.text:7C82D133                 inc     eax
.text:7C82D134                 pop     esi
.text:7C82D135                 jmp     short loc_7C82D11B
.text:7C82D135 GetComputerNameW endp
```

Figure 5. Defender's in-emulator `kernel32.dll` VDLL emulation of `GetComputerNameW`.

quest, in Defender, a custom non-standard `apicall` instruction provides this facility. When the CPU emulator sees the `apicall` instruction, it invokes special native emulation routines to handle emulation of a complex function.

The `apicall` instruction consists of a three byte opcode, `0f ff f0`, followed by a four byte immediate indicating a function to emulate. The four byte immediate value is the CRC32 of the DLL name in all caps xored with the CRC32 of the function's name.

```
0f ff f0          [four byte immediate]
apicall           which routine to emulate
```

These `apicall` functions are spread across Defender's virtual DLLs and used to trigger the more complex emulation certain functions may require. For example, the code below is used to trigger Defender's native emulation of the Sleep. This function with the actual `apicall` instruction is called by `kernel32!SleepEx`, which can be called directly, or by `kernel32!Sleep`, which is basically just a wrapper around `kernel32!SleepEx`. The same is true on a real Windows system.

```
8B FF                   mov     edi, edi
E8 00 00 00 00          call    $+5
83 C4 04                add     esp, 4
0F FF F0 B6 BE 79 57    apicall kernel32!Sleep
50                      push    eax
33 C0                   xor     eax, eax
58                      pop     eax
C2 04 00                retn    4
```

When the virtual CPU emulator sees the custom `apicall` opcode run, it ends up calling out through several functions until it ends up at `__call_api_by_crc(pe_vars_t *v, unsigned int apicrc)`. In this function, `pe_vars_t *v` is an enormous (almost half a megabyte) struct holding all the information needed to manage the emulator's state during emulation. `unsigned int apicrc` is the immediate of the `apicall` instruction, `crc32(dll name in all caps)` $\oplus$ `crc32(name of function)`. From here, the emulator searches the the global `g_syscalls` array for a function pointer that provides native emulation of the CRCed API function. As can be seen in Figure 6, the array is 119 `esyscall_t` structs, each consisting of a function pointer to an API emulation function followed by the corresponding CRC32 value.

These native functions are implemented in Defender's `mpengine.dll` as native x86 code. Like an OS kernel, they have privileged full control over processing being emulated - they can manipulate memory, register state, etc. These functions can also interact with internal data emulator data structures, such as those that store the virtual file system or heuristic information about malware behavior.

It's worth noting that since these 119 emulated functions are emulated with native code, any vulnerabilities in them can allow malware to break out of the emulator, escalate privilege to `NTAUTHORITY/SYSTEM` (which Defender currently runs as, unsandboxed), and gain code execution within an AV process itself - unlikely to be flagged by the AV for any malicious behavior it carries out.

Building files that get consistently emulated during scanning can be a challenge. Through a bit of trial and error, I was able to come up with Visual Studio build settings to produce Windows executables that are consistently scanned - this involved tweaking optimization levels, target OSes, and linking. The Visual Studio project included in this issue gets consistently emulated when I have Defender scan it.[66]

Creating an Output Channel

AV software's usual lack of output can make it particularly obtuse to approach for reverse engineers. When scanning a piece of potential malware, the AV will often respond with a malicious or not malicious classification, but little else. Naming conventions in identifying the malware may provide some indication of how it was scanned. (For example, seeing the identification "`Dropper:[malware name]`" is a strong indication that the malware was run in the AV's emulator, where it dropped a known piece of malware.)

The prior AVLeak research showed how malware identification itself may be exploited as a side channel to leak information out from these emulators, but this approach is generally only useful for AV evasion. (For example, creating malware that looks for particular unique identifiers in these emulated systems in order to know that it is being analyzed so it can then behave benignly.) This approach is

[66] `unzip pocorgtfo19.pdf defender.zip`

