```
*******************************************************************************
*                      DISASSEMBLING VISUAL BASIC APPLICATIONS                *
*                                          - Sanchit Karve                    *
*                                                                             *
*                                            born2c0de                        *
*                                      printf("I'm a %XR",195936478);         *
*                                                                             *
*                  CONTACT ME :          born2c0de AT hotmail DOT com         *
*                                                                             *
*******************************************************************************
```

LAST UPDATED : 26 JULY 2006

INDEX [INDX]

I. DISCLAIMER AND NOTICE [DISN]

The information provided in this tutorial must not be used for Reverse Engineering any application.

THE TEXT HAS BEEN WRITTEN IN SUCH A WAY THAT THE READER CAN LEARN, AND NOT JUST GAIN INFORMATION WITHOUT KNOWING HOW STUFF WORKS.

If the Reader still chooses to break Protection Mechamisms after reading this tutorial, he/she shall alone be responsible for the damages cause and not the Author.

If you wish to post certain Sections of the Tutorial on a Website, you are free to do so provided you inform the author and publish the Selected Text from the Tutorial as it is without modification.

The Author has not copied text or any other information directly from a Source. However, some information from some sources has been used to write this tutorial. These Sources have been mentioned in the References Section.

You are permitted to continue reading the tutorial only if you agree to the text given above.

II. READING THIS TUTORIAL [RTUT]

Each Section in this Tutorial has a specific Topic Code enclosed in square brackets. This arrangement has been made so that you can jump to a specific topic simply by searching for the topic code from your Browser.

At many places in the tutorial, I've explained a few things which are almost unnecessary to know when dealing with Visual BASIC programs but I've written them for those who are interested in Hacking (And I don't mean that 'hack-an-email-address' sort of a kid. The original meaning of Hacking has been ruined by pathetic people like them. Hacking in literal terms stands for 'curiousity'.
The original meaning of a Hacker is:
"A person who enjoys exploring the details of programmable systems, as opposed to most users, who prefer to learn only the minimum necessary."

The extra details I've given in this tutorial are for those who want to be such ethical hackers.

The Topic Code [XTRA] and [/XTRA] has been given for marking "extra-information" sections and you are free to skip such sections. Text within the [XTRA]..[/XTRA] blocks is given for extra information.
You can search for the Extra Information using the Topic Code.

III. INTRODUCTION [ITRO]

As long as you know Assembly Language, it is easy to read disassembled listings of executable files written in C/C++ or PASCAL, especially if you are using IDA Pro as your disassembler. This is so because C and C++ Compilers generate (or at least try to) efficient code. Some Compilers like Borland C++ use simple instructions for complex operations(also remember that this is not always the case) which make it easier to study them. Implementation of Code Constructs such as loops, IF statements, Ternary IF statements, switch constructs etc. can be found very easily as each one is unique and distinct.

However the same is not true for Applications written in Visual BASIC. VB Programs are said to be very slow and hence deliver poor performance. There is a reason for this. Visual BASIC programs unlike those written in other languages don't use Windows API Directly. Local functions present in VB Runtime Files are called which call functions from the Windows API.Most of the Visual BASIC functions are present in MSVBM60.DLL (if you've got Runtime Files ver. 6.0).
So to study VB programs, we must disassemble and analyze the MSVBM60.DLL file as well.
Since VB programs use such a complex API Function call procedure, programs tend to run slower.(There are other reasons as to why VB programs run slower but I won't be covering it as it's off-topic.)

It becomes difficult to analyze VB Programs as it uses functions which are not part of the Windows API and hence we are not acquainted with them.

My primary aim in this Tutorial is to teach the reader how to understand disassembled listings of programs written in Visual BASIC.

My secondary aim is to help you realise why Visual BASIC is not suitable for writing small,fast and efficient programs.

Almost all authors of Visual BASIC books mention that Visual BASIC does not give you applications with good performance.

This tutorial tells you why.

The Tutorial will talk about executable files compiled in Visual BASIC in Native Code ONLY and not p-code.

After reading this tutorial, you should be able to disassemble,debug and understand Visual Basic Applications. You may also be able to reverse engineer Protection Mechanisms written in Visual BASIC and that's where the next section comes in.

IV. ASSUMPTIONS [ASPT]

You are required to have a basic understanding of Visual BASIC,C,the Windows API and 80x86 Microprocessor Assembly Language.
It would be advisable to have a copy of Intel's 80x86 Instruction Set Manual. Intel provides this manual free of charge. If you need this manual, contact me.

This Instruction Set Reference is Volume 2 of Intel Architecture Software Developer's Manual.

