t runtime...

#1

der, shellcode, without runtime...

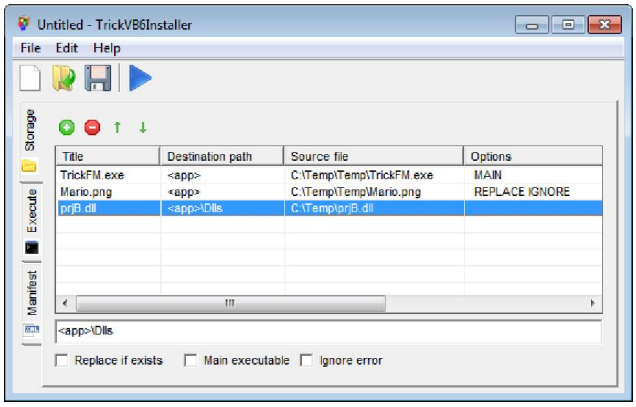
Hello everyone! Today i want to show you the quite interesting things. One day i was investigating the PE (portable executable) file format especially EXE. I decided to create a simple loader of the executable files specially for VB6-compiled applications. This loader should load an VB6-compiled exe from the memory without file. THIS IS JUST FOR THE EXPERIMENTAL PURPOSES IN ORDER TO CHECK POSSIBILITIES OF VB6. Due to that the VB6-compiled applications don't used most of the PE features it was quite simple objective. Most of programers says that a VB6-apllication is linked with the VB6 runtime (MSVBVM), a VB6 application doesn't work without the runtime and the runtime is quite slow. Today i'll prove that it is possible to write an application that absolutely doesn't use runtime (although i was already doing that in the driver). These projects i had written quite a long time ago, but these were in the russian language. I think it could be quite interesting for someone who wants to examine the basic principles of work with the PE files.
Before we begin i want to say couple words about the projects. These projects were not tested well enough therefore it can cause problems. The loader doesn't support most of the features of PE files therefore some executables may not work.
So...
This overview consists three projects:

1. Compiler - it is the biggest project of all. It creates an installation based on the loader, user files, commands and manifest;
2. Loader - it is the simple loader that performs commands, unpacks files and runs an executable from memory;
3. Patcher - it is the small utility that removes the runtime import from an executable file.

I call an exe that contains the commands, files and executable file the installation. The main idea is to put the information about an installation to the resources of the loader. When the loader is being loaded it reads the information and performs the commands from resources. I decided to use an special storage to save the files and exe, and other storage for commands.
The first storage stores all the files that will be unpacked, and the main executable that will be launched. The second storage stores the commands that will be passed to the ShellExecuteEx function after unpacking process will have been completed. The loader supports the following wildcards (for path):

1. <app> - application installed path;
2. <win> - system windows directory;
3. <sys> - System32 directory;
4. <drv> - system drive;
5. <tmp> - temporary directory;
6. <dtp> - user desktop.

Compiler.

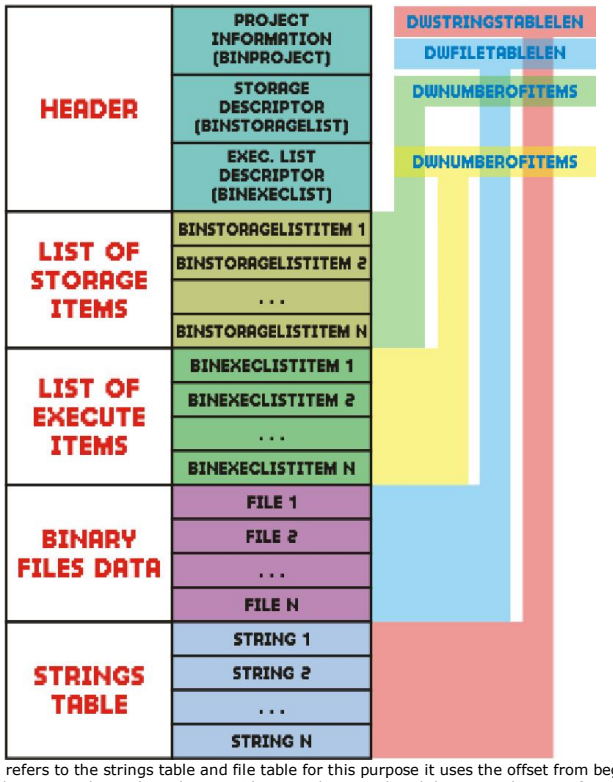


This is the application that forms the installation information and puts it to the loader resource. All the information is stored in a project. You can save and load a project from file. The **clsProject** class in VB project represents the compiler-project. This compiler has 3 sections: storage, execute, manifest.
The '*storage*' section allows to add the files that will be copied when the application is being launched. Each item in the list has flags: 'replace if exists', 'main executable', 'ignore error'. If you select 'replace if exists' flag a file will be copied even if one exists. The 'main executable' flag can be set only for the single executable file. It means that this file will be launched when all the operations have been performed. The 'ignore error' flag makes ignore any errors respectively. The order in the list corresponds the order of extracting the files except the main executable. The main executable is not extracted and is launched after all the operations. The storage section is represented as **clsStorage** class in the VB project. This class implements the standard collection of the **clsStorageItem** objects and adds some additional methods.The **MainExecutable** property determines the index of main executable file in the storage. When this parameter equal -1 executable file is not presented. The **clsStoragaItem** class represent the single item in the storage list. It has some properties that determine the behavior of item. This section is helpful if you want to copy files to disk before execution of the application.
The next section is the '*execute*'. This section allows execute any commands. This commands just pass to **ShellExecuteEx** function. Thus you can register libraries or do something else. Each item in the execution list has two properties: the executable path and parameters. Both the path and the parameters is passed to **ShellExecuteEx** function. It is worth noting that all the operations is performed synchronously in the order that set in the list. It also has the 'ignore error' flag that prevents appearance any messages if an error occurs. The execute section is represented as two classes: **clsExecute** and **clsExecuteItem**. These classes are similar to the storage classes.
The last section is '*manifest*'. It is just the manifest text file that you can add to the final executable. You should check the checkbox 'include manifest' in the 'manifest' tab if you wan to add manifest. It can be helpful for Free-Reg COM components or for visual styles.
All the classes refer to the project object (**clsProject**) that manages them. Each class that refers to project can be saved or loaded to the **PropertyBag** object. When a project is being saved it alternately saves each entity to the property bag, same during loading. It looks like a **IPersistStream** interface behavior. All the links to the storage items in the project is stored with relative paths (like a VB6 .vbp file) hence you can move project folder without issues. In order to translate from/to relative/absolute path i used **PathRelativePathTo** and **PathCanonicalize** functions.
So... This was basic information about compiler project. Now i want to talk about compilation procedure. As i said all the information about extracting/executing/launching is stored to the loader resources. At first we should define the format of the data. This information is represented in the following structures:

Code:

```
' // Storage list item
Private Type BinStorageListItem
    ofstFileName        As Long             ' // Offset of file name
    ofstDestPath        As Long             ' // Offset of file path
    dwSizeOfFile        As Long             ' // Size of file
    ofstBeginOfData     As Long             ' // Offset of beginning data
    dwFlags             As FileFlags        ' // Flags
End Type

' // Execute list item
Private Type BinExecListItem
    ofstFileName        As Long             ' // Offset of file name
    ofstParameters      As Long             ' // Offset of parameters
    dwFlags             As ExeFlags         ' // Flags
End Type

' // Storage descriptor
Private Type BinStorageList
    dwSizeOfStructure   As Long             ' // Size of structure
    iExecutableIndex    As Long             ' // Index of main executable
    dwSizeOfItem        As Long             ' // Size of BinaryStorageItem structure
    dwNumberOfItems     As Long             ' // Number of files in storage
End Type

' // Execute list descriptor
Private Type BinExecList
    dwSizeOfStructure   As Long             ' // Size of structure
    dwSizeOfItem        As Long             ' // Size of BinaryExecuteItem structure
    dwNumberOfItems     As Long             ' // Number of items
End Type

' // Base information about project
```

The 'BinProject' structure is located at beginning of resource entry. Notice that project is stored as **RT_RCDATA** item with 'PROJECT' name. The **dwSizeOfStructure** field defines the size of the **BinProject** structure, **storageDescriptor** and **execListDescriptor** represent the storage and execute descriptors respectively. The **dwStringsTableLen** field shows the size of strings table. The strings table contains all the names and commands in the unicode format. The **dwFileTableLen** field shows the size of all data in the storage. Both storage (**BinStorageList**) and execute list (**BinExecList**) have **dwSizeOfItem** and **dwSizeOfStructure** fields that define the size of a descriptor structure and the size of a list item. These structures also have **dwNumberOfItems** field that shows how many items is contained in the list. The '**iExecutableIndex**' field contains the index of executable file that will be launched. The common structure of a project in the resources is shown in this figure:



An item can refers to the strings table and file table for this purpose it uses the offset from beginning of a table. All the items is located one by one. Okay, you have explored the internal project format now i tell how can you build the loader that contains these data. As i said we store data to resources of the loader. I will tell about the loader a little bit later now i want to note one issue. When you put the project data to resources it doesn't affect to exe information. For example if you launch this exe the information contained in the resources of the internal exe won't be loaded. Same with icons and version information. You should copy the resources from the internal exe to loader in order to avoid this troubles. WinAPI provides the set of the functions for replacing resources. In order to obtain the list of resources you should parse the exe file and extract data. I wrote the 'LoadResources' function that extract all the resources of specified exe data to array.