```
5A129BA8 ; esyscall_t g_syscalls[119]
5A129BA8 g_syscalls  dd offset ?NTDLL_DLL_NtSetEventWorker@@YAXPAUpe_vars_t@@@Z
5A129BAC             dd 5F2823h
5A129BB0             dd offset ?NTDLL_DLL_NtResumeThreadWorker@@YAXPAUpe_vars_t@@@Z
5A129BB4             dd 2435AE3h
5A129BB8             dd offset ?NTDLL_DLL_NtSetInformationFileWorker@@YAXPAUpe_vars_t@@@Z
5A129BBC             dd 2DA9326h
5A129BC0             dd offset ?ADVAPI32_DLL_RegDeleteValueW@@YAXPAUpe_vars_t@@@Z
5A129BC4             dd 6A61690h
```

Figure 6. Definition of `g_syscalls` consisting of 119 `esyscall_t` structs.

also slow as it extracts information at the rate of bytes per second. Finally, AVLeak requires multiple rounds of malware scanning to extract complex multi-byte artifacts. This is fine for most artifacts of interest, such as usernames, timing measurements, and API call results, but some interesting artifacts may be randomized per run or too long to dump, such as bytes of library code after standard function prologues in Kaspersky AV's emulated DLLs or complete files from disk.

After seeing me present my AVLeak side channel research, my friend Mark suggested using function hooking to create a much larger bandwidth channel from within AV emulators to the outside. By hooking the native code-implemented functions inside the emulator's `g_syscalls` array, and then invoking those hooked functions with malware inside the emulator using arguments we'd like to pass to the outside world, we can effectively create an output channel for sharing information from inside.

In general, this technique requires solving the non-trivial technical challenge of actually locating emulation routines in memory, writing code to hook them, and then figuring out how to extract emulated parameters and potentially memory contents from the emulator. In the case of Windows Defender however, this is relatively easy, as these functions are conveniently labeled by Microsoft provided symbols, and the existing code already present gives us a good example to work off of.

While the in-emulator VDLL emulation functions can simply interact directly with memory inside the emulator, these native emulations functions must use APIs to programmatically change emulator state via the `pe_vars_t *v` parameter which all of them take. We can see an example of this in Figure 7's annotated Hex-Rays decompilation of `kernel32!WinExec`. Note how parameters are pulled out from the current emulation session, and parameter 0 (`LPCSTR lpCmdLine`) is a pointer within the emulator's virtual address space and must be handled through with `pe_read_string_ex` in order to retrieve the actual wide string at the supplied emulator address.

Reversing out how `pe_read_string_ex` and other APIs used to map in parameter-provided pointers, we come across the massive function: `BYTE * __mmap_ex(pe_vars_t *v, unsigned int size, unsigned __int64 addr, unsigned int rights)`, which returns a native pointer to a virtual memory inside an emulation session. Given this pointer, native code can now reach in and read or write (depending on rights) memory inside the emulator.

With our understanding of function emulation and memory management, we now have the tools to create a simple output channel from within the emulator. We begin with a simple function, one that is well suited to serve as an output channel: `kernel32!OutputDebugStringA`. Defender's provided native function of the function basically does nothing, it just retrieves its single parameter and bumps up the emulator tick count:

```
void __cdecl KERNEL32_DLL_OutputDebugStringA
(pe_vars_t *v){
  Parameters<1> arg; // [esp+4h] [ebp-Ch]

  Parameters<1>::Parameters<1>(&arg, v);
  v->m_pDTc->m_vticks64 += 32i64;
}
```

```
/*
Emulation of  UINT WINAPI WinExec( _In_ LPCSTR lpCmdLine, _In_ UINT uCmdShow);
*/
void __cdecl KERNEL32_DLL_WinExec(pe_vars_t *v)
{
  DT_context *pDTc; // ecx
  unsigned __int64 v2; // [esp+0h] [ebp-54h]
  CAutoVticks vticks; // [esp+10h] [ebp-44h]
  src_attribute_t attr; // [esp+1Ch] [ebp-38h]
  unsigned int Length; // [esp+30h] [ebp-24h]
  Parameters<2> arg; // [esp+34h] [ebp-20h]
  int unused; // [esp+50h] [ebp-4h]

  vticks.m_vticks = 32;
  pDTc = v->m_pDTc;
  vticks.m_init_vticks = &v->vticks32;
  vticks.m_pC = pDTc;
  unused = 0;