The Software Developer's Manual consists of 3 volumes:
: Basic Architecture - Order Number 243190
: Instruction Set Reference - Order Number 243191
: System Programming Guide - Order Number 243192

You can provide these Order Numbers to get a copy of these manuals. For this tutorial only Volume 2 is required.

V. REQUIRED TOOLS [RQRT]

You will need the following tools to proceed with the Tutorial.

* COMPILER : Visual Basic 6.0
* DISASSEMBLER : IDA Pro 4.x or higher
* DEBUGGER : OllyDebug Ver. 1.09d or higher
* WINDOWS API DOCUMENTATION
* NuMeGa SmartCheck 6.x
* VBDE version 0.85 by iorior
* VBReformer

VBDE is not required but it's always better to have it as it gives addresses of entry-points of most VB procedures.

VBReformer is used to see the Property values of all objects in a Visual Basic Form. It even allows you to change the value of Object Properties such as Forms, Command Buttons etc. Import Libraries can also been seen. This Application is not required for this Tutorial but it's better to have it.

NuMeGa SmartCheck is again not required but it is useful when we have no idea what a particular procedure of VB does. You can run a program from it like a Debugger and view its log files and find out which procedure is called and what operations are carried out etc.

You can have API Documentation from MSDN or you can use the API Text Viewer Tool supplied with Visual Studio or browse MSDN Online (msdn.microsoft.com). Certain Applications like APIViewer will also do.

I have given the names of the Tools that I have used. But you are free to use any disassembler and debugger as long as you are comfortable using it but I advice you to use the tools that I have used above. SoftIce is better than OllyDebug but the latter is good enough for VB Programs so it doesn't matter which one you use.

Once you have the necessary knowledge and tools, you can proceed further. Let's begin.

VI. STRUCTURE OF A VB PROGRAM [SVBP]

When you open a VB Program from IDA, you'll end up with the following code.

```
start:
                push    offset dword_4012B4
                call    ThunRTMain
 ; ---------------------------------------------------------------------------
                dd 0, 300000h, 400000h, 0, 0E9960000h, 82E6FCDFh, 939C4C23h
                dd 0EB969B2Fh, 73D5h, 0, 1, 34303230h, 72503033h, 63656A6Fh
                ; etc etc etc
```

This doesn't make any sense does it? If you keep scrolling further you will see sections of code and data. Each Section has a meaning in VB Programs and you can see a general idea of a Visual BASIC program's Section Map below.

```
00401000:
... IAT (First Thunk ok apis)

Next Section(NS):
... some data

NS:
... transfer area (Jumps to imported functions)

NS:
... lots of data

NS:
... local transfer area (for internal event handlers)

NS:
... other data

NS:
... code

NS:
... lots of data

NS:
... .data Section
```

Let us now start analysis from the entry point of the program.

```
push    offset RT_Struct
```

```
call    ThunRTMain
```

It's C equivalent would have been:
```
ThunRTMain(&RT_Struct);
```

A function ThunRTMain is called which accepts one parameter. We'll soon find out that the parameter is a structure.
Simply putting a step over command on the CALL statement results in the execution of the Application.
Wierd Isn't it?
For Pascal,C and C++ Programs there is always a start() function that takes all CommandLine Parameters,Gets ProcessThreads,Module Handles etc. We didn't see anything of the sort in a Visual BASIC Program.

But actually, VB does have a start function. The start function code is placed in the ThunRTMain Function. Let's verify that by disassembling the MSVBM60.DLL and viewing the ThunRTMain Function. I've mentioned only a part of the ThunRTMain Function Code.

```
 ThunRTMain      proc near

 arg_0           = dword ptr  8

                 mov     esi, [ebp+arg_0]
                 mov     dword_7352F7DC, esi
                 and     [ebp+var_4], 0
                 lea     eax, [ebp+StartupInfo]
                 push    eax             ; lpStartupInfo
                 lea     eax, [ebp+StartupInfo]
                 push    eax             ; lpStartupInfo
                 call    ds:GetStartupInfoA
                 movzx   eax, [ebp+StartupInfo.wShowWindow]
                 mov     dword_7352F7D8, eax
                 push    hModule
                 push    esi
                 mov     esi, offset dword_7352F470
                 mov     ecx, esi
                 call    sub_7342DECD
                 mov     [ebp+var_1C], eax
                 test    eax, eax
                 jl      short loc_7342DEC5
                 push    0               ; lParam
                 push    0               ; wParam
                 push    1069h           ; Msg
                 call    ds:GetCurrentThreadId
                 push    eax             ; idThread
                 call    ds:PostThreadMessageA
                 ; Other Code

                 or      [ebp+var_4], 0FFFFFFFFh
                 push    0               ; uExitCode
                 call    ds:ExitProcess
                 jmp     loc_734619B3

 loc_7342DEC5:                           ; CODE XREF: ThunRTMain+60□j
                 push    eax
                 call    sub_Free_Memory
                 jmp     short loc_7342DEB4
endp
```

