



ShellWasp and Offensive Usage of Windows Syscalls in Shellcode

Bramwell Brizendine¹

¹ University of Alabama in Huntsville, Huntsville, AL, USA
bramwell.brizendine@uah.edu

Abstract. While syscalls and Windows have exploded in popularity, permitting offensive security tools to weaponize direct Windows syscalls to avoid EDR, they have virtually never been utilized in the context of shellcode, except for Egghunters, a specialized shellcode that uses only one syscall. The reason why syscalls historically had not been used much, and why they have not been used in the context of shellcode, is the lack of portability for shellcode as the necessary syscall values (SSNs), which must be provided in the `eax` register. SSNs can change with each new OS build. Windows 10, for instance has more than thirteen distinct OS builds. This research provides a novel methodology to overcome the portability problem with shellcode. This research presents a novel tool, ShellWasp, which provides a solution to the portability problem, while automating much of the process in terms of utilizing said solution. With ShellWasp, the SSN for a particular syscall can be delivered at runtime, allowing for a syscall to be able to work across not only multiple OS builds, but multiple OS's. In addition to providing a solution to the aforementioned research problem, ShellWasp is able to reduce the human labor required to construct syscall shellcodes, by generating a template for syscall shellcode, with syscall parameter types and names labeled.

Keywords: Shellcode, Windows syscalls, Reverse Engineering, Offensive Security

1 Introduction

Prior to the work with ShellWasp it has been extremely rare for Windows syscalls to be used in shellcode. In fact, with the exception of egghunters there has only ever been one truly documented case of a shellcode with complex Windows syscall usage prior to this research. That was a limited experiment in 2005 during the Windows XP era. There have been two primary reasons why Windows syscalls have not been used in shellcode. First, there has been a significant issue of portability, where the necessary syscall number is not predictable and can change from one OS build, to the next. Thus, if somebody were to write a shellcode for a particular OS build of Windows, then with a new Windows update, then that shellcode may no longer work due to the fact that its syscall numbers had changed. Relevant to this is the fact that with Windows 10 there have been more than thirteen different OS builds, and for each of these, it is possible that the syscall numbers (SSNs) may have changed. Second, there has been a not in substantial lack of documentation in terms of how to utilize syscalls or native API (NTDLL) functions in general. Performing equivalent actions with WinAPIs is often significantly simpler and easier than doing so with NTDLL functions or syscalls. With the native API, often there may be many complex structures that need to be utilized, and some of these are automatically created by equivalent to WinAPIs. Due to lack of documentation, when using equivalent syscalls or native API, it may be necessary to reverse engineer equivalent WinAPIs, to discover necessary parameters and structures.

While syscalls have become very trendy in the world of red team offensive operations, their usage has been more confined to a smaller subset of syscalls. Moreover, these have been utilized in the context of higher-level languages, such as C, C++, C#, and with these languages it is much easier and simpler to create necessary resources. When performing equivalent actions in low-level Assembly, it is necessary to create needed structures in a more unorthodox fashion, such as multiple Assembly push instructions, used to build required structures, sometimes even with careful attention paid to stack alignment. In short, it is no wonder that Windows syscalls have virtually never been used in the context of shellcode outside of egghunters, and with a single exception.

We can define syscall shellcode as position independent code that is primarily, if not exclusively, comprised of Windows syscalls, while avoiding usage of traditional WinAPI functions. The reason why syscall shellcodes was not used, even in spite of their relative difficulty in usage, is again that lack of portability, where a shellcode with hardcoded syscall values could quickly become unusable. That was indeed the case with the 2005 syscall shellcode. Thus, the problem with Windows syscall usage in shellcode is the unpredictability of the required syscall values in their lack of portability given that required SSN's could quickly change without notice on new OS builds.

To overcome this research problem, I have created a way to dynamically resolve the necessary SSN's for syscall usage in shellcode for the 32-bit WoW64 environment. Thus, a shellcode could be viable across not just multiple OS builds, but multiple entire operating systems, including those which invoke the syscall differently. This technique is utilized and embodied in the design science research [1] artifact, ShellWasp. As such, for brevity's sake, I will refer to this as the ShellWasp approach to syscall shellcode. In short, with this technique the PEB is traversed to determine the target machine's OS build, and then a special syscall array is created that holds the values of the correct syscall numbers (SSN) needed for a particular syscall. In general, the syscall approach is compact in size, allowing the user to selectively target specific OS builds, such as the most recent ones, keeping the resulting Assembly smaller in size – a necessity for a lot of shellcode, although not all. The ShellWasp artifact also manages usage of the syscall array, ensuring the correct syscall value is supplied at runtime, without the need for the programmer to be cognizant of such details.

The organization of this paper is as follows. First, a detailed background on Windows syscalls will be provided. This is the result of original reverse engineering efforts and close examination of the existing literature. The focal point of this background is syscall usage for Windows show code in the 32-bit, WoW64 environment. This background also covers syscall usage in many of the existing research projects and tools. Second, we present the novel artifact ShellWasp, which not only provides a solution to the syscall portability problem with shellcode, but it also generates a complete template for desired syscalls, populating them with the necessary parameters, to be customized, and managing usage of the syscall array, so that the right syscalls are utilized at runtime. Third, we provide validation to the ShellWasp approach to syscall shellcode, by providing case studies consisting of original syscall shellcodes the utilize the ShellWasp technique with success.

2 Background

In the past there has been only very minimal discussion on Windows syscalls in academia [2]. They have only very briefly been referenced in the context of other academic research. Despite this lack of representation in the academic literature, they have been used in a much more limited context in the form of egghunters. An egg-hunter is a highly specialized shellcode designed to find a bigger shellcode or egg injected somewhere into process memory and an indeterminate location [3, 4]. In order to discover the location of an egg, it is necessary to use a syscall, such as NtAccessCheckAndAuditAlarm, to make sure that memory is valid before checking for a unique tag, such as w00tw00t. Once the location of the tag is discovered, the shellcode can then redirect control flow to the location designated by the tag. Syscalls have been utilized for this purpose simply because WinAPIs had not provided the corresponding functionality, so it was a necessity to use them. However, outside of this limited usage, syscalls have virtually never been utilized in shellcode.

Kernel32 provides an interface to the native API, or NTDLL functions. In turn, those NTDLL functions will both setup and invoke the syscall. Not all NTDLL functions have a one-to-one correspondence with a Windows syscall. In fact, some NTDLL functions may call multiple syscalls. With other native APIs, a syscall is immediately called. With those native APIs that exist only to be a wrapper to a corresponding syscall, the system service number (SSN) of a syscall is provided as value that is moved into eax. Following this, the syscall is then

invoked. The method by which it is invoked varies, both from 32- and 64 bit architecture, as well as different variations within 32-bit, WoW64. As this research is confined to the 32-bit, WoW64 environment, our discussion will be focused on this.

In WoW64, Windows 7, the syscall can be made rather simply by calling a function that is pointed to by an address stored at fs:0xc0. The TIB is an internal Windows structure that provides a tremendous wealth of information pertaining to introspection, and the TIB is pointed to by the fs register. In Windows 10 and 11, this has been supplanted by an offset to a function that is moved into edx, and which is then called. However, while Windows 10 and 11 do not utilize the call to fs:0xc0 directly, it is maintained for purposes of backwards compatibility. For purposes of shellcode, directly using the fs:0xc0 is far simpler. While it is not necessary, the alternative would be to traverse the PEB and walk the exports directory to resolve the address of the function that is called in Windows 10 and 11.

In this section, we provide a practical background to enable better comprehension of how Windows syscalls have been used in both current and historical versions of the Windows operating system. This information was obtained through a reverse engineering, and it is necessary to understand this, so that ShellWasp can be designed to work across multiple OSs. In addition, we provide a brief survey of the current and leading techniques for syscall usage, although again we point out that all of this research and techniques are outside of the context of shellcode. Still, since they involve an offensive approach to syscall usage, it is necessary to survey them.

2.1 Historical Usage of Windows Syscalls for Offensive Purposes

As has been previously mentioned, syscalls have a long history of being utilized as part of egghunter shellcodes. Many sources [3, 4] briefly allude to egghunters, although much of the discussion has been outside of academia. In general, while egghunters do indeed allow for a secondary shellcode fragment to be discovered in process memory, they typically are not utilized, unless it is absolutely necessary to do so. If a more straightforward path is available, then generally they are not utilized. This is due to various factors. First, the implementation details of their usage are not always well understood, and most people that utilize egghunters simply use or adapt a prebuilt egghunter, while not necessarily understanding all the intricate nuances. Second, an egghunter sometimes can be time intensive to discover the needed tag, and in fact they may not always succeed. Third, usage of egghunters and shellcode demands a greater low-level technical sophistication. For some egghunters to succeed or do so in a more reliable time frame, some minor adaptation may be necessary, and that is not possible if the practitioner does not have the requisite knowledge. As reliability is paramount in shellcode usage, this would rule egghunters out general except for cases where they are necessary. Egghunters have been used both in the world of exploitation as well as in malware. Indeed, malware will often use shellcode, and in fact shellcode used by malware tends to be more complex than exploitation shellcode. That is owing to the greater sophistication in general of malware authors as well as the fact they are not constrained by size limitations that often come into play with exploitation.