  // Pull two parameters off the stack from v into the local Parameters array arg.
  // This first parameter is just the literal raw value found on the stack, in this case,
  // it's an LPCSTR, but /in the emulator/, so it's a pointer in the emulators
  // virtual address space. The second parameter is a unsigned integer, so
  // the parameter value is literally just that integer

  Parameters<2>::Parameters<2>(&arg, v);

  // set return value to 1

  pe_set_return_value(v, 1ui64);
  *&attr.first.length = 0;
  *&attr.second.length = 0;
  attr.attribid = 12291;
  attr.second.numval32 = 0;
  Length = 0;

  // translate the parameter 0 pointer into a real native pointer that
  // the emulator can interact with

  attr.first.numval32 = pe_read_string_ex(v, arg.m_Arg[0].val64, &Length, v2);

  attr.first.length = Length;
  __siga_check(v, &attr);

  //emulate creating a new process, do various AV internal stuff

  vticks.m_vticks = pe_create_process(v, arg.m_Arg[0].val32, 0i64, v2) != 0 ? 16416 : 1056;
  CAutoVticks::~CAutoVticks(&vticks);
}
```

Figure 7. Annotated Hex-Rays decompilation of the emulated `kernel32!WinExec`.

We are going to implement our own function to replace `KERNEL32_DLL_OutputDebugStringA` that will actually print output to `stdout` so that we can pass information from inside of the emulator to the outside world.

We begin engineering by pulling down a copy of Tavis Ormandy's LoadLibrary, an open source harness that allows us to run `mpengine.dll` on Linux.[67] LoadLibrary parses and loads the `mpengine.dll` Windows PE into executable memory on Linux, and patches up the import address table to functions providing simple emulation of the Windows API functions that Defender invokes. Once loaded, the engine is initialized, and scanning is invoked by calling Defender's `__rsignal` function, which takes input and directs it to various AV scanning subsystems. While this research could also easily be done with a custom Windows harness for Defender, Tavis' tool is readily accessible and easy to use. Once we have LoadLibrary working, we can easily modify it to manipulate the loaded `mpengine.dll` library in memory.

Our first step is to hook the `KERNEL32_DLL_OutputDebugStringA` function. As the function is only ever invoked via function pointer, it's easiest to simply replace the function pointer in the `g_syscalls` array. We can write our own function with the same `__cdecl` calling convention that simply takes a `void *` and put a pointer to it in the `g_syscalls` table, replacing the original pointer to `KERNEL32_DLL_OutputDebugStringA`. Copying how the real Defender code does things, we call the Parameters<1>::Parameters<1> function to retrieve the one parameter passed to the function - this can be done easily by simply locating the function in the DLL, creating a correctly typed function pointer to it, and calling it as shown in Figure 8.

Running this code produces some basic output:

```
1 OutputDebugStringA called!
  OutputDebugStringA parameter: 0x4032d8
```

Simply knowing what parameters were passed to the function is nice, but not incredible useful. Copying the techniques used in other Defender native API emulation functions, we can use `__mmap_ex` to translate this virtual pointer to a real native pointer that we can read from. Unfortunately, calling `__mmap_ex` is not as painless as calling `Parameters<1>::Parameters<1>` as it has an odd optimized calling convention: `pe_vars_t *v` is passed in register `ecx` (like the `thiscall` convention), but then `unsigned int size` is passed in `edx`. I found the easiest way to get around this was to simply write my own a bit of x86 assembly we can trampoline through to get to it as shown in Figure 9.

Now we can add these calls to `e_mmap` into our code so that we can retrieve strings passed to `OutputDebugStringA` to obtain the implementation in Figure 10. Running this code yields our desired functionality:

```
OutputDebugStringA
OutputDebugStringA parameter: 0x4032d8 ->
    Hello World! This is coming from inside
    the emulator!
```

With this hook now set up, we have an easy way to pass information from within the emulator to outside of it. Exploring the environment inside the emulator is now as easy as literally printing to the terminal.

Using the APIs and techniques demonstrated to create a two-way IO channel where we can give input to the malware running inside the emulator (for example, to generate fuzzer test cases for emulated APIs on the outside and pass them to a malware binary on the inside) is left as an exercise for the reader.

[67] `git clone https://github.com/taviso/loadlibrary`

```
1  static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void * v)
   {
3      uint64_t Params[1] = {0};
       const char * debugString;
5
       printf("OutputDebugStringA called!\n");
7
       Parameters1(Params, v); //calling into mpengine.dll's Parameters<1>::Parameters<1>
9
       printf("OutputDebugStringA parameter: 0x%x\n", Params[0]);
11
       //don't worry about bumping the tick count
13
       return;
15 }

17 .text:5A129E20  dd offset ?KERNEL32_DLL_CopyFileWWorker@@YAXPAUpe_vars_t@@@Z
   .text:5A129E24  dd 0B27D5174h
19 //We'll replace this function pointer:
   .text:5A129E28  dd offset ?KERNEL32_DLL_OutputDebugStringA@@YAXPAUpe_vars_t@@@Z
21 .text:5A129E2C  dd 0B28014BBh
   .text:5A129E30  dd offset ?NTDLL_DLL_NtGetContextThread@@YAXPAUpe_vars_t@@@Z
23 .text:5A129E34  dd 0B363A610h

25 ...
       typedef uint32_t __thiscall(* ParametersCall)(void * params, void * v);
27     ParametersCall Parameters1;

29     ...

31     uint32_t * pOutputDebugStringA;
       //get the real address of the function pointer, mpengine.dll loaded image base + RVA
33     pOutputDebugStringA = imgRVA(pRVAs->RVA_FP_OutputDebugStringA);
       *pOutputDebugStringA = (uint32_t)KERNEL32_DLL_OutputDebugStringA_hook; //insert hook
35
       Parameters1 = imgRVA(pRVAs->RVA_Parameters1);
37 ...
```

Figure 8. Early `OutputDebugStringA` Hook

Defender defines `__mmap_ex` as:

```
char *__usercall __mmap_ex@<eax>(pe_vars_t *v@<ecx>, unsigned __int64 addr,
                                 unsigned int size@<edx>, unsigned int rights);
```

We emulate this function through the following call stack:

```
extern void * __cdecl ASM__mmap_ex(void * FP, void * params, uint32_t size,
                                   uint64_t addr, uint32_t rights);

void * e_mmap(void * V, uint64_t Addr, uint32_t Len, uint32_t Rights)
{
    //Trampoline through assembly with custom calling convention.
    //FP__mmap_ex is a global function pointer to the __map_ex function
    return ASM__mmap_ex(FP__mmap_ex, V, Len, Addr, Rights);
}
```

Where the function's assembly implementation is:

```
    ASM__mmap_ex:
        push ebp
        mov ebp, esp
        mov eax, [ebp+0x8]          ; function pointer to call
        mov ecx, [ebp+0xc]          ; pe_vars_t v
        mov edx, [ebp+0x10]         ; unsigned int size
        push dword [ebp+0x1c]       ; unsigned int rights
        push dword [ebp+0x18]       ; unsigned __int64 addr hi
        push dword [ebp+0x14]       ; unsigned __int64 addr low
        call eax
        add esp, 0xc
        pop ebp
        ret
```

Figure 9. Calling `__mmap_ex` with the unique calling convention.