(rate this post) Reply Reply With Quote

May 22nd, 2016, 06:50 PM #2

The trick

Thread Starter
Fanatic Member

Join Date: Feb 2015
Location: Russia, Astrakhan
Posts: 842

Re: [VB6] Loader, shellcode, without runtime...

PE format.

In order to obtain resources from an exe file, run EXE from memory and well know the stucture of an exe file we should examine the PE (portable executable) format. The PE format has the quite complex structure. When loader launches a PE file (exe or dll) it does quite many work. Each PE file begins with special structure **IMAGE_DOS_HEADER** aka. dos stub. Because both DOS and WINDOWS applications have exe extension you can launch an exe file in DOS, but if you try to do it DOS launches this dos stub. Usually it show the message: "This program cannot be run in DOS mode", but you can write any program:

```
C:\>TRICKC~1
This program cannot be run in DOS mode.

C:\>
```

Code:

```
Type IMAGE_DOS_HEADER
    e_magic                     As Integer
    e_cblp                      As Integer
    e_cp                        As Integer
    e_crlc                      As Integer
    e_cparhdr                   As Integer
    e_minalloc                  As Integer
    e_maxalloc                  As Integer
    e_ss                        As Integer
    e_sp                        As Integer
    e_csum                      As Integer
    e_ip                        As Integer
    e_cs                        As Integer
    e_lfarlc                    As Integer
    e_ovno                      As Integer
    e_res(0 To 3)               As Integer
    e_oemid                     As Integer
    e_oeminfo                   As Integer
    e_res2(0 To 9)              As Integer
    e_lfanew                    As Long
End Type
```

Since we don't write a dos program it doesn't matter. We wonder only two fields of this structure: **e_magic** and **e_lfanew**. The first field should contains the 'MZ' signature aka. **IMAGE_DOS_SIGNATURE** and **e_lfanew** offset to very crucial structure **IMAGE_NT_HEADERS** described as:

Code:

```
Type IMAGE_NT_HEADERS
    Signature                   As Long
```

```
    FileHeader                  As IMAGE_FILE_HEADER
    OptionalHeader              As IMAGE_OPTIONAL_HEADER
End Type
```

The first filed of this structure contains the signature 'PE' (aka. IMAGE_NT_SIGNATURE). The next field describes the executable file:

Code:

```
Type IMAGE_FILE_HEADER
    Machine                     As Integer
    NumberOfSections            As Integer
    TimeDateStamp               As Long
    PointerToSymbolTable        As Long
    NumberOfSymbols             As Long
    SizeOfOptionalHeader        As Integer
    Characteristics             As Integer
End Type
```

The 'Machine' field defines the processor architecture and should have the **IMAGE_FILE_MACHINE_I386** value in our case. The **NumberOfSections** filed determines the count of sections contained in the exe file.

> An exe file contains the sections anyway. Each section takes a place in the address space and optionally in file. A section can contain the code or data (initialized or not), has the name as well. The most common names: .text, .data, .rsrc. Usually, the .text section contains the code, the .data section initialized data and .rsrc - resources. You can change this behavior using the linker directives. Each section have address called virtual address. Generally there are several types of the addresses. The first is relative virtual address (RVA). Because of a PE file can be loaded to any address all the references inside the PE file use the relative addressing. RVA is the offset from beginning of the base address (the address of the first byte of the PE module in the memory). The sum of the RVA and the base address is the VA (the virtual address). Also, there is the raw offset addressing that shows the location of data in the file relative the RVA. Notice that RVA <> raw offset. When a module is being loaded each secion is placed to its address. For example a module could have the section that has no-initialized data. This section wouldn't take a place in the exe file but would occupy the address space. It is the very crucial aspect because we will work with the raw exe file.

The 'TimeDateStamp' field contains the creation date of the PE module in UTC format. The 'PointerToSymbolTable' and 'NumberOfSymbols' contain the information about symbols in the PE file. Generally this fields contains zero. This fields is always used in object files (*.OBJ, *.LIB) in order to resolve links during linking as well as debugging information for PE modules. The next field 'SizeOfOptionalHeader' contains the size of structure following after **IMAGE_FILE_HEADER** named **IMAGE_OPTIONAL_HEADER** that always is presented in PE files (although may be missing in OBJ files). This structure is very essential for loading a PE file to memory. Notice that this structure is different in x64 and x86 executable files. Eventually, the 'Characteristics' field contains the PE attributtes.

The **IMAGE_OPTIONAL_HEADER** structure has the following format:

Code:

```
Type IMAGE_OPTIONAL_HEADER
    Magic                       As Integer
    MajorLinkerVersion          As Byte
    MinorLinkerVersion          As Byte
    SizeOfCode                  As Long
    SizeOfInitializedData       As Long
    SizeOfUnitializedData       As Long
    AddressOfEntryPoint         As Long
    BaseOfCode                  As Long
    BaseOfData                  As Long
    ImageBase                   As Long
    SectionAlignment            As Long
    FileAlignment               As Long
    MajorOperatingSystemVersion As Integer
    MinorOperatingSystemVersion As Integer
    MajorImageVersion           As Integer
    MinorImageVersion           As Integer
    MajorSubsystemVersion       As Integer
    MinorSubsystemVersion       As Integer
    W32VersionValue             As Long
    SizeOfImage                 As Long
    SizeOfHeaders               As Long
    CheckSum                    As Long
    SubSystem                   As Integer
    DllCharacteristics          As Integer
    SizeOfStackReserve          As Long
    SizeOfStackCommit           As Long
    SizeOfHeapReserve           As Long
    SizeOfHeapCommit            As Long
    LoaderFlags                 As Long
    NumberOfRvaAndSizes         As Long
    DataDirectory(15)           As IMAGE_DATA_DIRECTORY
```

The first field contains the type of the image (x86, x64 or ROM image). We consider only **IMAGE_NT_OPTIONAL_HDR32_MAGIC** that represents a common 32-it application. The next two fields is not important (they were used in the old systems) and contain 4. The next group of fields contains the sizes of all the code, initialized data and uninitialized data. These values should be a multiple of 'SectionAlignment' of the structure (see later). The 'AddressOfEntryPoint' is very important RVA-value that sets the start point of a program (look like Sub Main). We will use this field when we have already loaded the PE-image to memory and it is necessary to run the program. The next crucial filds is 'ImageBase' that sets the prefered address of loading the module. When a loader is loading a module it tries to load it to thr prefered address (set in the 'ImageBase' filed). If this address is occupied then the loader checks the 'Characteristics' field in the 'IMAGE_FILE_HEADER' structure. If this field contains the **IMAGE_FILE_RELOCS_STRIPPED** it means that the module can't be loaded. In order to load such module we should add the reloaction information to PE that allows to set up the addresses if the module can't be loaded to prefered base address. We will use this field in the shellcode with the 'SizeOfImage' fields to reserve a memory region for the unpacked PE. The 'SectionAlignment' and 'FileAlignment' set the align in memory and file respectevely. By changing the file alignment we can reduce the PE file size but a system can not be possible to load this modified PE. The section alignment sets the align of a section (usually it equals to page size). The 'SizeOfHeaders' value sets the size of all headers (DOS header, NT headers, sections headers) aligned to the 'FileAlignment' value. The 'SizeOfStackReserve' and 'SizeOfStackCommit' determine the total and initial stack size. It's the same for 'SizeOfHeapReserve' and 'SizeOfHeapCommit' fields but for the heap. The 'NumberOfRvaAndSizes' fields contains the number of items in 'DataDirectory' array. This field always set to 16. The 'DataDirectory' array is very important because it contains the several data catalogs that contain essential information about import, export, resources, relocations, etc. We use only few items from this catalog that is used by VB6 compiler. I'll say about catalogs little bit later let's look what is behind the catalogs. There is list of the section headers. The number of section, if you remember, we can obtain from the **IMAGE_FILE_HEADER** structure. Let's consider the section header structure:

Code:

```
Type IMAGE_SECTION_HEADER
    SectionName(7)              As Byte
    VirtualSize                 As Long
    VirtualAddress              As Long
    SizeOfRawData               As Long
    PointerToRawData            As Long
    PointerToRelocations        As Long
    PointerToLinenumbers        As Long
    NumberOfRelocations         As Integer
    NumberOfLinenumbers         As Integer
    Characteristics             As Long
End Type
```