In 2005, Piotr Bania created a shellcode that made use of four syscalls, to allow for the shellcode to establish persistence for a binary by creating a registry key to cause an application to execute upon reboot. While his shellcode has four syscalls, actually only two of the syscalls are necessary, as the other two simply close a handle and terminate the process. Bania wrote also wrote a short, detailed paper on Windows syscall usage for the Windows XP era. While much of what he said remains true, parts of it are dated due to the passage of time and the evolution of Windows. In addition, Bania generously provides the source code for his shellcode. Written for MASM, his shellcode is interesting to examine, to see how it invokes the shellcode. The way in which it is invoked is now obsolete, but it was still useful to gain a better understanding of syscall usage in shellcode at the very beginning of this research. His shellcode, thus, is not viable and would not work on a modern OS, due to the change in how the syscall is invoked. However, as an early part of this research, a VERONA Lab member, Shelby VandenHoek, updated his shellcode, rewriting it from scratch to utilize the ShellWasp approach. The new version of the persistence syscall shellcode can run on Windows 7, 10 and 11.¹

Over the years, occasionally some people would utilize or toy with Windows syscalls, in the context of higher-level languages, not shellcode, but they were not widely used by the mainstream. By 2018, Hod Gavriel had

¹ We intentionally omit Windows 8, as we regard it as irrelevant and not widely used. Our focus and research is limited only to Windows 7, 10 and 11.

observed a noticeable increase in Windows syscall usage in malware [5]. He released a report on malware usage in new malware, and he provided hashes for eight malware samples that utilized syscalls, including Flokibot, Fastcash, LockPos, Trickbot, Formbook, Neurevt, Osiris, and Coininer. His report brought attention to the fact that Windows syscalls could be used offensively, and his report is directly responsible for the influx of offensive syscall tools that would soon follow in the following year.

2.2 Implementations of Syscalls in Different Windows Operating Systems

With Windows 7, WoW64, syscalls utilize the Thread Information Block (TIB), as pointed to by the fs register. At offset 0xC0 is the function X86SwitchTo64BitMode in wow64cpu.dll, as can be seen in the below figure. The SSN, as always, is loaded into eax. Stack cleanup occurs after the call to fs:[0xc0]; four bytes are added to esp to accommodate for the address of the next instruction added for the call. Next, additional stack cleanup occurs with the ret. With Windows 7 WoW64 the convention is to cleanup the syscall parameters with ret. In the example below, *ret 10h* is used. The 0x10 accounts for the four parameters for NtQuerySystemInformation; each takes up four bytes, so that is a total of 16, or 0x10, bytes.

```
0:000> u ntdll!ntquerysysteminformation
ntdll!NtQuerySystemInformation:
7702fdb0 b833000000 mov     eax,33h
7702fdb5 33c9      xor     ecx,ecx
7702fdb7 8d542404  lea     edx,[esp+4]
7702fdbb 64ff15c0000000 call    dword ptr fs:[0C0h]
7702fdc2 83c404    add     esp,4
7702fdc5 c21000    ret     10h
```

Fig. 1. A snippet from WinDbg shows fs:[0xC0] being used to facilitate syscalls.

This function is used to make a far jump that goes to CpuReturnFromSimulatedCode in wow64cpu.dll, as seen in the figure below. WoW64cpu.dll is a 64-bit library loaded into 32-bit address space, and by default it is hidden from the PEB, so when that address space appears in a debugger, it is not labelled [6]. The far jump will also involve a transition from 32-bit to 64-bit mode. All kernel-mode is 64-bit, and thus a transition to 64-bit mode is a necessary first step, and the 0x33 code segment selector denotes the transition to 64-bit mode. With WoW64, we are running a 32-bit application in a 64-bit operating system, so this emulation is necessary. Once it returns from 64-bit to 32-bit mode, a 0x23 selector is used to allow it to transition back to 32-bit mode.

```
0:001> dd fs:[0C0h]
0053:000000c0 73062320 00000409 00000000 00000000
0053:000000d0 00000000 00000000 00000000 00000000

0:001> u 73062320
73062320 ea1e2706733300 jmp     0033:7306271E
73062327 0000      add     byte ptr [eax],al
73062329 cc      int     3
```

Fig. 2. Fs:[0xC0] leads to a far jump, allowing a transition from 32-bit to 64-bit mode.

From the above figure, it is apparent that there will be a jump to 0x7306271e. However, if we go to that location in 32-bit mode, the code is interpreted incorrectly in the debugger, which is 32-bit. Since wow64cpu.dll is a 64-bit library in 32-bit address space, the transition is needed, as this function completes the final setup before transition to kernel-mode. What goes on beyond the call to fs:0xc0 and internally within these functions is out-

side the scope of this paper, as we simply take a black box approach, where we know that if we make a syscall in WoW64 in this manner, that it will lead to the actual true syscall subsequently.

```

0:001> u 7306271E
wow64cpu!CpuSimulate+0x16e:
00000000`7306271e 67448b0424      mov     r8d,dword ptr [esp]
00000000`73062723 458985bc000000      mov     dword ptr [r13+08Ch],r8d
00000000`7306272a 4189a5c8000000      mov     dword ptr [r13+0C8h],esp
00000000`73062731 498ba42480140000      mov     rsp,qword ptr [r12+1480h]
00000000`73062739 4983a4248014000000      and     qword ptr [r12+1480h],0
00000000`73062742 448bda              mov     r11d,edx
wow64cpu!TurboDispatchJumpAddressStart:
00000000`73062745 41ff24cf              jmp     qword ptr [r15+rcx*8]
wow64cpu!TurboDispatchJumpAddressEnd:
00000000`73062749 4189b5a4000000      mov     dword ptr [r13+0A4h],esi

```

Fig. 3. The location of the far jmp to transition to 64-bit mode can be viewed in a 64-bit debugger.

In order to see the contents of what is at the far jump, we need to view the 32-bit process in a 64-bit debugger. We can see below that it utilizes 64-bit registers to set up a dereferenced jump to a location that is not entirely clear. The `jmp qword ptr [r15+rcx*8]` remains unclear. My assumption is it is likely some kernel address, and that it likely is reliably pointed to in some fashion by r15.

Windows 7 WoW64 easily lends itself to syscall shellcode, as all that is required is to set up parameters for the syscalls, reproducing the Windows 7 setup to invoke the syscall:

```

xor ecx, ecx
lea edx, [esp+4]
call dword ptr fs:[0xc0]
add esp, 4
ret

```

Note that calling just `fs:[0xc0]` by itself is not sufficient. As we can see, it is the necessary to do stack cleanup after the syscall, removing the parameters as well as the address added by the call.

Windows 7 is known to only have two different releases with different OS builds, which are Service Pack 0 and Service Pack 1, and the OS builds are end in 0x1DB0 and 0x1DB1.

2.3 Syscalls in Windows 10 WoW64

Syscalls in Windows 10 differ from those of Windows 7, with a completely new approach to calling syscalls.

```

0:004> u ntdll!ntsetinformationvirtualmemory
ntdll!NtSetInformationVirtualMemory:
77c944d0 b89e010000      mov     eax,19Eh
77c944d5 ba408bca77      mov     edx,offset ntdll!Wow64SystemServiceCall (77ca8b40)
77c944da ffd2            call    edx
77c944dc c21800         ret     18h

```

19Eh = SSN for NtSetInformationVirtualMemory

```

0:004> u 77ca8b40
ntdll!Wow64SystemServiceCall:
77ca8b40 ff252892d477      jmp     dword ptr [ntdll!Wow64Transition (77d49228)]

```

Fig. 4. Windows 10 WoW64 syscall for NtAllocateVirtualMemory.

As can be seen, the `fs:[0xc0]` offset is no longer used as a means to transition from 32- to 64-bit to invoke a syscall. Instead, an offset to `Wow64SystemServiceCall` is used. Next, a dereferenced jump to

ntdll!Wow64Transition is made. Wow64SystemServiceCall then contains a far jump in wow64cpu.dll, CpuReturnFromSimulatedCode, as can be seen in the figure below.

```

0:004> u 77ca8b40
ntdll!Wow64SystemServiceCall:
77ca8b40 ff252892d477 jmp dword ptr [ntdll!Wow64Transition (77d49228)]

0:004> dd 77d49228
77d49228 77c17000 77d49000 00000000 00000000

0:004> u 77c17000
77c17000 ea0970c1773300 jmp 0033:77C17009
77c17007 0000 add byte ptr [eax],al

```

This far jump let's us transition from 32- to 64-bit code.

Fig. 5. In Windows 10 WoW64 the far jump to 64-bit mode can be reached from ntdll!Wow64Transition.