```
    static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void * v)
    {
        uint64_t Params[1] = {0};
        char * debugString;
        DWORD len = 0;

        printf("OutputDebugStringA\n");
        GetParams(v, Params, 1);

        debugString = e_mmap(v, Params[0], 0x1000, E_RW);

        printf("OutputDebugStringA parameter: 0x%x -> %s\n", Params[0], debugString);

        return;
    }
```

Figure 10. Final implementation of the `OutputDebugStringA` hook.

ret2apicall

As previously discussed, the `apicall` opcode (`0f ff f0`) is custom addition to Defender's CPU emulator used to trigger calls to native API emulation routines stored in the `g_syscalls` array. While these native API emulation routines include complex-to-emulate but standard Window APIs (`NtWriteFile`, `ReadProcessMemory`, `VirtualAlloc`, etc.), there are also a number of unique, Defender-specific functions reachable with the `apicall` instruction. These Defender-specific functions include various "`VFS_*`" functions (e.g., `VFS_Read`, `VFS_Write`, `VFS_CopyFile`, `VFS_GetLength`, etc.) providing low level access to the virtual file system[68] as well as internal functions allowing administration of the engine (`NtControlChannel`) and interfacing with the Defender's antivirus engine. (`Mp*` functions, such as `MpReportEvent`, which is used internally to report that malware took a particular action during emulation.) These special functions should normally only be invoked internally from the Defender emulator by code put there, for example as shown in Figure 11, the in-emulator emulation routine for `ntdll!ZwSetLdtEntries` invokes `MpReportEvent(0x3050, 0, 0)` - ostensibly the value (or "`attribid`" according to Microsoft symbols) 0x3050 indicates to some heuristic malware classification engine that `ZwSetLdtEntries` was called.

In Summer 2017, Tavis Ormandy of Google Project Zero took a look at internal functions and found vulnerabilities in them.[69] Tavis' `NtControlChannel` bug simply linked against `ntdll!NtControlChannel`, but his VFS bug PoC had to use the `apicall` instruction to hit `ntdll!VFS_Write`, which he did using standard `.text` code in his malware binary.[70]

After fixing these bugs, Microsoft attempted to lock down these attack surfaces by limiting where the `apicall` instruction could be used. Newly added checks in the 1.1.13903.0 (6/23/2017) `mpengine.dll` release look before the function actually dispatches to a native API emulation handler look if the instruction is being run from a VDLL page (`is_vdll_page`), and if not, if it is a dynamic page (`mmap_is_dynamic_page`). Using the instruction can even trigger a call to `MpSetAttribute` informing Defender that it was used - likely a very strong heuristic indicator of malicious intent.

```
1  ...
   if( !is_vdll_page(v5, v25) ) {
3    v14 = v6;
     if(!mmap_is_dynamic_page(v28, *(&v26-1))
5       || nidsearchrecid(v29) != 1 ){
       if( !*(v2 + 167454) ){
7        qmemcpy(&v36, &NullSha1, 0x14u);
         v15 = *v2;
9        MpSetAttribute(0,0,&v36,0,*(&v27-1));
         *(v2 + 167454) = 1;
11     }
       return 0;
13   }
   }
15 ...
```

Looking at that initial check, `!is_vdll_page`, it's quite obvious how we can get around it: we need to come from a VDLL page. As I've shown throughout this article, the `apicall` instruction can be found throughout the process memory space in VDLLs. Dumping out VDLLs,[71] we see that they contain `apicall` instructions (see Figure 12) for invoking many of the native emulation functions that Defender supports - both those necessary for the operations the particular VDLL may use as well as other ones that are not used by that particular VDLL.

Calling these internal APIs is a simple as just trampolining through these `apicall` instruction function stubs, which are accessible from executable memory loaded into the process space of the malware executing within the emulator. For example, in a particular build of the emulator where `kernel32.dll` has an `apicall` stub function for `VFS_Write` at `RVA +0x16e66`, the following code can

[68] The virtual file system is stored all in memory during emulation. On a real system usermode Native (Nt*) APIs would do system calls into the kernel where they would ultimately be handled. In Defender, the `VFS_*` functions are akin to these kernel level handlers, they provide low level access to operations on the in memory file system.

[69] `https://bugs.chromium.org/p/project-zero/issues/detail?id=1260`
`https://bugs.chromium.org/p/project-zero/issues/detail?id=1282`

[70] The `VFS_Write` function did little validation on input values, and Tavis was able cause heap corruption by writing odd values to it. As Defender's emulation of `ntdll!NtWriteFile` ultimately calls into `VFS_Write` after doing some input validation, fuzzing that API on the a old unpatched version of Defender, I was able to reproduce Tavis' same heap corruption, but using different inputs that passed `NtWriteFile` validation. (Tavis's inputs did not.)

[71] We can simply find them on disk in the virtual file system in the standard `C:\Windows\System32` directory, read them in, and then pass them out via an output channel like that discussed previously in "Creating an Output Channel."