As you can see, it does call all the Functions that the start() function does in C and PASCAL programs. But what about CommandLine() Function from KERNEL32.DLL? MSVBM60.DLL does call that function as well but that function call is placed in

deeply nested function calls. You can open the Imports Window to see the Imported Function and see the cross-reference to a procedure in MSVBM60.DLL

The sub_Free_Memory procedure calls various API Functions but if you keep reading the procedure, you'll soon come across the HeapFree() Function which is imported from kernel32.dll.

Now I guess you now know the purpose of the ThunRTMain Function.
Let us now see what structure is passed to it.

If we double-click on the RT_Struct offset, we reach an address containing certain values.
It is a huge structure and each part needs to be seen one at a time.

Explaining the Structure will take up a lot of time and since I want to focus on the Code Constructs of Visual BASIC, I won't explain the Structure Passed to ThunRTMain.

All I can tell you is that the structure contains the PE (Portable Executable) Header Details. It is this header that is read by Resource Editors.

I found a good source for understanding the structure that is passed to ThunRTMain and I suggest you read it if you are interested in knowing PE Header Details. The link to the Article is given in the References [REFR] Section. The article is titled "VISUAL BASIC REVERSED - A decompiling approach" and is written by Andrea Geddon.
If the link is dead by the time you are reading this, you can contact me on my email address to get the article.

VII. OUR FIRST PROGRAM [OFPR]

Create a Form with a CommandButton. Click the CommandButton and add a simple Msgbox Code as shown below:

```
Private Sub Command1_Click()
        Msgbox "Ssup"
End Sub
```

Open the Compiled EXE File with IDA Pro.
Click the Strings Tab to find the "Ssup" String.
Double-Click the String to find its cross-reference.
Scroll up to the top of the procedure.
You should see something like this:

[Explanation is partly given by comments after an instruction.]

```
Command1_Click  proc near

 var_64          = dword ptr -64h
 var_5C          = dword ptr -5Ch
 var_54          = dword ptr -54h
 var_4C          = dword ptr -4Ch
 var_44          = dword ptr -44h
 var_3C          = dword ptr -3Ch
 var_34          = dword ptr -34h
 var_2C          = dword ptr -2Ch
 var_24          = dword ptr -24h
 var_14          = dword ptr -14h
 var_C           = dword ptr -0Ch
 var_8           = dword ptr -8     ; Destructor Object
 var_4           = dword ptr -4
 form_object     = dword ptr  8
```

```
push    ebp                     ; These two instructions
mov     ebp, esp                ; open the Stack Frame.
sub     esp, 0Ch                ; Allocates 12 bytes on stack
push    (offset exception_handler+1); Starts Exception Handler
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
sub     esp, 88h                ; Allocates 136 bytes on stack
push    ebx
push    esi                     ; Saves Values of Registers
push    edi

                                ; Loads the Destructor
mov     [ebp+var_C], esp
mov     [ebp+var_8], offset destructor

                               ; Allocating Dynamic Resources
mov     eax, [ebp+form_object]
mov     ecx, eax
and     ecx, 1
mov     [ebp+var_4], ecx
and     al, 0FEh
push    eax
mov     [ebp+form_object], eax
mov     edx, [eax]
call    dword ptr [edx+4] ; Calls MSVBM60.Zombie_AddRef
mov     ecx, 80020004h
xor     esi, esi
mov     [ebp+var_4C], ecx
mov     eax, 0Ah
mov     [ebp+var 3C], ecx
mov     [ebp+var_2C], ecx
mov     [ebp+var_34], esi
mov     [ebp+var_44], esi
mov     [ebp+var_54], esi
mov     [ebp+var_64], esi
lea     edx, [ebp+var 64]
lea     ecx, [ebp+var_24]
mov     [ebp+var_24], esi
mov     [ebp+var_54], eax
mov     [ebp+var_44], eax
mov     [ebp+var_34], eax
mov     [ebp+var_5C], offset aSsup ; "Ssup"
mov     [ebp+var_64], 8
call    ds:__vbaVarDup
lea     eax, [ebp+var_54]
lea     ecx, [ebp+var_44]
push    eax
lea     edx, [ebp+var_34]
push    ecx
push    edx
lea     eax, [ebp+var_24]
push    esi
push    eax
call    ds:rtcMsgBox          ; Calls the MsgBox Function
lea     ecx, [ebp+var_54]
lea     edx, [ebp+var_44]
push    ecx
lea     eax, [ebp+var_34]
push    edx
lea     ecx, [ebp+var_24]
push    eax
push    ecx
push    4
```

```
                call    ds:__vbaFreeVarList
                add     esp, 14h
                mov     [ebp+var_4], esi
                push    offset continue_after_jump
                jmp     short fake_a_call_instr

                lea     edx, [ebp+var_54]
                lea     eax, [ebp+var_44]
                push    edx
                lea     ecx, [ebp+var_34]
                push    eax
                lea     edx, [ebp+var_24]
                push    ecx
                push    edx
                push    4
                call    ds:__vbaFreeVarList
                add     esp, 14h
                retn
; ---------------------------------------------------------------------------

fake_a_call_instr:
                retn
; ---------------------------------------------------------------------------

continue_after_jump:
                mov     eax, [ebp+arg_0]
                push    eax
                mov     ecx, [eax]
                call    dword ptr [ecx+8]  ; Calls MSVBM60.Zombie_Release
                mov     eax, [ebp+var_4]
                mov     ecx, [ebp+var_14]
                pop     edi
                pop     esi
                mov     large fs:0, ecx
                pop     ebx
                mov     esp, ebp
                pop     ebp                ; Closes Stack Frame
                retn    4
Command1_Click  endp
```