The first field contains the name of the section in the UTF-8 format with the NULL-terminated characters. This name is limited by 8 symbols (if a section has the size of the name equals 8 the NULL terminated character is skipped). A COFF file may have the name greater than 8 characters in this case the name begins with '/' symbol followed by the ASCII decimal representation offset in the string table (**IMAGE_FILE_HEADER** field). A PE file doesn't support the long names. The 'VirtualSize' and 'VirtualAddress' fields contain the size of section in memory (the address is set as RVA). The 'SizeOfRawData' and 'PointerToRawData' contain the data location in the file (if section has an initialized data). This is the key moment because we can calculate the raw offset by the virtual address using the

information from the section headers. I wrote function for it that translate a RVA address to the RAW offset in the file:

Code:

```
' // RVA to RAW
Function RVA2RAW(
                 ByVal rva As Long,
                 ByRef sec() As IMAGE_SECTION_HEADER) As Long
    Dim index As Long

    For index = 0 To UBound(sec)

        If rva >= sec(index).VirtualAddress And _
           rva < sec(index).VirtualAddress + sec(index).VirtualSize Then
            RVA2RAW = sec(index).PointerToRawData + (rva - sec(index).VirtualAddress)
            Exit Function
        End If

    Next

    RVA2RAW = rva

End Function
```

This function enumerates all the sections and checks if the passed address is within section. The next 5 fields are used in only COFF file and don't matter in a PE file. The 'Characteristics' field contains the attributes of section such as memory permissions and managment. This field is important as well. We will use this information for the memory protection of an exe file in the loader.

(rate this post) Reply Reply With Quote

May 22nd, 2016, 06:57 PM #3

The trick

Thread Starter
Fanatic Member

Join Date: Feb 2015
Location: Russia, Astrakhan
Posts: 842

Re: [VB6] Loader, shellcode, without runtime...

Okay, let's return to the data directories. As we saw there is 16 items in this catalog. Usually, a PE file doesn't use all of them. Let's consider the structure of single item:

Code:

```
Private Type IMAGE_DATA_DIRECTORY
    VirtualAddress                  As Long
    Size                            As Long
End Type
```

This structure consist two fields. The first fields contains the offset (RVA) to the data of the catalog, second - size. When an item of data catalog is not required it contains zero in the both fields. An VB6-compiled application generally contains only 4 items: import table , resource table , bound import table and import address table (IAT). Now we consider the resource table that has the **IMAGE_DIRECTORY_ENTRY_RESOURCE** index, because we work with this imformation in the compiler application.
All the resources in the exe file are represented as the tree with triple depth. The first level defines the resource type (RT_BITMAP, RT_MANIFEST, RT_RCDATA, etc.), the next level defines the resource identifier, the third level defines language. In the standard resource editor you can change the first two levels. All the resources placed in the resources table located in a '.rsrc' section of the exe file.
Regarding the resources structure we even can change it after compilation. In oredr to access to resources we should read the **IMAGE_DIRECTORY_ENTRY_RESOURCE** catalog from the optional header. The 'VirtualAddress' field of this structure contains the RVA of the resource table which has the following structure:

Code:

```
Type IMAGE_RESOURCE_DIRECTORY
    Characteristics             As Long
    TimeDateStamp               As Long
    MajorVersion                As Integer
    MinorVersion                As Integer
    NumberOfNamedEntries        As Integer
    NumberOfIdEntries           As Integer
End Type
```

This structure describes all the resources in PE file. The first four fields are not important, the 'NumberOfNamedEntries' and 'NumberOfIdEntries' contain the number of named items and the items with the numerical id respectively. For instance, when you add an image in 'VB Resource Editor' it'll add a numerical item with id = 2 (**RT_BITMAP**). The items are after this structure and have the following form:

Code:

```
Type IMAGE_RESOURCE_DIRECTORY_ENTRY
    NameId                      As Long
    OffsetToData                As Long
End Type
```

The first field of this structure defines either an item name or an item id depending on the most significant bit. If this bit is set the remained bits show the offset from beginning of resources to **IMAGE_RESOURCE_DIR_STRING_U** structure that has the following format:

Code:

```
Type IMAGE_RESOURCE_DIR_STRING_U
    Length                      As Integer
    NameString                  As String
End Type
```

Note that this is not the proper VB structure is shown for descriptive reasons. The first two bytes is the unsigned short (the closest analog is Integer) that show the length of the unicode string (in characters) that follows them. Thus, in order to obtain a string we should read the first two bytes to an integer, allocate memory for the string with the read value size, and read the remaining data to the string variable. Conversely, if the most significant bit of the 'NameId' field is cleared the field containts an identifier (RT_BITMAP in the previous case). The 'OffsetToData' field has two interpretations too. If the MSB is set it is the offset (from beginnig of resources) to the next level of tree i.e. to an **IMAGE_RESOURCE_DIRECTORY** structure. Otherwise, if the MSB is cleared this is the offset (from beginning of resources too) to a "leave" of the tree, i.e. to structure **IMAGE_RESOURCE_DATA_ENTRY**:

Code:
```
Type IMAGE_RESOURCE_DATA_ENTRY
    OffsetToData                As Long
    Size                        As Long
    CodePage                    As Long
    Reserved                    As Long
End Type
```

The most important fields of this structure are 'OffsetToData' and 'Size' that contain the RVA and Size of the raw data of the resource. Now we can get all the resources from a PE file.

Compilation.

So... When you start the compilation it calls the **Compile** function. Firstly it packs all the storage items and execute items to binary format (BinProject, BinStorageListItem, etc.) and forms the string table and the files table. The string table is saves as the unicode strings with the null-terminating characters. I use special class **clsStream** for safe working with binary data. This class allows to read/write any data or streams into the binary buffer, compress buffer. I use **RtlCompressBuffer** function for compression stream that uses LZ-compression method. After packing and compression it check output project format. It is supports two formats: bin (raw project data), and exe (loader). The binary format is not interesting right now, we will consider the exe format. Firstly, it extracts all the resources from the main executable to the three-level catalog. This operation is performed by **ExtractResources** function. An identifier name is saved as the "#" symbol with the appended string that represents the resource id in the decimal format. Afterwards it clones the loader template to the resulting file, then begins to modify the resources in this file using **BeginUpdateResource** api. Then it alternately copies all the extracted resources (**UpdateResource**), binary project and mainfest (if needed) to the resulting file and applies changes with **EndUpdateResource** function. Again, the binary project is saved with "PROJECT" name and RT_DATA type. Basically that's it.

(rate this post) Reply | Reply With Quote

May 22nd, 2016, 07:01 PM #4

The trick
Thread Starter
Fanatic Member

Join Date: Feb 2015
Location: Russia, Astrakhan
Posts: 842

Re: [VB6] Loader, shellcode, without runtime...

Loader.

So... I think it the most interesting part. So, we must avoid the usage of the runtime. How to do it? I give some rules:

1. Set an user function as startup;
2. Avoid any objects in project;
3. Avoid immediate arrays. The fixed arrays in a type is not forbidden;
4. Avoid string variables as well Variant/Object. In some cases Currency/Date;
5. Avoid the API functions with Declare statement.
6. Avoid VarPtr and some other standard functions.
7. ....

It isn't the complete list of restrictions and during the shellcode execution it adds additional restrinctions.
So, begin. In order to avoid the usage of a string variable i keep all the string variables as Long pointer to the string. There is an issue with loading of a string because we can't access to any string to load it. I decide to use resources as the storage of strings and load it by ID. Thus, we can save the pointer to a string into a Long variable without references to runtime. I used a TLB (type library) for all the API functions without **usesgetlasterror** attribute in order to avoid the Declare statement. In order to set a startup function i use the linker options. The startup function in the loader is 'Main'. Note, if in the IDE you select the startup function 'Main' actually this function is not startup because any EXE application written in VB6 begins with **ThunRTMain** function, that loads project and initialize runtime and thread. Generally, the loader conist three modules:

1. modMain - startup function and working with storage/execute items;
2. modConstants - working with string constants;
3. modLoader - loader of EXE files.

When loader has been launched it starts the Main function.