Fortunately, the old method of invoking syscalls with fs:[0xc0] has been retained, as can be seen below. Looking at both the new and old ways of invoking it, we see that they point to the same location. This fact enables us to ignore the new way and simply continue to use the old fs:[0xc0] way of invoking syscalls.

```

0:004> dd fs:0xc0
0053:000000c0 77c17000 00000409 00000000 00000000

0:004> u 77c17000
77c17000 ea0970c1773300 jmp 0033:77C17009
77c17007 0000 add byte ptr [eax],al

0:001> u 77C17009
wow64cpu!KiFastSystemCall+0x9:
00000000 77c17009 41ffa7f8000000 jmp qword ptr [r15+0F8h]

```

This far jump let's us transition from 32- to 64-bit code.

r15+0xf8: wow64cpu.dll's CpuReturnFromSimulatedCode

Fig. 6. Windows 10 WoW64 maintains allows the far jump to 64-bit mode to be reached still via fs:[0xc0].

In Windows 10, when we examine the far jump and where we lead us, we see that the location to CpuReturnFromSimulatedCode can now only be reached via 64-bit code that points to a register that does not exist in 32-bit mode [7]. The r15+0xf8 maintains the address for this function. As before, CpuReturnFromSimulatedCode will establish the proper context and perform the actual system call.

In examining the disAssembly for syscalls made in Windows 10, we see that unlike with Windows 7, there is no *add esp, 0x04*. The stack parameters are cleaned up as before with a *ret*, and the number of bytes to be removed. For instance, a *ret 18* would remove 24 bytes (0x18 bytes) from the stack, as we can see with our example earlier from NtSetInformationVirtualMemory.

2.4 Windows 11 WoW64 Syscalls

From the perspective of an exploit developer, there is virtually no difference and utilizing Windows syscalls in Windows 10 and 11. However, the specific functions utilized differ than what is done in Windows 10. The function ntdll!RtlInterlockedCompareExchange64+0x180 calls upon ntdll!Wow64Transition, which the same function called in Windows 10. From the user-mode side, everything else is virtually identical. As before, Wow64Transition is able to give us an address that can be used to provide a link to the far jump that allows us to transition from 32- to 64-bit mode

```

0:000> u ntdll!ntallocatevirtualmemory
ntdll!NtAllocateVirtualMemory:
77884d50 b818000000 mov     eax,18h
77884d55 ba408f8a77 mov     edx,offset ntdll!RtlInterlockedCompareExchange64+0x180
(778a8f40)
77884d5a ffd2        call    edx
77884d5c c21800     ret     18h
77884d5f 90        nop

0:000> u 778a8f40
ntdll!Wow64SystemServiceCall:
778a8f40 ff2520c29377 jmp     dword ptr [ntdll!Wow64Transition (7793c220)]
778a8f46 cc        int     3

0:000> dd 7793c220
7793c220 77806000 7793c000 00000000 00000000

0:000> u 77806000
77806000 ea096080773300 jmp     0033:77806009

```

18h = SSN for NtAllocateVirtualMemory

Fig. 7. Windows 11 WoW64 is very similar to Windows 10 at the user-mode level.

The fs:[0xc0] register continues to point to the same location as is pointed to by ntdll!Wow64Transition. Again, this permits us to be able to reach the exact same far jump, allowing us to transition from x86 to x64 mode. In terms of utilizing Windows syscalls in shellcode, no difference in usage from Windows 10 has been detected.

```

0:000> dd fs:c0
0053000000c0 77806000 00000409 00000000 00000000

```

Fig. 8. In Windows 11 WoW64, fs:[0xc0] points to the same far jump as does ntdll!Wow64

2.5 Windows Syscall Used Outside the Context of Shellcode

Prior to the highly influential article by Gavriel, where he describes a sharp increase in Windows syscall usage and modern malware, there had been no syscall tools. In fact, at that time it was also very uncommon for syscalls to be used for general red team purposes. Since the release of his report, there have been several syscall tools and techniques created, which had been widely used, both in the red team community and by offensive security enthusiasts. Windows syscalls have become trendy, and any and all new advances in syscalls have attracted attention. While many of these tools have been very effective for the OS builds they were initially targeting, none of them are applicable to shellcode. All have addressed Windows syscalls from the standpoint of trying to weaponize syscalls for use in much larger software projects, such as those written in higher-level languages, like C++, C#, C, etc. While many of these may provide Assembly stubs or headers, or may minimally use Assembly, they are designed to be used in projects that are predominantly not Assembly. As this does not involve the usage of position independent shellcode, and as these various languages can easily allow an individual to provide access to necessary resources needed by syscalls, they can be significantly easier to use with syscalls, than creating equivalent actions in shellcode. As will be explored elsewhere, some of the efforts required to do equivalent actions in shellcode may involve considerably more work and effort.

The below comprises a survey of some of the more established and well known syscall tools and techniques.

Dumpert. In June 2019, Dumpert [9, 10] appeared as the first POC offensive security tool to utilize Windows syscalls. This was created in direct response to Gabriel's report on increased syscall usage in malware. Precom-

puted SSNs are utilized by Dumpert, and Dumpert also utilizes the native API `RtlGetVersion` to ascertain the OS version. With the correct OS version discovered, then the appropriate SSN's can be deployed. Dumpert was designed as POC code, able to perform LSASS memory dumps with Cobalt Strike.

SysWhispers. Released in December 2019, SysWhispers is a Python tool by Jacksson T., which can generate headers and ASM pairs for all syscalls contained in `ntoskrnl.exe`. SysWhispers pays homage to Dumpert, by utilizing precomputed SSN's they correspond to a specific OS version and OS build. Rather than using the native API to determine the OS version, SysWhispers utilizes the PEB to identify OS major version, OS minor version, and OS build. This tool was designed to work with 64-bit implementations of syscalls, rather than those that target the 32-bit, WoW64. SysWhispers was the first tool to make use of the PEB to determine both the Windows version as well as the OS build. Given that SysWhispers is designed for 64-bit architecture, it queried `GS:[0x60]` and then offset `0x120` to determine the OS Build. Prior to checking the OS Build, it checked the major version of Windows at offset `0x118` and the minor version of Windows at offset `0x11c`. By utilizing the PEB, the exact syscalls could be determined, by using SSNs from the syscall table prepared by Jurczyk [8]. The Assembly to determine the OS major version, minor version, OS build for each syscall was extremely long.

FreshyCalls. Released by ElephantSe4l in December 2020, FreshyCalls [11, 12] revolutionized Windows syscalls by creating a technique to programmatically determine syscall values on the fly. This is based on ElephantSe4l's research that showed that if addresses for syscall that begin with `Nt` were sorted, then it can be observed that the SSN values for each that are moved into `eax` will increment by one. Thus, his tool is able to utilize this information to programmatically determine a particular syscall's SSN based on its location, after having been sorted. With this technique, it then was possible to determine and SSN for a syscall without the need to know the OS version or OS build.

SysWhispers2. Released by Jackson T. in January 2021, SysWhispers2 [13] pays homage to ElephantSe4l in utilizing a variation of his sort by address technique. Thus, this is a reimagining of SysWhispers, designed to generate Assembly files and headers for for x64/AMD64 platforms. As with FreshyCalls, the Export Address Table of NTDLL is parsed and addresses are sorted. The primary difference is that now it begins with NTDLL functions beginning with `Zw`, instead of `Nt`. With this sorting by address technique, it is not necessary to use precomputed syscall values. SysWhispers2 allows the user to utilize an implant of the syscall stub, which is often just a few lines of Assembly code, to then call the syscall. While the chunk of code used for the implant is x64 Assembly often extracted or slightly changed from NTDLL, the intent is just to use that system call in a higher-level language as if it were a normal Native API function, often as a means to avoid EDR.

SysWhispers3. Released in March 2022, SysWhispers3 [14, 15] is a recent fork of SysWhispers2, that like SysWhispers2 generates syscall headers and Assembly pairs, allowing them to be used in C/C++ code, to evade hooks. SysWhispers3 includes other features, some of which have since been incorporated into SysWhispers2, such as support for generating WoW64 Assembly and header pairs. SysWhispers3 will replace the syscall instruction with "`W00tW00t`"—which it then replaces with the 64-bit *syscall* Assembly instruction, to help avoid detection.

Hell's Gate. Released in June 2020, Hell's Gate [16] is a technique to find syscalls dynamically by parsing NTDLL.dll and then locating the syscalls and extracting the SSNs, saving them to memory to be reused. In the white paper for this tool, the author describes using PEB walking to eventually traverse the PE file format, to locate the exports table. Once the start of an NTDLL function is found, it performs what it calls the "pseudo-disassembling," looking for the `mov eax` opcode, `0xb8`. Once found, it extracts the SSN by taking the next two bytes, and saving them to memory, where they can be used again. Hell's Gate will be able to extract the correct SSN. Another potential problem with Hell's Gate is that some EDR products will make changes to NTDLL, to deliberately make it so that the technique will not work on some high value native API's, which of course lead to their corresponding syscalls. There are techniques that can help overcome this, such as dual loading of NTDLL,

which can also be done through a syscall, if it is not modified by EDR. However, the ways in which in NTDLL can be dual loaded can be detected. If successful in dual loading a copy of NTDLL, then the clean version of the module can then be used with Hell's Gate.

