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

itm4n.github.io/vba-runpe-part2

December 29, 2018

In the previous part, I discussed the method used by Didier Stevens to run `cmd.exe` within *Excel* (or *Word*) thanks to a custom shellcode in VBA. I also outlined its limitations. In this part, I'll try to explain how I was able to address them in order to provide a more versatile method that pentesters can easily reuse when required. The code can be found here.

RunPE, what is it?

When I searched the Internet for a full implementation of the RunPE technique in VBA, I was quite surprised to not find anything so I decided to do it myself. But, in my previous post, I didn't really explain what this means. Since I wasn't too familiar with this concept when I first started this small project, I will say a few words about it before going into the technical details.

Disclaimer: this technique has been used by malware authors for many, many years so there is nothing groundbreaking in what I'm going to explain.

RunPE is a trick used by malwares to hide code inside a legit process. The overall idea is to create an instance of a legit process and replace its memory with the content of a malicious PE. Here is a high-level algorithm representing the main steps:

1. **Select a legit process**, `svchost.exe` or `explorer.exe` for example.
2. Start **a new instance** of this process in a `SUSPENDED` state.
3. **Allocate** enough **memory** within this process to **copy the content** of the actual PE to execute.
4. **Adjust the entry point**. The *Image Base* address referenced in the context of the suspended process must be replaced by the one specified in the headers of the PE file.
5. **Resume** the process' main thread.

This technique is widely documented and it has already been implemented in different languages. So, I started by playing around with the one published by ZeroMemory on GiHub. I chose this one because it is written in C++ so the code is very small and efficient for that purpose.

Objectives & Challenges

The objectives of this project are the following:

1. It must be **versatile**. Ideally, we should be able to run any PE file without modifying the entire code.
2. It must be **architecture-independent**. The code must be compatible with both the 32-bit and 64-bit versions of *Microsoft Office*.
3. It must be as **lightweight** as possible. We should be able to copy/paste the code easily to a target machine.

Another thing that I didn"t mention in my previous post is the conversion process from C++ to VBA. In my defense, there wasn't that much to say, apart from the definition of the functions imported from the Windows API. Here, we will see that things will be ***slightly*** different.

Import functions from the Windows API

For our RunPE implementation, we will need the following API functions:

- `RtlMoveMemory()` - to copy the content of a buffer to another;
- `GetModuleFileName()` - to get the path of the current process;
- `CreateProcess()` - to create a new process in a suspended state;
- `GetThreadContext()` - to adjust the entry point of the process;
- `ReadProcessMemory()` - to get the address of the *Image Base* of the process;
- `VirtualAllocEx()` - to allocate memory within the process;
- `WriteProcessMemory()` - to write the content of a PE file to the allocated memory;
- `SetThreadContext()` - to apply the changes made to the entry point;
- `ResumeThread()` - to resume the execution flow;
- `TerminateProcess()` - to terminate the suspended process in case of failure.

Let's take a closer look at the `CreateProcess()` function for example. Here is how it is defined according to the Microsoft documentation.

```
BOOL CreateProcessA (
  LPCSTR
lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES
lpProcessAttributes,
  LPSECURITY_ATTRIBUTES
lpThreadAttributes,
  BOOL
bInheritHandles,
  DWORD
dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR
lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION
lpProcessInformation
);
```

We can see that it requires different Win32 types such as `LPSTR` but it also requires complex structures such as `PROCESS_INFORMATION`. Remember, VBA is *not aware* of these types and structures. So, we will have to define them manually if we want to use this function.

Dealing with basic Win32 types such as `LPSTR` will be pretty straightforward. Here is a non exhaustive correlation table between some Win32 and VBA types.

C++ (Win32)	VBA	Arch
BOOL	Boolean	32 & 64
BYTE	Byte	32 & 64
WORD	Integer	32 & 64
DWORD, ULONG, LONG	Long	32 & 64
DWORD64	LongLong	64
LPSTR, LPCSTR	String	32 & 64
HANDLE, LPBYTE, LPVOID	LongPtr	32 & 64

We are starting to see that we are going to face some issues with the process architecture (32/64 bits). Namely, the `LongLong` type only exists in the 64-bit version of *Office* for example.