Code:
```
' // Startup subroutine
Sub Main()

    ' // Load constants
    If Not LoadConstants Then
        MessageBox 0, GetString(MID_ERRORLOADINGCONST), 0, MB_ICONERROR Or MB_SYSTEMMODAL
        GoTo EndOfProcess
    End If

    ' // Load project
    If Not ReadProject Then
        MessageBox 0, GetString(MID_ERRORREADINGPROJECT), 0, MB_ICONERROR Or MB_SYSTEMMODA
        GoTo EndOfProcess
    End If

    ' // Copying from storage
    If Not CopyProcess Then GoTo EndOfProcess

    ' // Execution process
    If Not ExecuteProcess Then GoTo EndOfProcess

    ' // If main executable is not presented exit
    If ProjectDesc.storageDescriptor.iExecutableIndex = -1 Then GoTo EndOfProcess

    ' // Run exe from memory
    If Not RunProcess Then
        ' // Error occrurs
        MessageBox 0, GetString(MID_ERRORSTARTUPEXE), 0, MB_ICONERROR Or MB_SYSTEMMODAL
    End If

EndOfProcess:
```

Firstly, it function call **LoadConstants** function to load all the needed constants from resources:

Code:
```
' // modConstants.bas - main module for loading constants
' // © Krivous Anatoly Anatolevich (The trick), 2016

Option Explicit
```

```
Public Enum MessagesID
    MID_ERRORLOADINGCONST = 100     ' // Errors
    MID_ERRORREADINGPROJECT = 101   '
    MID_ERRORCOPYINGFILE = 102      '
    MID_ERRORWIN32 = 103            '
    MID_ERROREXECUTELINE = 104      '
    MID_ERRORSTARTUPEXE = 105       '
    PROJECT = 200                   ' // Project resource ID
    API_LIB_KERNEL32 = 300          ' // Library names
    API_LIB_NTDLL = 350             '
    API_LIB_USER32 = 400            '
    MSG_LOADER_ERROR = 500
End Enum

' // Paths

Public pAppPath  As Long            ' // Path to application
Public pSysPath  As Long            ' // Path to System32
Public pTmpPath  As Long            ' // Path to Temp
Public pWinPath  As Long            ' // Path to Windows
Public pDrvPath  As Long            ' // Path to system drive
Public pDtpPath  As Long            ' // Path to desktop

' // Substitution constants

Public pAppRepl  As Long
```

The 'LoadConstants' function loads all the needed variables and string (hInstance, LCID, command line, wildcards, default paths, etc.). All the strings is stored in the BSTR unicode format. The 'GetString' function loads a string from resource by number. The 'MessagesID' contains some string identifiers needed in program (error messages, libraries names, etc.). When all the constants are loaded it calls the **ReadProject** function that loads the binary project:

Code:

```
' // Load project
Function ReadProject() As Boolean
    Dim hResource       As Long:                Dim hMememory       As Long
    Dim lResSize        As Long:                Dim pRawData        As Long
    Dim status          As Long:                Dim pUncompressed   As Long
    Dim lUncompressSize As Long:                Dim lResultSize     As Long
    Dim tmpStorageItem  As BinStorageListItem:  Dim tmpExecuteItem  As BinExecListItem
    Dim pLocalBuffer    As Long

    ' // Load resource
    hResource = FindResource(hInstance, GetString(PROJECT), RT_RCDATA)
    If hResource = 0 Then GoTo CleanUp

    hMememory = LoadResource(hInstance, hResource)
    If hMememory = 0 Then GoTo CleanUp

    lResSize = SizeofResource(hInstance, hResource)
    If lResSize = 0 Then GoTo CleanUp

    pRawData = LockResource(hMememory)
    If pRawData = 0 Then GoTo CleanUp

    pLocalBuffer = HeapAlloc(GetProcessHeap(), HEAP_NO_SERIALIZE, lResSize)
    If pLocalBuffer = 0 Then GoTo CleanUp

    ' // Copy to local buffer
    CopyMemory ByVal pLocalBuffer, ByVal pRawData, lResSize

    ' // Set default size
    lUncompressSize = lResSize * 2

    ' // Do decompress
```

As you can see i use the heap memory instead arrays. Firstly, it loads the 'PROJECT' resource and copies one to heap memory then tries to decompres using the **RtlDecompressBuffer** function. This function is not returns the sufficient output buffer size therefore we try to do decompress of the buffer increasing the output buffer size. Afterwards it checks all parameters and initializes the global project pointers.

(rate this post) Reply Reply With Quote

May 22nd, 2016, 07:03 PM #5

The trick

Thread Starter
Fanatic Member

Join Date: Feb 2015
Location: Russia, Astrakhan
Posts: 842

Re: [VB6] Loader, shellcode, without runtime...

If it is succeeded then it launches the 'CopyProcess' procedure that unpacks all the storage items according project data:

Code:

```
' // Copying process
Function CopyProcess() As Boolean
    Dim bItem       As BinStorageListItem:  Dim index       As Long
    Dim pPath       As Long:                Dim dwWritten   As Long
    Dim msg         As Long:                Dim lStep       As Long
    Dim isError     As Boolean:             Dim pItem       As Long
    Dim pErrMsg     As Long:                Dim pTempString As Long

    ' // Set pointer
    pItem = pStoragesTable

    ' // Go thru file list
    For index = 0 To ProjectDesc.storageDescriptor.dwNumberOfItems - 1

        ' // Copy file descriptor
        CopyMemory bItem, ByVal pItem, Len(bItem)

        ' // Next item
        pItem = pItem + ProjectDesc.storageDescriptor.dwSizeOfItem

        ' // If it is not main executable
        If index <> ProjectDesc.storageDescriptor.iExecutableIndex Then

            ' // Normalize path
            pPath = NormalizePath(pStringsTable + bItem.ofstDestPath, pStringsTable + bIte

            ' // Error occurs
            If pPath = 0 Then

                pErrMsg = GetString(MID_ERRORWIN32)
                MessageBox 0, pErrMsg, 0, MB_ICONERROR Or MB_SYSTEMMODAL
                GoTo CleanUp
```

This procedure goes through all the storage items and unpacks all the items one by one except the main executable file. The 'NormalizePath' function replace the wildcards in the path to the real strings path. There is the 'CreateSubdirectories' function that creates the intermediate directories (if needs) for specified path. Then it calls the **CreateFile** function and copy data to it through **WriteFile**. If an error occurs it shows the message box with the standard suggestions: Retry, Abort, Ignore.

Code:

```
' // Create all subdirectories by path
Function CreateSubdirectories( _
```

```
            ByVal pPath As Long) As Boolean
    Dim pComponent As Long
    Dim tChar      As Integer

    ' // Pointer to first char
    pComponent = pPath

    ' // Go thru path components
    Do

        ' // Get next component
        pComponent = PathFindNextComponent(pComponent)

        ' // Check if end of line
        CopyMemory tChar, ByVal pComponent, 2
        If tChar = 0 Then Exit Do

        ' // Write null-terminator
        CopyMemory ByVal pComponent - 2, 0, 2

        ' // Check if path exists
        If PathIsDirectory(pPath) = 0 Then

            ' // Create folder
            If CreateDirectory(pPath, ByVal 0&) = 0 Then
                ' // Error
                CopyMemory ByVal pComponent - 2, &H5C, 2
                Exit Function
            End If
```

After unpacking it is call the 'ExecuteProcess' function that launches all the commands using **ShellExecuteEx** function:

Code:

```
' // Execution command process
Function ExecuteProcess() As Boolean
    Dim index       As Long:                Dim bItem       As BinExecListItem
    Dim pPath       As Long:                Dim pErrMsg     As Long
    Dim shInfo      As SHELLEXECUTEINFO:    Dim pTempString As Long
    Dim pItem       As Long:                Dim status      As Long

    ' // Set pointer and size
    shInfo.cbSize = Len(shInfo)
    pItem = pExecutesTable

    ' // Go thru all items
    For index = 0 To ProjectDesc.execListDescriptor.dwNumberOfItems - 1

        ' // Copy item
        CopyMemory bItem, ByVal pItem, ProjectDesc.execListDescriptor.dwSizeOfItem

        ' // Set pointer to next item
        pItem = pItem + ProjectDesc.execListDescriptor.dwSizeOfItem

        ' // Normalize path
        pPath = NormalizePath(pStringsTable + bItem.ofstFileName, 0)

        ' // Fill SHELLEXECUTEINFO
        shInfo.lpFile = pPath
        shInfo.lpParameters = pStringsTable + bItem.ofstParameters
        shInfo.fMask = SEE_MASK_NOCLOSEPROCESS Or SEE_MASK_FLAG_NO_UI
        shInfo.nShow = SW_SHOWDEFAULT

        ' // Performing...
        status = ShellExecuteEx(shInfo)
```

As you can see it's like the previous procedure. Here is the same it only uses **ShellExecuteEx** instead unpacking. Note that each operation performs synchronously, i.e. each calling of **ShellExecuteEx** waits for operation will have been done.

(rate this post) Reply Reply With Quote

May 22nd, 2016, 07:04 PM #6

The trick

Thread Starter
Fanatic Member

Join Date: Feb 2015
Location: Russia, Astrakhan
Posts: 842

Re: [VB6] Loader, shellcode, without runtime...

If it is succeeded then it is call the 'RunProcess' function that prepares the data for executing main executable from memory.