Halo's Gate. Released in April 2021, Halo's Gate [17] is an extension of Hell's Gate that could overcome protection from EDR's, to prevent Hell's Gate from working. With Halo's Gate, the first several bytes of each native API is checked to see if they are hooked. If the opcodes for a *jmp* instruction, 0xe9, are present, then it is assumed that the API has been hooked. If that is the case, then Halo's Gate will search for a neighboring or close-by native API that is not hooked. Its SSN value is taken, and then its distance to the hooked native API is determined. By adding or subtracting the distance between the two native APIs, the SSN can then be determined for those functions that have been hooked. Thus, Halo's Gate will utilize the Hell's Gate approach, until a hooked native API is encountered, and then it makes use of these sorting by address technique, with its underlying assumption that each SSN will increment by one, from one syscall to the next.

FireWalker. FireWalker [18] causes a Vectored Exception Handler to try to locate system call stub after single stepping with use of it of EFLAGS register.

3 ShellWasp

ShellWasp [19] is a design science research artifact designed to overcome the portability problem associated with utilizing Windows syscalls in shellcode. It also creates a custom shellcode template for utilizing syscalls. With the shellcode templates produced, each parameter for each Windows syscall is enumerated in the comments, for ease of use by the programmer, who can use the template to write a full-fledged syscall shellcode. The template is also set up so that each SSN can be determined programmatically and supplied at runtime, while using precomputed SSNs, allowing for a single shellcode to run on multiple operating systems, including those with a different method of invoking the syscall. ShellWasp is the first tool to address the portability problem by parsing the 32-bit PEB, and it is the only tool or approach to creating syscall shellcode. ShellWasp provides support for all the latest OS builds of Windows 7, 10, and 11. ShellWasp's identification of syscalls and saving them into a syscall array is very compact in size. Additionally, given that most people have automatic updates, it is not necessary to exhaustively search every single OS build, so ShellWasp provides functionality for the user to determine the OS builds to include or exclude. Once a shellcode is written, it is very easy to change the OS builds included in the shellcode, without the need to rewrite any portion of the shellcode that follows.

3.1 ShellWasp Origins

The inspiration for ShellWasp came during the design of an original shellcode analysis framework, SHAREM[20]. Thus, the origins of ShellWasp was tied to a desire to provide support for emulating Windows syscalls, and an exhaustive search was made to find any and all instances of syscall shellcode. Aside from egg-hunters, as mentioned before, only a single syscall shellcode was known to exist, so there appeared to almost no samples to examine and study, for purposes of emulation. Given that the 2005 sample was obsolete, we only had egg-hunters as potential samples for emulation. In consulting with other cybersecurity experts, it was clear that they also knew of no other prior instances of syscall shellcode. While there may have been an extremely limited number of syscall shellcodes created prior to this work, that we do not know about, it was clear that largely syscall shellcode was something that had really not been done before. The reason for this is quite apparent, as the portability problem with using syscalls is a significant hurdle to overcome, and the overall difficulty in writing syscall shellcode relative to traditional Windows shellcode would effectively make writing syscall shellcode a mere curiosity for many. However, now with increased interest in utilizing the Windows syscalls for offensive security purposes as a means to evade EDR, the practical relevance of syscall shellcode cannot be overstated enough.

In order to be able to successfully emulate syscall shellcode, it was necessary to then discover how syscall shellcode works or should work, given that again largely syscall shellcode was not done and thus it was not

documented. Thus, a novel solution to the portability problem with syscalls was devised, so that syscalls could be used across multiple OS builds and OS versions. More relevant to syscall shellcode emulation, however, were some of the practical details of how each operating system invokes the syscall. That involved careful, original reverse engineering with Windows 7, 10 and 11, to understand the nuances of each. The existing literature of Windows syscalls was exhaustively studied, but much of that had only very minimal relevance to syscall shellcode. After all, doing things like Hell's Gate or Halo's Gate in shellcode, while certain possible, seemed impractical and a bit much. The 2005 syscall shellcode was examined carefully. However, that was only of minimal use, given that the method of invoking syscalls in 2005 is now obsolete. Egghunters samples for Windows 7 and 10 were also looked at closely, but both provided a number of unanswered questions regarding nuances that could only be answered through careful reverse engineering. Once an accurate understanding of utilizing Windows syscalls and shellcode was obtained, it was then necessary to generate a multitude of samples, so that they could be emulated. In order for them to be able to be emulated, they must first be able to work in the real world.

The research and effort necessary to be able to emulate Windows syscalls for SHAREM was so extensive, that it led to the creation of ShellWasp, which is an embodiment of concepts and ideas that were developed. Since its creation, other refinements and additions have been made, and others may be planned for the future.

A considerable amount of research had been done for SHAREM, and some of that was able to be integrated into ShellWasp. For instance, ShellWasp can generate comments for each parameter of the syscall, with the name and type of the handle identified. In order for this to be possible, it was necessary to have a dictionary of function prototypes for syscalls. That was laboriously created by finding examples of Windows syscalls at various sources, including Microsoft itself, NTAPI Undocumented Functions [21], and numerous other sources, including many open source projects. Given that much of the native API is not documented, if different sources wish to utilize a native API, then they need to somehow obtain or reverse engineer the function prototype. While about 230 or so native API's were able to be found via Microsoft or NTAPI Undocumented Functions, it was necessary to search for each additional undocumented native API one by one. For 99% of user-mode native API's, a function prototype was able to be found. All function prototypes were then aggregated into a single file, and some normalization was performed. A script was then written to parse these function prototypes, creating an original Python dictionary. This Python dictionary was already created for SHAREM, and so it was easy to adapt it and utilize it for ShellWasp. Since that time, I have also been made aware that there was already an existing listing of function prototypes for native API's. Had some of this initial work not already been completed, then writing ShellWasp could have taken considerably longer, than the small amount of time needed.

In wanting to overcome the portability problem of syscalls, careful study was made of the detailed PEB structure at Vergilius [22]. Having taught the writing of shellcode to hundreds of students at undergraduate and graduate level over several years, and in having closely studied the inner workings of modern shellcode for several years, I was convinced that there must be some element of the PEB that would indicate the version of the operating system. After all, if we could summarize the overall purpose of the PEB, it is to provide various forms of introspection for use with Windows internals. Thus, I simply carefully scrutinized the different members of the PEB, hoping to find one that would look useful, so that I could then test it in a debugger. I then came upon the PEB structure members OSMajorVersion, OSMinorVersion, and OS BuildNumber, which are located at offsets 0xa4, 0xa8, and 0xac of the PEB. To explore these, I then wrote sample inline Assembly code, so that I could check these in a debugger, while performing additional research to see if they would be useful. I was exhilarated to see that they were indeed as useful as they had at first appeared to be. Subsequently, I discovered that the first SysWhispers had walked the PEB in order to identify OS build, and major and minor version. However, I was not aware of this, and even if I had been, SysWhispers utilized the 64-bit PEB, which has entirely different offsets and uses the gs rather than the fs register. Still, ShellWasp is the first tool to utilize the 32-bit PEB or to target 32-bit, WoW64. Our approach is not so much as simply utilizing the PEB, but rather in creating the novel syscall array, so that the shellcode can utilize the correct SSNs. Other than using the PEB for identification, our approach is an low-level Assembly approach that bears no similarity to any other existing techniques.

3.2 Using Precomputed Syscall Tables

ShellWasp utilizes precomputed syscall tables for the Windows releases that targets For Windows 7 to Windows 10, OS build 20H2, this were prepared by Jurczyk [8] and released online at his web site and GitHub. For more

recent versions of OS builds for Windows 10 and 11, these were generated ourselves by either myself or Shelby VandenHoek. In all, several have been generated by us. As part of this research to generate accurate syscall results for ShellWasp, I developed my own technique to obtain SSN's. Prior to this, I had attempted to use another tool to parse this called values, but I had noted that the results were not accurate and that is what had led me to investigate this matter more closely. My approach is somewhat simple, but highly effective. I simply provide a script for WinDbg that is able to extract the single line of Assembly where the SSN is moved into eax. A secondary script is then used to convert that to a Python dictionary or JSON with all SSNs for a given OS build. This was initially developed for my own internal use, although I will release it on GitHub. Somebody more ambitious could turn it into a script that performs the entire process of extraction, from start to finish. Given this technique utilizes actual disassembly, it would be extremely unlikely that there would be any errors in the results produced. In all, it only takes a few minutes to use the two scripts to generate accurate SSN's for a particular OS build. With this approach, we are able to see accurate SSN's, including some that are much larger than before. Reliance on precomputed SSNs from a technique that is assured to be error-free would enhance the reliability of syscall efforts that target more recent OS builds.

Fig. 9. In Windows 11 WoW64, fs:[0xc0] points to the same far jump as does ntdll!Wow64

The assumption is that the syscall tables prepared by Jurczyk are reliable and without fault; we have not attempted any sort of verification, which would entail redoing all the work that he had completed. Initially, the desire was to use precomputed syscall tables as it was felt that it would be a far more compact and simple way to do it with shellcode, which typically would only use a handful of functions. For other approaches, it was thought the necessary Assembly likely would be too large, or various weird problems could be encountered.

3.3 Design of ShellWasp

ShellWasp must be designed such that it can generate fully accurate shellcode that would work not only on different operating systems, but different OS builds. For each it must create a template that can be adapted and used by the user.

Invoking the Syscall on Different Windows Operating Systems.