```
    public ZwSetLdtEntries
2   ZwSetLdtEntries proc near

4   mov     edi, edi
    push    ebp
6   mov     ebp, esp
    push    0
8   push    0
    push    3050h
10  call    apicall_KERNEL32_DLL_MpReportEvent
    pop     ebp
12  jmp     loc_7C96B6C2

14  loc_7C96B6C2:
    mov     edi, edi
16  call    $+5
    add     esp, 4
18  apicall ntdll!NtSetLdtEntries
    retn    18h
```

Figure 11. Disassembly of `ntdll!ZwSetLdtEntries`.

```
1  .text:7C816E3E 8B FF                    mov     edi, edi
   .text:7C816E40 E8 00 00 00 00           call    $+5
3  .text:7C816E45 83 C4 04                 add     esp, 4
   .text:7C816E48 0F FF F0 41 3B FA 3D     apicall ntdll!VFS_GetLength
5  .text:7C816E4F C2 08 00                 retn    8
   .text:7C816E52 ; ---------------------------------------------------------------------------
7  .text:7C816E52 8B FF                    mov     edi, edi
   .text:7C816E54 E8 00 00 00 00           call    $+5
9  .text:7C816E59 83 C4 04                 add     esp, 4
   .text:7C816E5C 0F FF F0 FC 99 F8 98     apicall ntdll!VFS_Read
11 .text:7C816E63 C2 14 00                 retn    14h
   .text:7C816E66 ; ---------------------------------------------------------------------------
13 .text:7C816E66 8B FF                    mov     edi, edi
   .text:7C816E68 E8 00 00 00 00           call    $+5
15 .text:7C816E6D 83 C4 04                 add     esp, 4
   .text:7C816E70 0F FF F0 E7 E3 EE FD     apicall ntdll!VFS_Write
17 .text:7C816E77 C2 14 00                 retn    14h
   .text:7C816E77 ; ---------------------------------------------------------------------------
19 .text:7C816E7A 8B FF                    align 4
   .text:7C816E7C E8 00 00 00 00           call    $+5
21 .text:7C816E81 83 C4 04                 add     esp, 4
   .text:7C816E84 0F FF F0 1D 86 73 21     apicall ntdll!VFS_CopyFile
23 .text:7C816E8B C2 08 00                 retn    8
```

Figure 12. Dump from `kernel32.dll` showing functions that use the apicall instruction.

```
unsigned int offset_apicall_KERNEL32_DLL_VFS_Write = 0x16e66;

typedef bool (WINAPI * apicall_VFS_Write_t)(uint32_t HFile, void * Buf,
                uint32_t BufSize, uint32_t Offset, uint32_t * PBytesWritten);

apicall_VFS_Write_t VFS_Write;

kernel32Base = (uint32_t)GetModuleHandleA("kernel32.dll");
VFS_Write = (apicall_VFS_Write_t)(kernel32Base + offset_apicall_KERNEL32_DLL_VFS_Write);