Code:

```
' // Run exe from project in memory
Function RunProcess() As Boolean
    Dim bItem       As BinStorageListItem:  Dim Length      As Long
    Dim pFileData   As Long

    ' // Get descriptor of executable file
    CopyMemory bItem, ByVal pStoragesTable + ProjectDesc.storageDescriptor.dwSizeOfItem *
                      ProjectDesc.storageDescriptor.iExecutableIndex, Len(bItem)


    ' // Alloc memory within top memory addresses
    pFileData = VirtualAlloc(ByVal 0&, bItem.dwSizeOfFile, MEM_TOP_DOWN Or MEM_COMMIT, PAG
    If pFileData = 0 Then Exit Function

    ' // Copy raw exe file to this memory
    CopyMemory ByVal pFileData, ByVal pFilesTable + bItem.ofstBeginOfData, bItem.dwSizeOfI

    ' // Free decompressed project data
    HeapFree GetProcessHeap(), HEAP_NO_SERIALIZE, pProjectData
    pProjectData = 0

    ' // Run exe from memory
    RunExeFromMemory pFileData, bItem.dwFlags And FF_IGNOREERROR

    ' ------------------------------------------------
    ' // An error occurs
    ' // Clean memory

    VirtualFree ByVal pFileData, 0, MEM_RELEASE

    ' // If ignore error then success
    If bItem.dwFlags And FF_IGNOREERROR Then RunProcess = True
```

It allocates the memory in the top area of virtual addresses because the most of exe files is loaded to quite low addresses (usually 0x00400000). Afterwards it free the memory of the project data because if the process is launched this memory won't release, then it call the 'RunExeFromMemory' function that does next step of loading an exe. If for any reasons loading of an exe file wouldn't done it frees the allocated memory and return the control to the 'Main' function. So, in order to load an exe file we should release the loader memory, i.e. unload us loader. We should leave small piece of code that will load an exe and run it. I decide to use the shellcode, although it is possible to use a dll. The shellcode is the small base-independent code (this code doesn't refer to external data) that allows to do the usefull stuff. Anyway we should ensure the access to API functions from the shellcode. You can't call an api function directly from the shellcode because the main exe is unloaded and any reference to the import table of main exe occurs crash. The second restriction is the 'CALL' instruction can use relative offset (it

is most frequently case). Therefore we should initialize some "springboard" that will jump an api function. I decide to do it using splicing method. I just replace the first 5 bytes of a stub function to JMP assembler instruction that refers to the needed API:

Code:

```
' // Run EXE file by memory address
Function RunExeFromMemory(_
                ByVal pExeData As Long,_
                ByVal IgnoreError As Boolean) As Boolean
    Dim Length  As Long:    Dim pCode       As Long
    Dim pszMsg  As Long:    Dim pMsgTable   As Long
    Dim index   As Long:    Dim pCurMsg     As Long

    ' // Get size of shellcode
    Length = GetAddr(AddressOf ENDSHELLLOADER) - GetAddr(AddressOf BEGINSHELLLOADER)

    ' // Alloc memory within top addresses
    pCode = VirtualAlloc(ByVal 0&, Length, MEM_TOP_DOWN Or MEM_COMMIT, PAGE_EXECUTE_READWR

    ' // Copy shellcode to allocated memory
    CopyMemory ByVal pCode, ByVal GetAddr(AddressOf BEGINSHELLLOADER), Length

    ' // Initialization of shellcode
    If Not InitShellLoader(pCode) Then GoTo CleanUp

    ' // Splice CallLoader function in order to call shellcode
    Splice AddressOf CallLoader, pCode + GetAddr(AddressOf LoadExeFromMemory) - GetAddr(Ad

    ' // Check ignore errors
    If Not IgnoreError Then

        ' // Alloc memory for messages table
        pMsgTable = VirtualAlloc(ByVal 0&, 1024, MEM_TOP_DOWN Or MEM_COMMIT, PAGE_READWRIT
        If pMsgTable = 0 Then GoTo CleanUp

        ' // Skip pointers
        pCurMsg = pMsgTable + FM_END * 4
```

As you can see it calculates the size of shellcode using the difference between the extreme functions **ENDSHELLLOADER** and **BEGINSHELLLOADER**. These functions should surround the shellcode and have the different prototypes because VB6 compiler can union identical functions. Then it allocates the memory for the shellcode and copies the shellcode to there. Afterwards it calls the 'InitShellLoader' function, that splaces all the function in shellcode:

Code:

```
' // Shellcode initialization
Function InitShellLoader(_
                ByVal pShellCode As Long) As Boolean
    Dim hLib    As Long:        Dim sName   As Long
    Dim sFunc   As Long:        Dim lpAddr  As Long
    Dim libIdx  As Long:        Dim fncIdx  As Long
    Dim libName As MessagesID:  Dim fncName As MessagesID
    Dim fncSpc  As Long:        Dim splAddr As Long

    ' // +-------------------------------------------------------------------+
    ' // |                      Fixing of API addresses                      |
    ' // +-------------------------------------------------------------------+
    ' // | In order to call api function from shellcode i use splicing of    |
    ' // |    our VB functions and redirect call to corresponding api.       |
    ' // |     I did same in the code that injects to other process.         |
    ' // +-------------------------------------------------------------------+

    splAddr = GetAddr(AddressOf tVirtualAlloc) - GetAddr(AddressOf BEGINSHELLLOADER) + pSh

    ' // Get size in bytes between stub functions
    fncSpc = GetAddr(AddressOf tVirtualProtect) - GetAddr(AddressOf tVirtualAlloc)

    ' // Use 3 library: kernel32, ntdll и user32
    For libIdx = 0 To 2

        ' // Get number of imported functions depending on library
        Select Case libIdx
        Case 0: libName = API_LIB_KERNEL32: fncIdx = 13
        Case 1: libName = API_LIB_NTDLL:    fncIdx = 1
        Case 2: libName = API_LIB_USER32:   fncIdx = 1
        End Select
```

Firstly it calculates the offset of the first "springboard" function (in this case it is **tVirtualAlloc** function) from beginning of the shellcode and calculates the distance in bytes between "springboard" functions. When VB6-compiler compiles an module it puts all the functions in the same order as in the code. The needed condition is to ensure the unique returned value from each function. Then it goes through all the needed libraries (kernel32, ntdll, user32 - in this order) and their functions. The first item in the resource strings is the library name followed by the functions names in this library. When an item is obtained it translates the function name to ANSI format and calls **GetProcAddress** function. Afterwards it calls the **Splice** function that makes up the "springboard" to the needed function from the shellcode. Eventually, it modified the **CallByPointer** function in order to ensure the jump from the shellcode to the loaded exe. Okay, further the **RunExeFromMemory** function splices the **CallLoader** in order to ensure the jump to shellcode from the main executable. After this operation is done the function begins to form the error message table (if needs) that is just the set of pointers to the messages strings. Eventually it call the spliced **CallLoader** function that jumps to the **LoadExeFromMemory** shellcode function that has already not been placed in main exe.

Last edited by The trick; May 23rd, 2016 at 10:41 AM.

(rate this post) Reply | Reply With Quote

May 22nd, 2016, 07:07 PM #7

The trick

Thread Starter
Fanatic Member

Join Date: Feb 2015
Location: Russia, Astrakhan
Posts: 842

Re: [VB6] Loader, shellcode, without runtime...

Inside shellcode.

So, i made the several function inside the shellcode:

1. LoadExeFromMemory - is the main function of the shellcode;
2. GetImageNtHeaders - returns the **IMAGE_NT_HEADERS** structure and its address by the passed base address;
3. GetDataDirectory - returns the **IMAGE_DATA_DIRECTORY** structure and its address by the passed base address and catalog index;
4. EndProcess - shows the error message (if any) and ends of the process;
5. ProcessSectionsAndHeaders - allocates the memory for all headers (DOS, NT, sections) and all the sections. Copies all data to the sections;
6. ReserveMemory - reserves the sufficient memory for EXE;
7. ProcessRelocations - adjusts the addresses if an exe has not been loaded to base address;
8. ProcessImportTable - scans the import table of an exe file, loads the needed libraries and

fills the import address table;
9. SetMemoryPermissions - adjusts the memory permissions for each section;
10. UpdateNewBaseAddress - refresh the new base address in the system structures PEB and LDR.