ShellWasp creates a special function that contains the method or methods of invoking the syscall, relevant to the target Oss. It can be observed that the method of invoking syscalls, while identical on Windows 10 and 11 from our perspective, differs from Windows 7, and an attempt to invoke the syscall in Windows 10 and 11 would fail on Windows 7. To address this, ShellWasp must identify the OS major version. Windows 10 and 11 share the same OS major version, and we need not differentiate between those two. The OS major version is stored right in front of the syscall array, for easy retrieval, at edi-4. Thus, if the target OS is Windows 7, then we invoke the syscall in the Windows 7 fashion, whereas if the target OS is Windows 10 or 11, then we invoke the syscall in the Windows 10 fashion. If the user selects to target both Windows 7 and 10 or 11, then the syscall

function has both implementations of the syscall, and before its invocation of the syscall, it will check the OS major version, which is stored at edi-4. One of the hallmarks of ShellWasp is its flexibility and potential customization, so the user is able to select which OS builds that they wish to target. If, for some reason, they are only targeting the Windows 7, then only that implementation of how to invoke the syscall is included, whereas with the user who is targeting only modern OS's, then the Windows 7 implementation can be omitted. The user does not need to select the OS major versions that they are targeting, as that is already made clear by the OS builds that they choose to target. The assumption is that they will only target more recent OS builds of operating systems, although some may wish to target older systems such as Windows 7.

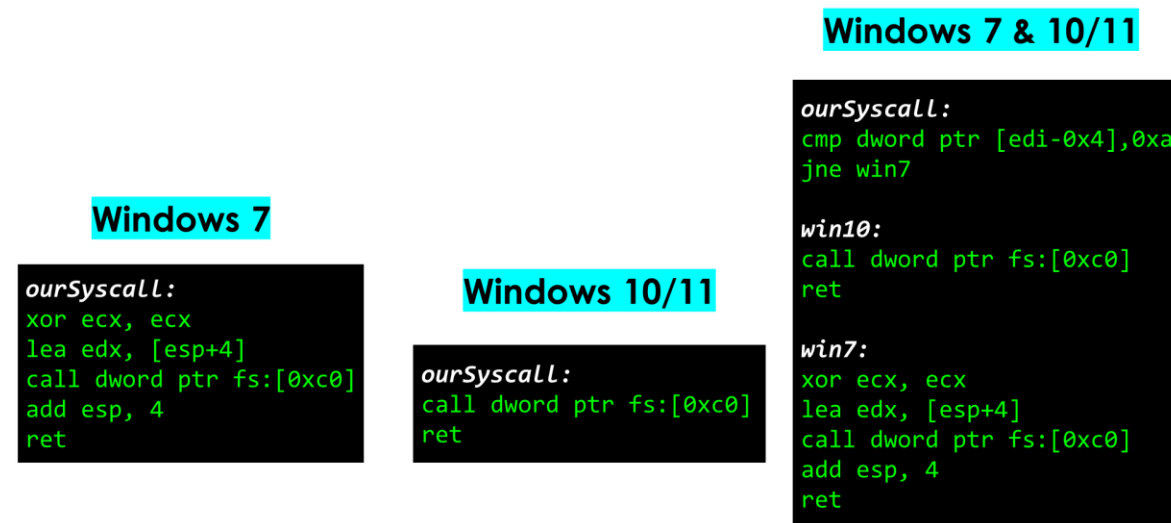


Fig. 10. ShellWasp supports different ways of invoking the syscall, depending on the OS builds that the user targets.

Addressing the Portability Problem with Syscalls in ShellWasp

In order to address the portability problem associated with using Windows syscalls and shellcode, ShellWasp uses the PEB to identify the OS build. It does not check the OS major version or OS minor version, unless the user selects to target multiple incompatible operating systems. Otherwise, it is unnecessary to know the OS major version or OS minor version, and those are instructions that can be omitted from the shellcode. Once the OS build is determined, then ShellWasp will check the targets OS build against the selected OS builds. Once found, then each SSN for each syscall is then pushed onto the stack, creating a syscall array. If it is necessary for ShellWasp to check the OS major version, meaning that multiple incompatible operating systems are targeted, then the value of the OS major version is stored immediately before the syscall array. This syscall array is always pointed to by the edi register, so edi and an offset can always be dereferenced, in order to obtain the correct SSN value.

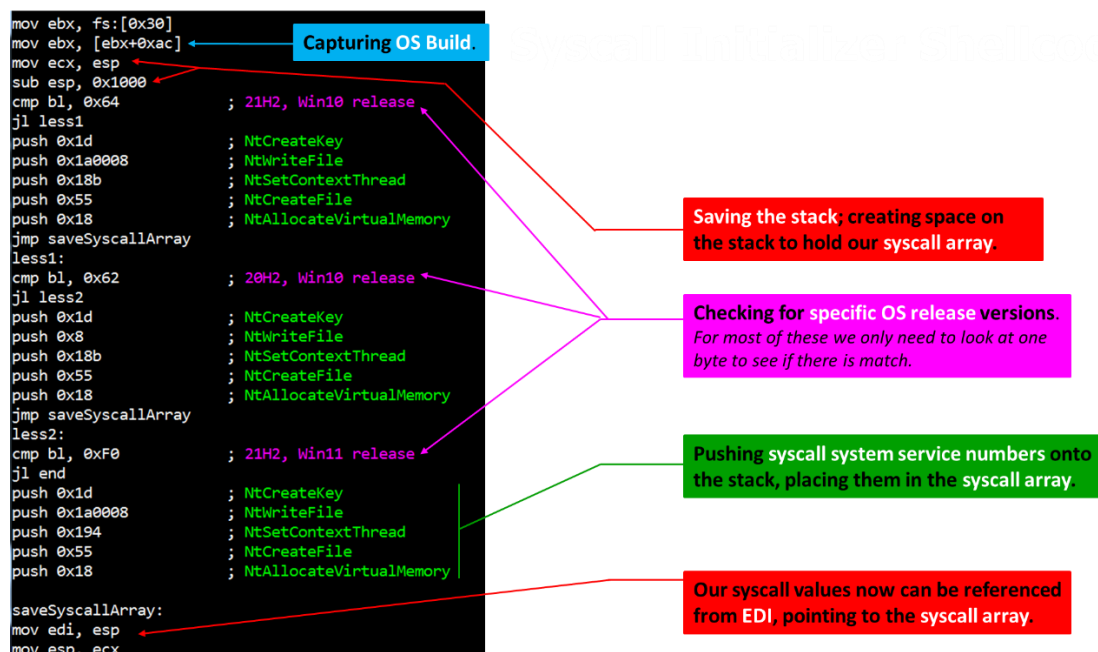


Fig. 91. ShellWasp captures the OS build and then generates the needed syscall array for selected syscalls and select OS builds, all of which are determined by the user.

By pushing SSN's onto the sack, we are able to have them be stored at a predictable location that will be the same, regardless of the OS build. The edi register was selected as a convenient location to always point to this call stack, partly because many people tend to prefer to use other registers in writing shellcode. One problem associated with using edi, is that with each invocation of the syscall, value of edi will be obliterated. Thus, it is necessary to preserve the location in EDI. This is done with a push instruction before the setup of each distinct syscall. Thus, with ShellWasp, there is a continuous saving and restoring of the address of the syscall array and edi. In an earlier version of ShellWasp, there was start cleanup utilizing `add esp, 0xYY` followed by a `pop edi`. While that does indeed work, it was found that with more complex and lengthier syscall shellcodes, there could be some stability problems with respect to the stack, where certain values could be potentially overwritten, even while trying to be careful to avoid doing so. To avoid this, a slightly different approach was taken, so that there is no stack clean up after each syscall. Instead, there is a `mov edi, [esp+0xYY]` instruction. It achieves the same aim as what was done previously, except that the stack values are not unnecessarily cleaned up, as indeed some of them should not be cleaned up. After all, complex structures or parameters potentially could be built on the stack, only to then be overwritten prematurely. Another approach that a user could potentially take is to simply store the pointer to the syscall array it's a predictable location, such as `ebp+0x200`. In some cases that could be more desirable, but from the standpoint of automating this process, it is not possible able to be known that a particular address will always be reliable or not. Therefore, using a register to point to the start of the syscall array seems preferable, as largely we can be assured of its reliability. Another more complex solution would be to create an allocation of memory, such as with the `NtAllocateVirtualMemory` syscall, and for that to be used to point to the syscall array, or to store needed structures or parameters that must be created.

It is important to bear in mind that ShellWasp it's designed only to provide a template for a syscall shellcode, and to automate many aspects of utilizing a syscall array. Thus, ShellWasp will automatically calculate the exact number of bytes to be used to restore the pointer to the syscall array. For instance, eight bytes for two parameters would require an adjustment of eight bytes. However, given the complexity of many syscalls, some users may utilize multiple pushes to create necessary parameters or structures on the stack. This could cause the number of bytes required to be adjusted to change. That is something that a user must be cognizant of and change as necessary.