VFS_Write(...);
```

be used to reach it from within the emulator.

With the ability to hit these internal APIs, attackers have access to a great attack surface, with a proven history of memory corruption vulnerabilities. They can also cause trouble by changing various signatures hits and settings via `MpReportEvent` and `NtControlChannel`. Finally, if an attacker does find a vulnerability in the engine, invoking `NtControlChannel(3, ...)` provides engine version information, which can be helpful in exploitation, if you have pre-calculated offsets for ROP or other memory corruption.

When I reported this issue to Microsoft, they said "*We did indeed make some changes to make this interface harder to reach from the code we are emulating - however, that was never intended to be a trust boundary. [...] Accessing the internal APIs exposed to the emulation code is not a security vulnerability.*"

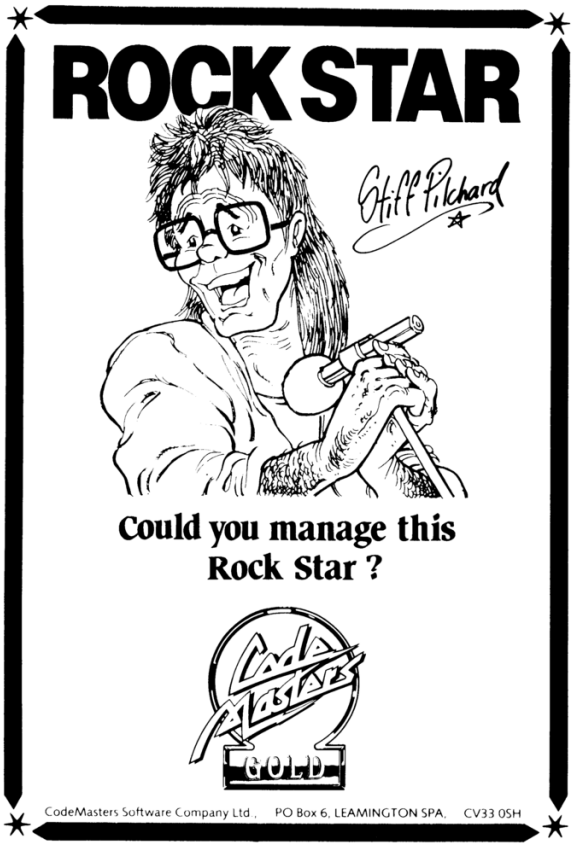
Disassembling Apicall Instructions

Throughout this article, I've shown disassembly from IDA with the `apicall` instruction cleanly disassembled. As this is a custom opcode only supported by Windows Defender, IDA obviously can't normally disassemble it. After I dumped VDLLs out of the emulator from the system32 directory, I found they could be loaded into IDA cleanly, but the dissasember was getting confused by `apicalls`.

As a reminder, this instruction is formed by the bytes `0f ff f0` followed by a four byte immediate of the CRC32 of the uppercase DLL name xored with the CRC32 of the function name.

Attempting to this code, IDA chokes on the `0f ff f0` bytes, and then attempts to disassemble the bytes after it, for example, the four byte immediate. We can see this in `ntdll!MpGetCurrentThreadHandle`:

```
.text:7C96C577 MpGetCurrentThreadHandle_0:
.text:7C96C577 8B FF          mov     edi, edi
.text:7C96C579 E8 00000000    call    $+5
.text:7C96C57E 83 C4 04       add     esp, 4
.text:7C96C581 0F FF F0       db 0Fh,0FFh,0F0h
.text:7C96C584 D5 60          aad     60h
.text:7C96C586 D5 8C          aad     8Ch
.text:7C96C588 C3             retn
```



Using a lesser-known feature of IDA's scripting interface, we can write a processor module extension. I based my code off of Rolf Rolles' excellent blogs on writing processor module extensions.

This processor module extension runs during module loading and analysis, and outputs disassembly for the `apicall` instruction. The full code is included in this issue, here I'll walk through some of the interesting parts.

As this script is invoked for every binary we load in IDA, we want to make sure that it only steps in to do disassembly for binaries we know to be Defender related. The checks in the init function shown in Figure 13 make sure that the plugin will only run for x86 binaries with "`.mp.dll`" in their name.

Our `parse_apicall_hook` class inherits from `idaapi.IDP_Hooks`, and we provide implementations for several of the classes methods.

The `hashesToNames` map is a map of function CRCs to their names. A script to generate this map is included in the comments of the included `apicall` parsing script. This and other functions discussed here are shown in Figure 14.

`ev_ana_insn` fires for each instruction IDA analyzes. In this function we grab three bytes at the address where IDA thinks there is an instruction, and check if they are `0f ff f0`. If they are, we look up the function hash to see if we have an implementation for it, and also set a few traits of the instruction - setting it to be seven bytes wide (so that IDA will know to disassembly the next instruction seven bytes later), and setting it to having a dword immediate operand of the API CRC immediate.

`ev_out_mnem` actually outputs the mnemonic string for the instruction - in this case we print out `apicall` and some spaces.

Finally, `ev_out_operand` outputs the operand value - since we know all the instruction CRC hashes, we can output those names as immediates.

With this extension dropped in our IDA plugins folder, we get clean disassembly of the `apicall` instruction when loading binaries that use it.

In conclusion, we've looked at three tricks for reverse engineering and attacking Windows Defender. While these tricks are Defender specific, the general intuition about AV emulator design and how a reverse engineer might go about approaching them should hold for other AVs. This article has mostly looked at techniques - for a look at Window Defender emulator internals, readers are encouraged to check out my conference presentations on the topic and to reverse the engine themselves.