Actually, once we know how to convert basic Win32 types to VBA types, a major part of the work is done because VBA enables us to define our own structures using the following syntax: `Private Type [...] End Type`.

For example, the `PROCESS_INFORMATION` structure is defined as follows in the Windows API:

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD  dwProcessId;
  DWORD  dwThreadId;
} PROCESS_INFORMATION, *PPROCESS_INFORMATION,
*LPPROCESS_INFORMATION;
```

In VBA, it would thus be defined as:

```
Private Type
PROCESS_INFORMATION
    hProcess As LongPtr
    hThread As LongPtr
    dwProcessId As Long
    dwThreadId As Long
End Type
```

Easy! Right?! Well… …wait a minute. Let’s take a look at `IMAGE_NT_HEADERS`:

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER32
OptionalHeader;
} IMAGE_NT_HEADERS32,
*PIMAGE_NT_HEADERS32;
```

It uses two other structures: `IMAGE_FILE_HEADER` and `IMAGE_OPTIONAL_HEADER32`. This means that we will have to define all the structures we need recursively until only basic types are used. It isn't that complicated actually but it's a **very tedious** task.

An architecture-independent code…

Earlier, I mentionned that we could face some issues with the architecture of the running process. Spoiler alert: we will…

1) Functions

Let's warm up with a problem that we will be able to address quickly: functions are not defined the same way in 32-bit and 64-bit. Here is an example with the `ResumeThread()` function.

```
' 64-bit mode
Private Declare PtrSafe Function ResumeThread Lib "KERNEL32" (ByVal hThread As
LongPtr) As Long
' 32-bit mode
Private Declare Function ResumeThread Lib "KERNEL32" (ByVal hThread As Long) As
Long
```

We must add the `PtrSafe` safe keyword in the declaration if *Office* runs in 64-bit mode. Apart from that, there is no difference. Fortunately for us, VBA has Conditional Compilation Constants that will enable us to use these two declarations in the same code:

```
#If Win64 Then
  Private Declare PtrSafe Function ResumeThread Lib "KERNEL32" (ByVal hThread As
LongPtr) As Long
#Else
  Private Declare Function ResumeThread Lib "KERNEL32" (ByVal hThread As Long) As
Long
#End If
```

2) Structures

The content of some Win32 structures wil vary depending on the process architecture. The most obvious one is `CONTEXT`. It's the one that holds the state of a process, namely the value of each processor register. Once again, the `Win64` Conditional Compilation Constant will enable us to use two different declarations of the same structure *dynamically*.

In addition, the `LongPtr` type in VBA will help us a lot. This type was added by *Microsoft* in *Office 2010*. Its size will adapt to the running process. If *Office* runs in 32-bit mode, it will be 32-bit else if *Office* runs in 64-bit mode, it will be 64-bit. This is particularly helpful when dealing with pointers such as the common *Windows* `HANDLE`. Whenever a pointer is needed, we can use this type without worrying too much about how it is handled internally.

3) PE files

And, here comes the tricky part…

If *Office* is running in 32-bit mode, we will be able to inject 32-bit PE files only. On the other hand, if *Office* is running in 64-bit mode, we will be able to inject 64-bit PE files only. This makes sense, right?!

I started to think about ways to overcome this limitation by implementing a cross-architecture code but this seems to require a lot of effort for a very little gain. In addition, I don't even know whether it would be possible to do so in VBA.

Bring all the pieces together

The first main method I implemented is `RunPE()`. It contains all the logic behind the RunPE algorithm. It takes two arguments: the content of the PE file to inject as a `Byte Array` and a `String` representing the command line arguments.

The second main method I implemented is `Exploit()`. This name is not very well chosen since it's not an *exploit*. However it's the method that you want to customize to fit your needs. Indeed, that's where you can specify the path of the PE file to inject along with the command line arguments I mentionned previously. I hardcoded the following values because it's the most standard use case.

```
strSrcFile =
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
strArguments = "-exec Bypass"
```

The PE file is read and its content is converted to a `Byte Array`. This array is then passed as an argument to the `RunPE()` method along with the command line arguments.

```
baFileContent =
FileToByteArray(strSrcFile)
Call RunPE(baFileContent,
strArguments)
```