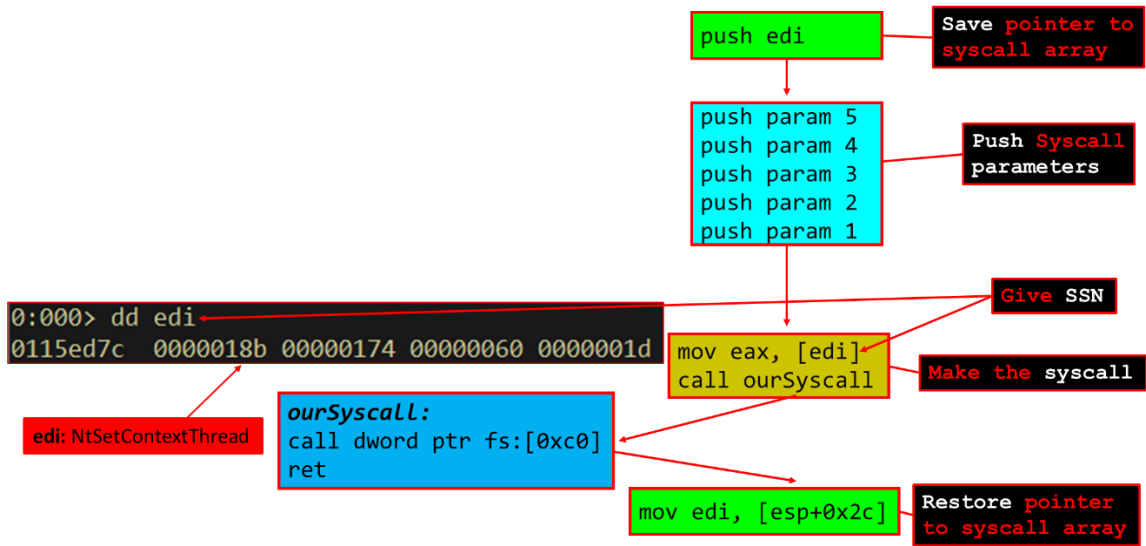


Fig. 10. ShellWasp creates a syscall array to address the portability problem inherent with Windows syscall usage.

The syscall array can then be dereferenced at runtime to provide the appropriate SSN, thereby avoiding the need to hardcode an SSN, as seen below. With this approach, we can achieve true portability, allowing for a syscall shellcode to run across multiple operating systems or OS builds. In actual practice, we can have complex syscall shellcodes that work across multiple operating systems without issue.

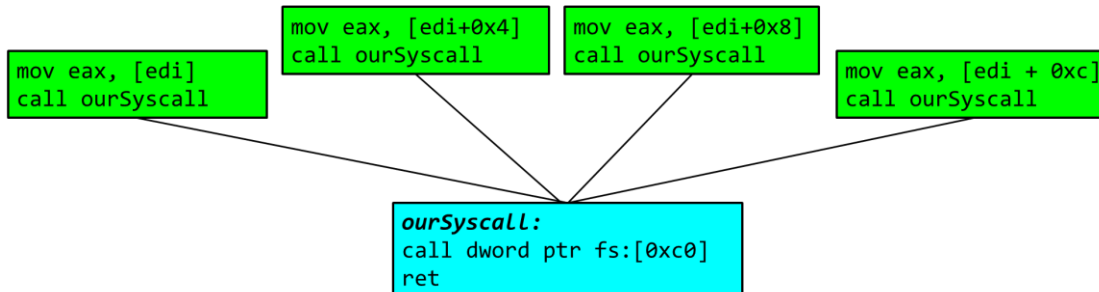


Fig. 13. ShellWasp allows for SSNs to be dereferenced at runtime, to avoid the need for hardcoding SSNs, and then a special function is called, which contains the method of invoking the syscall as relevant for the user's OS build selections open.

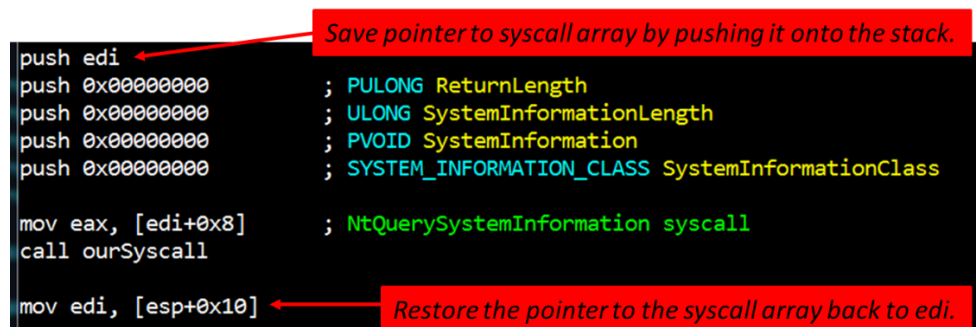
Given that the SSN's are only dereferenced at runtime, and that the Assembly required to do so remains identical, regardless of the particular OS builds being targeted, it is then very easy and simple to change they selected OS releases. In ShellWasp, this is extremely easy to do through the user interface, with only a small number of keystrokes. Alternatively, the user soon configure the selected OS builds with simple Boolean values in the config file. By default, the config file is always loaded, so pre-selected OS builds or syscalls can provide a starting point. Editing either the selected syscalls or OS builds is very simple through the user interface.

3.4 Syscall Prototypes in the Generated ShellWasp Template

It is pragmatic to bear in mind that ShellWasp does not create complete syscall shellcodes, ready to be used with no changes. Such an effort would seem to be impractical and limited only to a relatively small number of prebuilt syscalls. In actual practice, there is so much potential variation that differs greatly based on the needs of the user, that it would be foolhardy to embrace such an approach. The goal with ShellWasp is to automate as much of the process is as possible, and to allow for the user to then concentrate on the more demanding aspects

of building a complex syscall shellcode that has their desired functionality. If a user is trying to do something complex and novel, they may spend considerable time in attempting to do something that may not have been done before.

Thus, ShellWasp can provide an invaluable starting point. The user, for instance, might select to utilize eight syscalls, of which three of them maybe the same, but just reused. ShellWasp manages as much of this as possible, including having only five entries for syscalls, rather than eight. For syscalls used multiple times, the same location can be dereferenced more than once. ShellWasp will also provide the required number of pushes, one for each parameter, and each it has a comment that identifies the parameter type and name. For users this can help avoid simple errors. Additionally, ShellWasp will calculate the required number of bytes needed to be adjusted from esp, to restore edi, although as mentioned before, users should be cognizant of changing this value with additional pushes. It is a trivial matter to adjust the number of bytes to use an offset to restore the pointer to the syscall array, and if there is any difficulty, utilizing a debugger can help determine the number of bytes to adjust. That could be done by noting the address of the syscall array once it is pushed onto the stack.² Once it is time to restore the syscall array, then the difference can be calculated between the current location of esp and the location of the syscall array.



```

push edi
push 0x00000000 ; PULONG ReturnLength
push 0x00000000 ; ULONG SystemInformationLength
push 0x00000000 ; PVOID SystemInformation
push 0x00000000 ; SYSTEM_INFORMATION_CLASS SystemInformationClass

mov eax, [edi+0x8] ; NtQuerySystemInformation syscall
call ourSyscall

mov edi, [esp+0x10]

```

Save pointer to syscall array by pushing it onto the stack.

Restore the pointer to the syscall array back to edi.

Fig. 11. ShellWasp prepares a template that includes a syscall to NtQuerySystemInformation. It has labelled the type and name for each expected syscall parameter, and it calculates the number of bytes to adjust to restore edi.

4 Evaluation of ShellWasp

With original scientific contributions such as ShellWasp, it is necessary to provide some type of validation that conclusively demonstrates the research contribution is valid and effective. Without adequate validation, then it is hard to take a design science research artifact seriously. As such, ShellWasp has been approached from multiple angles to ensure that validation has been achieved.

The first approach to validating ShellWasp just to make sure that it has accurate and correct Assembly within the shellcode. Firstly, the shellcode must be able to compile. Compilation can differ based on how it is performed, but ShellWasp internally uses Keystone, which always generates the necessary bytes that would result from compilation. These bytes are not shown, as it may be deceptive to a novice user, who might mistakenly think the shellcode could be used by exporting the resulting bytes. A series of hexadecimal bytes would not have any value to a user, given that the syscall shellcode template must be customized. It is effective, however, in ensuring correctness and accuracy, because the “compilation” from Keystone would not occur otherwise, and an error message instead would be present. Secondly, we must approach this from the perspective of while it is possible to have Assembly that can compile, it must also do what it intends to do without errors. Thus, the syscall array and its usage have been extensively tested very carefully and analyzed in a debugger, to ensure that it behaves exactly as it intends. Given that a large enough set of samples has been analyzed and found to be free of errors, and given that the same patterns repeat, regardless of input of particular syscalls or OS builds, we can

² Using a debugger to calculate the number of bytes to adjust may be useful in complex syscall shellcodes, where different sizes are pushed onto the stack, as bytes, word, and dwords all could be pushed onto the stack, making it tedious to keep track of each.

be assured as to the correctness and accuracy of the usage and creation of the syscall array. This verification entails looking at all aspects of the syscall array, from its creation, to dereferencing the SSN, etc.