```
    class apicall_parse_t(idaapi.plugin_t):
2       flags = idaapi.PLUGIN_PROC | idaapi.PLUGIN_HIDE
        comment = "MsMpEng apicall x86 Parser"
4       help = "Runs transparently during analysis"
        wanted_name = "MsMpEng_apicall"
6       hook = None

8       def init(self):
            self.hook = None
10          if not ".mp.dll" in idc.GetInputFile() or idaapi.ph_get_id() != idaapi.PLFM_386:
                return idaapi.PLUGIN_SKIP
12
            print "\n\n-->MsMpEng apicall x86 Parser Invoked!\n\n"
14
            self.hook = parse_apicall_hook()
16          self.hook.hook()
            return idaapi.PLUGIN_KEEP
18
        def run(self, arg):
20          pass

22      def term(self):
            if self.hook:
24              self.hook.unhook()

26  def PLUGIN_ENTRY():
        return apicall_parse_t()
```

Figure 13. IDA processor module initialization code.

```
hashesToNames = {3514167808L: 'KERNEL32_DLL_WinExec',
                 3018310659L: 'NTDLL_DLL_VFS_FindNextFile',...}

NN_apicall = ida_idp.CUSTOM_INSN_ITYPE
class parse_apicall_hook(idaapi.IDP_Hooks):
    def __init__(self):
        idaapi.IDP_Hooks.__init__(self)

    def ev_ana_insn(self, insn):
        global hashesToNames

        insnbytes = idaapi.get_bytes(insn.ea, 3)
        if insnbytes == '\x0f\xff\xf0':
            apicrc = idaapi.get_long(insn.ea+3)
            apiname = hashesToNames.get(apicrc)
            if apiname is None:
                print "ERROR: apicrc 0x%x NOT FOUND!"%(apicrc)

            print "apicall: %s @ 0x%x"%(apiname, insn.ea)

            insn.itype = NN_apicall
            insn.Op1.type = idaapi.o_imm
            insn.Op1.value = apicrc
            insn.Op1.dtyp = idaapi.dt_dword
            insn.size = 7 #eat up 7 bytes

            return True
        return False

    def ev_out_mnem(self, outctx):
        insntype = outctx.insn.itype

        if insntype == NN_apicall:
            mnem = "apicall"
            outctx.out_line(mnem)

            MNEM_WIDTH = 8
            width = max(1, MNEM_WIDTH - len(mnem))
            outctx.out_line(' ' * width)

            return True
        return False

    def ev_out_operand(self, outctx, op):
        insntype = outctx.insn.itype

        if insntype == NN_apicall:
            apicrc = op.value
            apiname = hashesToNames.get(apicrc)

            if apiname is None:
                return False
            else:
                s = apiname.split("_DLL_")
                operand_name = "!".join( [s[0].lower(), s[1]] )
            print "FOUND:", operand_name

            outctx.out_line(operand_name)

            return True
        return False
```

Figure 14. Excepts from the IDA processor module for parsing apicall instructions.