Due to the fact i can't use the **VarPtr** function, i made the similar function using the **lstrcpyn** function - **IntPtr**. So, the **'LoadExeFromMemory'** function obtain firstly the NT headers and checks the processor architecture, whether the PE file is executable and whether the PE file is 32 bit application. If it is succeeded then the shellcode unload the main exe file from memory using the **ZwUnmapViewOfSection** function. If function has been succeeded the main exe file isn't in the memory anymore and the memory occupied by exe has been released. Henceforth we can't directly use API function, we should use our "springboards":

Code:

```
' // Parse exe in memory
Function LoadExeFromMemory(_
                ByVal pRawData As Long, _
                ByVal pMyBaseAddress As Long, _
                ByVal pErrMsgTable As Long) As Boolean
    Dim NtHdr   As IMAGE_NT_HEADERS
    Dim pBase   As Long
    Dim index   As Long
    Dim iError  As ERROR_MESSAGES
    Dim pszMsg  As Long

    ' // Get IMAGE_NT_HEADERS
    If GetImageNtHeaders(pRawData, NtHdr) = 0 Then
        iError = EM_UNABLE_TO_GET_NT_HEADERS
        EndProcess pErrMsgTable, iError
        Exit Function
    End If

    ' // Check flags
    If NtHdr.FileHeader.Machine <> IMAGE_FILE_MACHINE_I386 Or
       (NtHdr.FileHeader.Characteristics And IMAGE_FILE_EXECUTABLE_IMAGE) = 0 Or
       (NtHdr.FileHeader.Characteristics And IMAGE_FILE_32BIT_MACHINE) = 0 Then Exit Funct

    ' // Release main EXE memory. After that main exe is unloaded from memory.
    ZwUnmapViewOfSection GetCurrentProcess(), GetModuleHandle(ByVal 0&)

    ' // Reserve memory for EXE
    iError = ReserveMemory(pRawData, pBase)
    If iError Then
        EndProcess pErrMsgTable, iError
        Exit Function
    End If
```

Then shellcode calls the **ReserveMemory** function shown below. This function extracts the NT header from the loadable exe and tries to reserve the memory at 'ImageBase' address with the 'SizeOfImage' size. If it isn't succeeded the function checks if the exe file contains the relocation information. If so, it tries to reserve memory at any address. The relocation information allows to load an PE file to any address other than 'ImageBase'. It contains all the places where an exe uses the absolute addressing. You can adjust these places using the difference between the real base address and the 'ImageBase' field:

Code:

```
' // Reserve memory for EXE
Function ReserveMemory(
                ByVal pRawExeData As Long,
                ByRef pBase As Long) As ERROR_MESSAGES
    Dim NtHdr       As IMAGE_NT_HEADERS
    Dim pLocBase    As Long

    If GetImageNtHeaders(pRawExeData, NtHdr) = 0 Then
        ReserveMemory = EM_UNABLE_TO_GET_NT_HEADERS
        Exit Function
    End If

    ' // Reserve memory for EXE
    pLocBase = tVirtualAlloc(ByVal NtHdr.OptionalHeader.ImageBase, _
                        NtHdr.OptionalHeader.SizeOfImage,
                        MEM_RESERVE, PAGE_EXECUTE_READWRITE)
    If pLocBase = 0 Then

        ' // If relocation information not found error
        If NtHdr.FileHeader.Characteristics And IMAGE_FILE_RELOCS_STRIPPED Then

            ReserveMemory = EM_UNABLE_TO_ALLOCATE_MEMORY
            Exit Function

        Else
            ' // Reserve memory in other region
            pLocBase = tVirtualAlloc(ByVal 0&, NtHdr.OptionalHeader.SizeOfImage, _
                                MEM_RESERVE, PAGE_EXECUTE_READWRITE)

            If pLocBase = 0 Then

                ReserveMemory = EM_UNABLE_TO_ALLOCATE_MEMORY
```

Okay, if memory reserving failed it shows the message with error and ends the application. Otherwise it calls the **ProcessSectionsAndHeaders** function. This function places all the headers to the allocated memory, extracts the information about all the sections and copies all the data to sections. If an section has the uninitialized data it fills this region with zero:

Code:

```
' // Allocate memory for sections and copy them data to there
Function ProcessSectionsAndHeaders(_
                ByVal pRawExeData As Long,
                ByVal pBase As Long) As ERROR_MESSAGES

    Dim iSec    As Long
    Dim pNtHdr  As Long
    Dim NtHdr   As IMAGE_NT_HEADERS
    Dim sec     As IMAGE_SECTION_HEADER
    Dim lpSec   As Long
    Dim pData   As Long

    pNtHdr = GetImageNtHeaders(pRawExeData, NtHdr)
    If pNtHdr = 0 Then
        ProcessSectionsAndHeaders = EM_UNABLE_TO_GET_NT_HEADERS
        Exit Function
    End If

    ' // Alloc memory for headers
    pData = tVirtualAlloc(ByVal pBase, NtHdr.OptionalHeader.SizeOfHeaders, MEM_COMMIT, PA
    If pData = 0 Then
        ProcessSectionsAndHeaders = EM_UNABLE_TO_ALLOCATE_MEMORY
        Exit Function
    End If

    ' // Copy headers
    tCopyMemory pData, pRawExeData, NtHdr.OptionalHeader.SizeOfHeaders

    ' // Get address of beginnig of sections headers
    pData = pNtHdr + Len(NtHdr.Signature) + Len(NtHdr.FileHeader) + NtHdr.FileHeader.Size
```

Then the **LoadExeFromMemory** function calls the **UpdateNewBaseAddress** function that update the new base address in the *user-mode* system structures. Windows creates the special stucture named PEB (Process Environment Block) for each process. This is the very usefull structure that allows to obtain the very many information about the process. Many API functions gets information from this structure. For example **GetModuleHandle(NULL)** takes the returned value from the **PEB.ImageBaseAddress** or **GetModuleHandle("MyExename")** takes the returned value from the **PEB.Ldr** list of the loaded modules. We should update this information according the new base address in order to API functions retrieve the correct values. The small part of **PEB** structure is shown below:

Code:

```
Type PEB
    NotUsed                     As Long
    Mutant                      As Long
```

```
    ImageBaseAddress                As Long
    LoaderData                      As Long ' // Pointer to PEB_LDR_DATA
    ProcessParameters               As Long
    ' // ....
End Type
```

We are interested only the 'ImageBaseAddress' and 'LoaderData' fields. The first field contains the base address of an exe file. The second field contains the pointer to the **PEB_LDR_DATA** structure that describes all the loaded modules in the process:

Code:

```
Type PEB_LDR_DATA
    Length                          As Long
    Initialized                     As Long
    SsHandle                        As Long
    InLoadOrderModuleList           As LIST_ENTRY
    InMemoryOrderModuleList         As LIST_ENTRY
    InInitializationOrderModuleList As LIST_ENTRY
End Type
```

This structure contains the three doubly-linked lists that describe each module. The 'InLoadOrderModuleList' list contains the links to items in the loading oreder item, i.e. the items in this list is placed in loading order (the first module is at beginning). The 'InMemoryOrderModuleList' is same only in order of placing in memory, 'InInitializationOrderModuleList' in initialization order. We should get the first element of 'InLoadOrderModuleList' list that is the pointer to structure **LDR_MODULE**:

Code:

```
Type LDR_MODULE
    InLoadOrderModuleList           As LIST_ENTRY
    InMemoryOrderModuleList         As LIST_ENTRY
    InInitOrderModuleList           As LIST_ENTRY
    BaseAddress                     As Long
    EntryPoint                      As Long
    SizeOfImage                     As Long
    FullDllName                     As UNICODE_STRING
    BaseDllName                     As UNICODE_STRING
    Flags                           As Long
    LoadCount                       As Integer
    TlsIndex                        As Integer
    HashTableEntry                  As LIST_ENTRY
    TimeDateStamp                   As Long
End Type
```

This structure describes an module. The first element of 'InLoadOrderModuleList' is the main exe module descriptor. We should change the 'BaseAddress' field to new value and save changes. So, in order to obtain the **PEB** structure we can use the universal function **NtQueryInformationProcess** that extract the many useful information about process (read more in 'Windows NT/2000 Native API Reference' by Gary Nebbett). The **PEB** structure can be obtained from the **PROCESS_BASIC_INFORMATION** structure that describes the basic information about the process:

Code:

```
Type PROCESS_BASIC_INFORMATION
    ExitStatus                      As Long
    PebBaseAddress                  As Long
    AffinityMask                    As Long
    BasePriority                    As Long
    UniqueProcessId                 As Long
    InheritedFromUniqueProcessId    As Long
End Type
```

The 'PebBaseAddress' field contains the address of the **PEB** structure.

Last edited by The trick; May 22nd, 2016 at 07:15 PM.

(rate this post) Reply | Reply With Quote

May 22nd, 2016, 07:09 PM #8

The trick

Thread Starter
Fanatic Member

Join Date: Feb 2015
Location: Russia, Astrakhan
Posts: 842

Re: [VB6] Loader, shellcode, without runtime...