Accuracy and correctness has also been approached from the standpoint of determining the best approach to maintain the pointer to the syscall array. Previously, an alternative way of restoring the syscall array was used, and in many cases it was viable without issue, but it was noticed that with more complex shellcode, such as ones with eight or ten syscalls with many structures and parameters being built on the stack, that this was not necessarily the most reliable way of restoring the pointer to the syscall array. Indeed, the pointer to the syscall array could be overwritten in nonobvious ways. Thus, a more reliable way of restoring this pointer to the syscall array, without actually performing stack cleanup, was devised. Through extensive testing, there is high confidence in the accuracy and correctness of the Assembly that is produced.

Validation of ShellWasp can also be looked at from the perspective of at being able to provide correct information for the syscalls utilized. That is, does it produce the correct number of parameters for each syscall, and are the parameters labeled correctly. ShellWasp uses a Python dictionary or JSO for this purpose I originally generated, after spending multiple days hunting down each syscall, and trying to find what I felt was likely an appropriate function prototype. Given that many Windows syscalls are undocumented, there is an inherent unreliability with function prototypes. That is due to the fact that some may be the result of reverse engineering in good faith by others, so it is possible there could be some minor errors. Differences in nomenclature are also possible, although that is trivial.

The most important aspect of validating ShellWasp is in seeing if it is capable of creating complex and relevant syscall shellcodes. Additionally, one could look at it from whether or not it is able to implement a particular syscall in isolation. The answer to that is yes, there have been numerous syscalls created in isolation for testing purposes. What is more relevant, however, is if multiple syscalls can be combined to achieve a more sophisticated and desirable functionality, and if so, can they work across multiple OS builds or multiple operating systems. Often a single syscall may do only one minor thing, which in isolation is of little practical value. More significant functionality requires a chain of syscalls, with many calling upon various complex structures that must be created.

What follows is a description of different case studies that highlight the successful use of ShellWasp approach to syscall shellcode development. We begin first with a rather sophisticated syscall.

4.1 Case Study for Shellcode to Create a Urlmon.dll in Discord.exe to Inject it with a Second Stage Shellcode.

In this case study, I implement ten different Windows syscalls, many of which require complex set up with multiple complex structures or parameters needing to be created in memory, which in this case was the stack.³ The goal of this shellcode is to enumerate all active processes, find Discord and determine its PID, and then to create a library, Urlmon.dll, which is then used to inject a second stage payload. The original process then must activate the second stage shellcode, which is present in Discord.exe. The syscalls utilized follow:

1. NtAllocateVirtualMemory
2. NtQuerySystemInformation
3. NtOpenProcess
4. NtCreateFile
5. NtCreateSection
6. NtMapViewofSection
7. NtProtectVirtualMemory
8. NtWriteVirtualMemory
9. NtCreateThreadEx
10. NtWaitForSingleObject

I will briefly discuss the purpose of each syscall in greater detail, although for brevity's sake I omit much of the set up of numerous pointers to parameters and structures. NtAllocateVirtualMemory is used to create an

³ Greater reliability could be possible by creating some of the structures or parameters in alternative memory, such as a region allocated by NtAllocateVirtualMemory.

area of memory in the original process. This memory will be used for a large `SystemProcessInformation` structure. Some `SystemProcessInformation`'s can be relatively large, and thus the stack is not well-suited for it. `NtQuerySystemInformation` is an extremely powerful syscall, as it generates the `SystemProcessInformation` specified by the user, of which there are many possibilities. Each provides a wealth of information. This is a function that could be used to retrieve a lot of information found in tools such as Process Explorer. In our case, we wish to identify the PID for `Discord.exe`, as we need that PID to get a handle to it. The resulting `SystemProcessInformation` has many processes, and it is necessary to navigate the results, in order to find a match to our target, `Discord.exe`. This is the most complex part of the shellcode.⁴ Once the PID is obtained, then `NtOpenProcess` is used to obtain a handle to the process. As with many things involving native APIs, there are often numerous handles required. Next, `NtCreateFile` is used to create a handle to `urlmon.dll`. Once that handle is obtained, then a section for `Urlmon.dll` can be created, providing a handle to it. With this handle to the section, then `NtMapViewofSection` can be used to map that section of `Urlmon.dll` into our target external process, `Discord.exe`. However, `Urlmon.dll` will lack `RWX` permissions, meaning if I want to use it to inject the stage two shellcode, I need to change its permissions to `RWX`. I do exactly that by using `NtProtectVirtualMemory`. With the `RWX` permissions, I can then write the second stage shellcode payload into `Urlmon.dll` with `NtWriteVirtualMemory`. With our stage two shellcode now in place, it is necessary to cause that shellcode to begin to execute. That can be achieved by using the `NtCreateThreadEx` syscall, which immediately causes the stage two shellcode to begin to execute inside of the `Discord.exe` process. The stage two shellcode utilizes traditional WinAPIs to produce a simple messagebox as a simple proof of concept. In some cases, with other applications, it might be necessary to cause the thread to begin execution with `NtWaitForSingleObject`. Although it is not needed here, I implement `NtWaitForSingleObject` regardless.

In testing, this syscall works well, and numerous structures and parameters are created to allow highly complex and relatively sophisticated functionality to be achieved. Although there could be simpler ways to do process injection, the goal here was to show off what could be done with syscalls, and so hiding a second stage payload 0x3000 bytes into `Urlmon.dll` was chosen.

It was an adventure in creating this syscall shellcode. It should be noted, however, that while this works flawlessly on Windows 7, and all the syscalls “work” on Windows 10, that there is a special problem with Windows 10. Control Flow Guard (CFG) is able to detect if indirect calls are valid targets for indirect calls. If they are not, then the application with the invalid indirect call immediately terminates. Thus, while debugging `Discord.exe`, I observed that the shellcode would indeed move to the designated location to begin the second stage shellcode. However, it would immediately have an `int 0x29` interrupt from `NtFailFast`. In tracing this, it was discovered that this was due to CFG.

```

(9e64.3a28): Security check failure or stack buffer overrun - code c0000409 (!!! second chance !!!)
eax=00000000 ebx=00000000 ecx=0000000a edx=6a283000 esi=6a283000 edi=6a283000
eip=77058b30 esp=00a5fd80 ebp=00a5fdac iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!RtlFailFast2:
77058b30 cd29             int     29h

```

Fig. 12. CFG prevents some injected shellcode in an external process from starting via `NtCreateThreadEx` in Windows 10.

While this does not have a bearing on the efficacy of `ShellWasp` and our general approach, as all syscalls worked, and the OS just responded appropriately due to the CFG violation, I wanted to try to handle and deal with this CFG problem. In researching how to do this, I found research on a method to bypass CFG that could be applicable. I discovered that the `NtSetInformationVirtualMemory` syscall could be used to create exceptions for CFG, including in external processes, such as CFG. As this was a fairly complex, advanced syscall, with many parameters and structures, this was not a simple task. Moreover, as this was undocumented, it was much less simple. Some research on a blog from 2016 described a WoW64, 32-bit usage of `NtSetInformationVirtualMemory` in a higher-level language[23]. That provided some insights into some undocumented features. However, I would learn later that some internal aspects of the function and its usage seemed to have changed since the blog was written. A more recent article [24] discussed using `NtSetInformationVirtualMemory` in a 64-

⁴ Part of the parsing of the resulting `SystemProcessInformation` was adapted from a previous shellcode Tarek did for this project for our DEFCON presentation.

bit environment, which means there will be differences in a 64-bit environment. Finally, I tried reverse engineering the syscall by discovering the proper WinAPI, `SetProcessValidCallTargets`, to call to achieve the same purpose. The WinAPI is vastly easier to use, as in general is the case with most WinAPIs. I indeed was able to utilize `SetProcessValidCallTargets` to create a CFG exception, to allow this syscall shellcode to execute in a Windows 10 environment, allowing it to spawn the second stage payload, the messagebox. The goal was to get that to work and then use that as a means to reverse engineering and implement the same functionality via syscall. I was not successful in getting `NtSetInformationVirtualMemory` to produce a CFG exception, but I did notice some differences from existing research. I exceeded the amount of time I wanted to spend with `NtSetInformationVirtualMemory`. Likely with more time, it would be possible to use `NtSetInformationVirtualMemory` with Windows 10.

`NtSetInformationVirtualMemory` is a good example of a syscall that is very powerful but not well-documented, and as with a lot of complex syscalls, some reverse engineering and experimentation may be required. While corresponding WinAPIs generally are well documented, the same cannot be said for the native API. However, in many cases reverse engineering can provide solutions that may not otherwise be available.

It is also important to note that an application must be compiled specifically to utilize CFG, and not all applications have been.

4.2 Case Study for Shellcode to Identify Notepad and then Cause it Immediately Terminate: Kill Notepad

This syscall shellcode is a variation on the previous case study. This syscall shellcode only utilizes four syscalls. In its present form it could be regarded as more of a nuisance, although the same syscalls could be used and weaponized further for more destructive purposes. The syscalls utilized are as follows:

- `NtAllocateVirtualMemory`
- `NtQuerySystemInformation`
- `NtOpenProcess`
- `NtTerminateProcess`

With `NtAllocateVirtualMemory` a region of memory is created that can be used to store a `SystemProcessInformation` structure. With `NtQuerySystemInformation`, a `SystemProcessInformation` is created and stored at the newly created allocation of memory. As described in the previous case study, a PID is obtained for the target process, in this case, Notepad. `NtOpenProcess` is then utilized to take the PID and create a handle for Notepad. The handle for Notepad is then used by `NtTerminateProcess` to terminate Notepad.