In order to obtain the **PROCESS_BASIC_INFORMATION** structure we should pass the **ProcessBasicInformation** as the class information to **NtQueryInformationProcess** function. Because of structure size may change in various versions of Windows i use the heap memory for extracting the **PROCESS_BASIC_INFORMATION** structure. If the size doesn't suit it increases the size and repeats again:

Code:

```
Function UpdateNewBaseAddress(_
                ByVal pBase As Long) As ERROR_MESSAGES
    Dim pPBI    As Long:                        Dim PBIlen  As Long
    Dim PBI     As PROCESS_BASIC_INFORMATION:   Dim cPEB    As PEB
    Dim ntstat  As Long
    Dim ldrData As PEB_LDR_DATA
    Dim ldrMod  As LDR_MODULE

    ntstat = tNtQueryInformationProcess(tGetCurrentProcess(), ProcessBasicInformation, Int

    Do While ntstat = STATUS_INFO_LENGTH_MISMATCH

        PBIlen = PBIlen * 2

        If pPBI Then
            tHeapFree tGetProcessHeap(), HEAP_NO_SERIALIZE, pPBI
        End If

        pPBI = tHeapAlloc(tGetProcessHeap(), HEAP_NO_SERIALIZE, PBIlen)
        ntstat = tNtQueryInformationProcess(tGetCurrentProcess(), ProcessBasicInformation,

    Loop

    If ntstat <> STATUS_SUCCESS Then
        UpdateNewBaseAddress = EM_PROCESS_INFORMATION_NOT_FOUND
        GoTo CleanUp
    End If

    If pPBI Then
        ' // Copy to PROCESS_BASIC_INFORMATION
        tCopyMemory IntPtr(PBI.ExitStatus), pPBI, Len(PBI)
    End If
```

After updating of the base address in the system structures the shellcode calls the

ProcessImportTable function that loads the needed libraryes for exe file. Firstly it gets the **IMAGE_DIRECTORY_ENTRY_IMPORT** directory that contains the RVA of the array of the **IMAGE_IMPORT_DESCRIPTOR** structures:

Code:

```
Type IMAGE_IMPORT_DESCRIPTOR
    Characteristics                 As Long
    TimeDateStamp                   As Long
    ForwarderChain                  As Long
    pName                           As Long
    FirstThunk                      As Long
End Type
```

Each structure describes the single DLL. The 'pName' field contains the RVA to the ASCIIZ library name. The 'Characteristics' field contains the RVA to the table of the imported function names and 'FirstThunk' contains the RVA of the import addresses table. The names table is the array of **IMAGE_THUNK_DATA** structures that is the 32 bit Long value. If the most significant bit is set the remaining bits represents the ordinal of the function (import by ordinal). Otherwise the remaining bits contains the RVA of the function name prenexed by 'Hint' value. If the **IMAGE_THUNK_DATA** structure contains zero it means that no more names. If all the fields of the **IMAGE_IMPORT_DESCRIPTOR** equal zero it means that list of structureas is ended.

Code:

```
' // Process import table
Function ProcessImportTable( _
                ByVal pBase As Long) As ERROR_MESSAGES
    Dim NtHdr           As IMAGE_NT_HEADERS:       Dim datDirectory    As IMAGE_DATA_DIRE
    Dim dsc             As IMAGE_IMPORT_DESCRIPTOR: Dim hLib            As Long
    Dim thnk            As Long:                    Dim Addr            As Long
    Dim fnc             As Long:                    Dim pData           As Long

    If GetImageNtHeaders(pBase, NtHdr) = 0 Then
        ProcessImportTable = EM_UNABLE_TO_GET_NT_HEADERS
        Exit Function
    End If

    ' // Import table processing
    If NtHdr.OptionalHeader.NumberOfRvaAndSizes > 1 Then

        If GetDataDirectory(pBase, IMAGE_DIRECTORY_ENTRY_IMPORT, datDirectory) = 0 Then
            ProcessImportTable = EM_INVALID_DATA_DIRECTORY
            Exit Function
        End If

        ' // If import table exists
        If datDirectory.Size > 0 And datDirectory.VirtualAddress > 0 Then

            ' // Copy import descriptor
            pData = datDirectory.VirtualAddress + pBase
            tCopyMemory IntPtr(dsc.Characteristics), pData, Len(dsc)

            ' // Go thru all descriptors
            Do Until dsc.Characteristics = 0 And _
                     dsc.FirstThunk = 0 And _
                     dsc.ForwarderChain = 0 And
```

The **ProcessRelocation** function is called then. This functions adjust all the absolute references (if any). It obtains the **IMAGE_DIRECTORY_ENTRY_BASERELOC** catalog that contains the RVA to the array of **IMAGE_BASE_RELOCATION** structures. Each item in this list contains the settings within 4KB relative 'VirtualAddress' fields:

Code:

```
Type IMAGE_BASE_RELOCATION
    VirtualAddress                  As Long
    SizeOfBlock                     As Long
End Type
```

The 'SizeOfBlock' contains the size of item in bytes. The array of 16 bits numbers is placed after the each **IMAGE_BASE_RELOCATION** structure. You can calculate number of this strucuture as (SizeOfBlock - Len(IMAGE_BASE_RELOCATION)) \ Len(Integer). Each element of the array of the descriptors has the following structure:

The high four bits contains the type of relocation. We are interested the **IMAGE_REL_BASED_HIGHLOW** type that means we should add the difference (RealBaseAddress - ImageBaseAddress) to a Long that is at the address 'VirtualAddress' + 12 least bits of descriptors. Array of **IMAGE_BASE_RELOCATION** structures is ended with stucture where all fields is zero:

Code:

```
' // Process relocations
Function ProcessRelocations( _
                ByVal pBase As Long) As ERROR_MESSAGES
    Dim NtHdr           As IMAGE_NT_HEADERS:       Dim datDirectory    As IMAGE_DATA_DIRE
    Dim relBase         As IMAGE_BASE_RELOCATION:  Dim entriesCount    As Long
    Dim relType         As Long:                   Dim dwAddress       As Long
    Dim dwOrig          As Long:                   Dim pRelBase        As Long
    Dim delta           As Long:                   Dim pData           As Long

    ' // Check if module has not been loaded to image base value
    If GetImageNtHeaders(pBase, NtHdr) = 0 Then
        ProcessRelocations = EM_UNABLE_TO_GET_NT_HEADERS
        Exit Function
    End If

    delta = pBase - NtHdr.OptionalHeader.ImageBase

    ' // Process relocations
    If delta Then

        ' // Get address of relocation table
        If GetDataDirectory(pBase, IMAGE_DIRECTORY_ENTRY_BASERELOC, datDirectory) = 0 Then
            ProcessRelocations = EM_INVALID_DATA_DIRECTORY
            Exit Function
        End If

        If datDirectory.Size > 0 And datDirectory.VirtualAddress > 0 Then

            ' // Copy relocation base
            pRelBase = datDirectory.VirtualAddress + pBase
            tCopyMemory IntPtr(relBase.VirtualAddress), pRelBase, Len(relBase)
```

Last edited by The trick; May 22nd, 2016 at 07:15 PM.

EXE loader|Inline assembler Add-in|Kernel-mode driver on VB6|Multithreading in StandartEXE|Native VB6-DLL
Code injection to other process|DLL injection|DirectX9|DirectSound|TrickControls|MP3 player from memory
Easy calling by pointer|Hash-table|Safe subclassing|Vocoder|Creation of native DLL|COM-DLL without registration
FM-synthesizer|FFT spectrum-analyser|3D Fir-tree|Asynch waiter|Wave steganography|Fast AVI-trimmer
Windows with custom rendering|String to integer and vice versa|Multithreading with marshaling|Owner-draw listbox mod
Advanced math|Library info|Create GIF-animation|Store data to self-EXE|Computer creates a music|TrickSound class

Support me:
YandexMoney: 410011032160614
BTC: 1B9xD98ThjstyQUP2BUz9sBYhWT9rzqWfe

(rate this post) Reply Reply With Quote

May 22nd, 2016, 07:13 PM #9

The trick

Thread Starter
Fanatic Member

Join Date: Feb 2015
Location: Russia, Astrakhan
Posts: 842

Re: [VB6] Loader, shellcode, without runtime...

After relocations processing shellcode calls the function **SetMemoryPermissions** that adjusts the memory protection for each section according to the 'Characteristics' field of **IMAGE_SECTION_HEADER** structure. It just calls the **VirtualProtect** function with the certain memory attributes:

Code:

```
' // Set memory permissions
Private Function SetMemoryPermissions(_
                ByVal pBase As Long) As ERROR_MESSAGES
    Dim iSec    As Long:                Dim pNtHdr  As Long
    Dim NtHdr   As IMAGE_NT_HEADERS:    Dim sec     As IMAGE_SECTION_HEADER
    Dim Attr    As MEMPROTECT:          Dim pSec    As Long
    Dim ret     As Long

    pNtHdr = GetImageNtHeaders(pBase, NtHdr)
    If pNtHdr = 0 Then
        SetMemoryPermissions = EM_UNABLE_TO_GET_NT_HEADERS
        Exit Function
    End If

    ' // Get address of first section header
    pSec = pNtHdr + 4 + Len(NtHdr.FileHeader) + NtHdr.FileHeader.SizeOfOptionalHeader

    ' // Go thru section headers
    For iSec = 0 To NtHdr.FileHeader.NumberOfSections - 1

        ' // Copy section descriptor
        tCopyMemory IntPtr(sec.SectionName(0)), pSec, Len(sec)

        ' // Get type
        If sec.Characteristics And IMAGE_SCN_MEM_EXECUTE Then
            If sec.Characteristics And IMAGE_SCN_MEM_READ Then
                If sec.Characteristics And IMAGE_SCN_MEM_WRITE Then
                    Attr = PAGE_EXECUTE_READWRITE
                Else
                    Attr = PAGE_EXECUTE_READ
                End If
            Else
```

Eventually it frees the message table (if any) and calls the entry point of the loaded exe. In the previous version of the loader i unloaded the shellcode too but some exe doesn't call **ExitProcess** therefore it can causes the crash. The loader has been done.
Although we write the loader without runtime usage the VB6 compiler adds it because all the OBJ files has references to MSVBVM60 during compilation. We have to remove the runtime from the import table of the loader manually. I made the special utility - *Patcher* that searches runtime in the import table and the bound import table and removes it. This utility is helpful for the VB kernel drivers too. I won't describe the its work because it uses same concepts of the PE format that we already examined. Overall we get the working exe that doesn't use MSVBVM60 runtime on the target machine.