This syscall shellcode is not extremely malicious in nature, but it could be weaponized further. For instance, it could serve as a second stage payload to be injected into some external process and then started via something such as `NtCreateThreadEx`. This shellcode could then be adapted to run in a loop, so that all instances of a particular process would be terminated ad infinitum until the shellcode itself ends. These extra, more destructive measures were not done, although it would not be hard to do so.

4.3 Case Study for Shellcode to Delete a File

This is an example of a syscall shellcode that could be relatively simple and succinct, and still could achieve some pragmatic functionality. In this shellcode, `NtDeleteFile` is utilized to delete a specified file. A special `OBJECT_ATTRIBUTES` structure first must be created with the relevant information for the target file, such as its file location. Once that structure has been created, then `NtDeleteFile` called to delete the file. It is noteworthy that this syscall method of deleting a file utilizes only one syscall, where traditional ways of deleting files will typically require three syscalls. In some instances, it was observed that the file may be deleted, but it would still appear and Windows explorer with an icon but 0 bytes. Sometimes it would be necessary to refresh for the file to disappear the resulting behavior from `NtDeleteFile` seemed to vary.

4.4 Case Study for Shellcode to Load Urlmon.dll so that UrlDownloadToFileA Could Be Called

In this example, three syscalls were used to map Urlmon.dll into a process. The goal would be to have something comparable to the WinAPI loadlibrary, but via syscalls. While there is a corresponding native API, that function does not have a one-to-one corresponding syscall. Thus, NtCreateFile was used to create a handle to Urlmon.dll. That handle was then used with NtCreateSection, in order to create a section of Urlmon.dll. This section was then mapped into the current process with NtMapViewOfSection. The goal then was to find the exports directory, parse it, and find the address of UrlDownloadToFileA, so that files could then be downloaded, and then perhaps used to create persistence. However, while I did indeed find UrlDownloadToFileA, and I was able to call the WinAPI function, it—and all other functions—would fail at a certain point. I then learned that some addresses need to be “updated” for the DLL to work, even after being mapped as a DLL. While I found WinAPIs that could achieve this, I did not find comparable syscalls, though likely if you reverse engineered the WinAPIs, it is possible (although not known) that there could be multiple syscalls being used. While the syscalls I created worked without issue from a ShellWasp standpoint, ultimately it did not achieve the full functionality I desired. While I could call the function I wanted, the DLL was not mapped into memory the way DLLs normally are.

4.5 Case Study for Shellcode to Cause an Arbitrary WinAPI to be Called with Desired Parameters

In this case study, I used NtContinue to create a Context structure for a current thread. With this Context, I could provide all the values I desired for registers. That included being able to create a fake stack, that I could place in the Context. With this fake stack, I could cause a desired function of my choice to be called with the parameters that I provided, via the fake stack. I was successful with this syscall in the current process. If used in an external process, one would need to determine the address of the target function in an external process, although that of course is easy to do in a current process. Thus, while interesting, causing just one WinAPI to be called with parameters is of limited value in shellcode. In order for this syscall to work, it was necessary to create the somewhat large Context structure, one dword at a time, through a series of push's.

5 Conclusion

Prior to this research, syscalls had virtually never been used in the context of shellcode, outside of egghunters. Given that ShellWasp now provides a clear path forward to create complex Windows syscalls, it seems inevitable that users will begin to use ShellWasp to help create their own syscall shellcodes. With syscalls being rather trendy in the red team community, it seems likely that some of the more skilled and astute will wish to try their hand at invoking syscall functionality in shellcode. After all, while syscalls are certainly popular with red teams, there are also many enthusiasts who wish to experiment with syscalls, just for their own skill development. It seems like a foregone conclusion that syscall shellcode would hold similar allure to such individuals.

The reality of syscall shellcodes is that it takes the difficulty of Windows syscalls and turns it up a notch. Some types of things that can be much more easily and quickly created a higher-level language, will require far greater effort in the context of shellcode. For practical purposes, the fact that syscalls can evade EDR can make them extremely appealing.

Given that there are far fewer Windows syscalls than WinAPIs, what one can do with syscalls is far more limited. However, when taking a more technical and reverse engineering-oriented approach to syscalls, some of those hurdles in terms of limited syscalls could be overcome. After all, some WinAPI functions or even native API's are just a series of multiple syscalls, with the appropriate parameters and structures being passed—sometimes many of the above. Thus, in a sense it could be possible to “unlock” nonobvious functionality by thinking outside the box or tracing syscalls that are called by certain desired API's, and determining how that could be replicated with multiple, equivalent syscalls and appropriate Windows structures.

The future looks bright for syscall shellcode. As this research has shown, ShellWasp can be an extremely effective tool, while providing a novel approach that can help to automate and simplify certain aspects of syscall show code creation. This allows users to concentrate on the more time-consuming aspects of writing shellcode. In conclusion, we can reiterate the primary research contributions are in devising a method to overcome the

portability problem for syscall shellcode, while managing and automating much of how this solution is carried out, with a syscall array. The second research contribution is in generated a tool that can embodies the technique to overcome the syscall array, while also generating a complete, accurate and correct shellcode template, ready to be adapted and deployed.

References

1. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Q. 75–105 (2004)
2. Research, Mds.: Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams, <https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/>
3. Van Eeckhoutte, P.: Windows 10 Egghunter (Wow64) and More, <https://www.corelan.be/index.php/2019/04/23/windows-10-egghunter/>
4. Stevens, D.: Hancitor Maldoc Bypasses Application Whitelisting, <https://isc.sans.edu/forums/diary/Hancitor+Maldoc+Bypasses+Application+Whitelisting/21683/>
5. Gavriel, H.: Malware Mitigation when Direct System Calls are Used, <https://web.archive.org/web/20220204174358/https://www.cyberbit.com/blog/endpoint-security/malware-mitigation-when-direct-system-calls-are-used/>
6. Margaritelli, S.: On Windows Syscall Mechanism and Syscall Numbers Extraction Methods, <https://www.evilssocket.net/2014/02/11/On-Windows-syscall-mechanism-and-syscall-numbers-extraction-methods/>
7. Hutchins, M.: Windows 10 System Call Stub Changes, <https://www.malwaretech.com/2015/07/windows-10-system-call-stub-changes.html>
8. Jurczyk, M.: Windows X86-64 System Call Table (XP/2003/Vista/2008/7/2012/8/10), <https://j00ru.vexillium.org/syscalls/nt/64/>
9. de Plaa, C., Stanhegt: Dumpert, <https://github.com/outflanknl/Dumpert>
10. de Plaa, C.: Red Team Tactics: Combining Direct System Calls and sRDI to bypass AV/EDR, <https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/>
11. ElephantSe4l: FreshyCalls: Syscalls Freshly Squeezed!, <https://github.com/crummie5/FreshyCalls>
12. ElephantSe4l: FreshyCalls: Syscalls Freshly Squeezed!, <https://www.crummie5.club/freshycalls/>
13. T., J.: SysWhispers2, <https://github.com/jthuraisamy/SysWhispers2>
14. KlezVirus: <https://github.com/klezVirus/SysWhispers3>, <https://github.com/klezVirus/SysWhispers3>
15. KlezVirus: SysWhispers is dead, long live SysWhispers!, https://klezvirus.github.io/RedTeaming/AV_Evasion/NoSysWhisper/
16. Laîné, P., Smelly_vx: Hell's Gate. (2020)
17. Reenz0h: Halo's Gate - twin sister of Hell's Gate, <https://blog.sektor7.net/#!/res/2021/halosgate.md>
18. Winter-Smith, P.: FireWalker: A New Approach to Generically Bypass User-Space EDR Hooking, <https://www.mdsec.co.uk/2020/08/firewalker-a-new-approach-to-generically-bypass-user-space-edr-hooking/>
19. Brizendine, B.: ShellWasp, <https://github.com/Bw3ll/ShellWasp>
20. Brizendine, B.J., Babcock, A.: SHAREM. Shellcode Analysis Framework and Disassembler., <https://github.com/brbriz/SHAREM>, (2021)
21. Nowak, T.: The Undocumented Functions Microsoft Windows NT/2000/XP/Win7
22. Storchak, S., Podobry, S.: 1903 19H1 (May 2019 Update) _PEB, [https://www.vergiliusproject.com/kernels/x86/Windows 10/1903 19H1 \(May 2019 Update\)/_PEB](https://www.vergiliusproject.com/kernels/x86/Windows%2010/1903%2019H1%20(May%202019%20Update)/_PEB)
23. Liberman, T.: Undocumenting the Undocumented: Adding CFG Exceptions, <https://www.fortinet.com/blog/threat-research/documenting-the-undocumented-adding-cfg-exceptions>
24. Chamberlain, B.: Sleeping with Control Flow Guard, <https://icebreaker.team/blogs/sleeping-with-control-flow-guard/>