In order to use the loader you should compile it then you should run the patcher and patch compiled loader. Afterwards you can use the compiler.

I hope you enjoyed it. Thank for attention!
Regards,
Кривоус Анатолий (The trick).

Attached Files
VBLoader.zip (176.4 KB, 109 views)

Last edited by The trick; Jul 30th, 2016 at 08:54 AM. ***Reason:*** *Fixed small bug in Patcher utility*

EXE loader|Inline assembler Add-in|Kernel-mode driver on VB6|Multithreading in StandartEXE|Native VB6-DLL
Code injection to other process|DLL injection|DirectX9|DirectSound|TrickControls|MP3 player from memory
Easy calling by pointer|Hash-table|Safe subclassing|Vocoder|Creation of native DLL|COM-DLL without registration
FM-synthesizer|FFT spectrum-analyser|3D Fir-tree|Asynch waiter|Wave steganography|Fast AVI-trimmer
Windows with custom rendering|String to integer and vice versa|Multithreading with marshaling|Owner-draw listbox mod
Advanced math|Library info|Create GIF-animation|Store data to self-EXE|Computer creates a music|TrickSound class

Support me:
YandexMoney: 410011032160614
BTC: 1B9xD98ThjstyQUP2BUz9sBYhWT9rzqWfe

(rate this post) Reply Reply With Quote

Oct 6th, 2016, 12:42 PM #10

Elroy

PowerPoster

Join Date: Jun 2014
Location: Near Nashville TN
Posts: 2,698

Re: [VB6] Loader, shellcode, without runtime...

Absolutely incredible. I'm still a ways from digesting all of this, but it's just an amazing exposé of the guts of an executable.

Many Thanks,
Elroy

https://www.youtube.com/watch?v=wXlvy3sTTBk

(rate this post) Reply Reply With Quote

Oct 6th, 2016, 08:35 PM #11

fafalone

Frenzied Member

Join Date: Jul 2010
Location: NYC
Posts: 1,694

Re: [VB6] Loader, shellcode, without runtime...

Wow, amazing work. You've managed to raise the bar for VB6 capabilities once again.

[TaskDialogIndirect Full-featured class, now with Input Boxes. ComboBoxes, And More!] ~ [Undocumented ListView: Footer items | Virtual Mode Grouping | Subsetted groups]
[Intro to easier subclassing with SetWindowSubclass]
[oleexp.tlb: olelib expansion w/Modern Shell Interfaces (Vista+ features in VB6)]
oleexp projects: [Advanced Thumbnails LV for All Files] ~ [Customized Open/Save dialogs with IFileDialog] ~ [Replace Directory TV/LV/LB with real Explorer instance] ~ [Custom Autocomplete Lists] ~ [ITaskbarList3: Overlays, progress, and more] ~ [IFileOperation for modern Copy/Move] ~ [List/execute associated handlers with IAssocHandler] ~ [SHDoDragDrop/SHCreateDataObject for OLE/IDropSource-less File Dragging] ~ [Show Explorer drag image on any control] ~ [Show previews for files beyond just images: IPreviewHandler] ~ [IStorage for Unzip w/o shell object/3rd party DLL, and create/add to zips with IDropTarget] ~ [Easy image display/rotate/scale: IShellImageData; IImageList primer] ~ [Core Audio For VB6] ~ [List *all* file properties] ~ see details & more oleexp projects ~ All of my CodeBank submissions

(rate this post) Reply Reply With Quote

Oct 7th, 2016, 11:46 PM #12

xxdoc123

Addicted Member

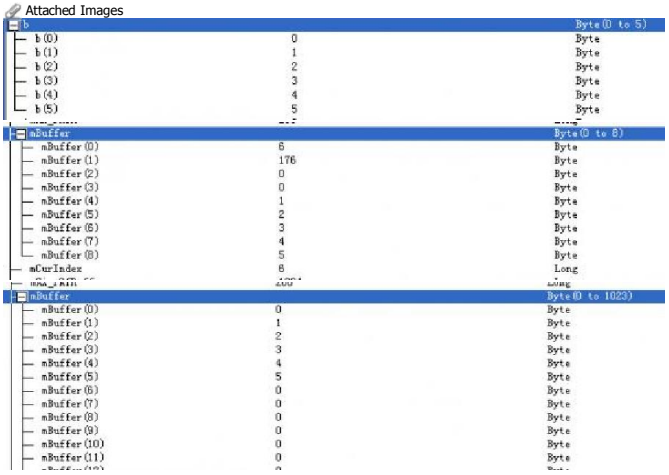
Join Date: Aug 2016
Posts: 144

Re: [VB6] Loader, shellcode, without runtime...

Code:

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
  Persistable = 0  'NotPersistable
  DataBindingBehavior = 0  'vbNone
  DataSourceBehavior  = 0  'vbNone
  MTSTransactionMode  = 0  'NotAnMTSObject
END
Attribute VB_Name = "clsStream"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
' // clsStream.cls - binary stream class
' // ?Krivous Anatoly Anatolevich (The trick), 2016

Option Explicit

Private Declare Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long

Private Declare Function ReadFile
                Lib "kernel32" (ByVal hFile As Long, _
                                ByRef lpBuffer As Any, _
                                ByVal nNumberOfBytesToRead As Long, _
                                ByRef lpNumberOfBytesRead As Long, _
                                ByRef lpOverlapped As Any) As Long

Private Declare Function GetFileSizeEx
                Lib "kernel32" (ByVal hFile As Long,
                                ByRef lpFileSize As LARGE_INTEGER) As Long

Private Declare Function CreateFile
```

Attached Images



Last edited by xxdoc123; Mar 8th, 2017 at 01:09 AM.

(rate this post) Reply Reply With Quote

Oct 8th, 2016, 04:43 AM #13

The trick

Thread Starter
Fanatic Member

Join Date: Feb 2015
Location: Russia, Astrakhan
Posts: 842

Re: [VB6] Loader, shellcode, without runtime...

> *CompressStream why the array sizi became 9 not 6 .*

This is the function features and compression algorithm. For example it has 2 bytes header.

> *how can i used RtlDecompressBuffer to deCompress*

You should look at the launcher solution, there is the code for decompression data.

EXE loader|Inline assembler Add-in|Kernel-mode driver on VB6|Multithreading in StandartEXE|Native VB6-DLL
Code injection to other process|DLL injection|DirectX9|DirectSound|TrickControls|MP3 player from memory
Easy calling by pointer|Hash-table|Safe subclassing|Vocoder|Creation of native DLL|COM-DLL without registration
FM-synthesizer|FFT spectrum-analyser|3D Fir-tree|Asynch waiter|Wave steganography|Fast AVI-trimmer
Windows with custom rendering|String to integer and vice versa|Multithreading with marshaling|Owner-draw listbox mod
Advanced math|Library info|Create GIF-animation|Store data to self-EXE|Computer creates a music|TrickSound class

Support me:
YandexMoney: 410011032160614
BTC: 1B9xD98ThjstyQUP2BUz9sBYhWT9rzqWfe

(rate this post) Reply Reply With Quote

+ Reply to Thread Quick Navigation CodeBank - Visual Basic 6 and earlier Top

Quick Reply

Post Quick Reply | Go Advanced | Cancel

« Previous Thread | Next Thread »

VBForums > VBForums CodeBank > CodeBank - Visual Basic 6 and earlier > [VB6] Loader, shellcode, without runtime...

Thread Information

There are currently 1 users browsing this thread. (1 members and 0 guests)

dz32

Tags for this Thread

loader, shellcode

View Tag Cloud

Posting Permissions

You may post new threads	**BB code** is On
You may post replies	**Smilies** are On
You may post attachments	**[IMG]** code is On
You may edit your posts	**[VIDEO]** code is On
	HTML code is Off
	Forum Rules

Contact Us VB Forums Top

All times are GMT -4. The time now is 10:58 AM.