Simply by looking at the entire procedure you can't exactly figure out what the hell happens when the whole subroutine is executed. If you know Assembly well and have had the patience to read through the code, you should notice a few neat things in the code.

[XTRA]

Before I begin explaining the procedure, I want to teach you how to recognise a procedure in Visual BASIC. They can be called Procedure Signatures.
1) A Procedure has the open and close Stack Frame instructions.
2) The First Procedure in a VB Program is always preceded by
   12 0xCC Bytes (which corresponds to the INT 3 Instruction) followed by
    4 'T' bytes (0xE9) followed by 12 0xCC bytes.
3) Procedures other than the first are preceded by 10 NOP(0x90) Instructions.

: 1) STACK FRAME:

The Open/Close Stack Frame Instructions are even found in C/C++ and Pascal programs and hence can be termed as a universal method of determining procedures. However that is not always the case.

--> Many compilers just JMP instructions to fake a Call Instruction. This Jump
    is at times a CALL to a procedure. IDA Pro does not detect such

CALL 'emulating' instructions but OllyDebug does recognise such
code patterns.

--> Visual C++ allows the programmer to write naked functions. Naked functions
mean that the compiler does not allocate space for its arguments nor does it
include the stack open and close frame instructions.

But since we are dealing with Visual BASIC, we can ignore the second case. You will see an example of the first case shortly.

: 2) THE 0xCC BYTE

The 0xCC Byte is used to Generate the INT 3 Exception, which is known as the "CALL TO INTERRUPT" Procedure. It is used by Debuggers such as OllyDebug and SoftIce to set software Breakpoints. Debuggers insert the 0xCC byte before the instruction which it wants to set a breakpoint on. As soon as the INT 3 Instruction is executed, Control is passed onto the Debuggers Exception Handler.

Here is the description taken directly from Intel's Software Developers Manual Volume 2 : Instruction Set Reference.
"The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code). To further support its function as a debug breakpoint, the interrupt generated with the CC opcode also differs from the regular software interrupts as follows:
- Interrupt redirection does not happen when in VME mode; the interrupt is handled by a protected-mode handler.
- The virtual-8086 mode IOPL checks do not occur. The interrupt is taken without faulting at any IOPL level."

That's how debuggers work. That's also the concept of certain anti-debugging techniques. Since the 0xCC code is injected by Debuggers before an instruction, the CRC (Cyclic Redundancy Check) Value of the code also changes. Some Antidebugging techniques encrypt the program with a key which is the CRC value of the program. When a program is being debugged, its CRC value changes and with the result the program doesn't get decrypted.
Such methods are effective in stopping amateur wannabe hackers from understanding their code but its not foolproof and an expert hacker can get past this technique with ease.

So much for what '0xCC' is. But why is it placed before the First Procedure in VB Programs?
I've found no answer to that so far. This wastes a lot of space in a program.

If you try to disassemble a Console Program written in Visual C++, you'll find many instructions which set parts of the stack to the '0xCC' value. You will also find 0xCC bytes scattered across the disassembled listing.

If only Visual Studio was Open Source, we could have seen the code generation code and come up with an answer and improve the code generation code too.
I hope you also realise why Open Source is slowly gaining momentum.

3) The 0x90 Byte

Here is the description taken directly from Intel's Software Developers Manual Volume 2 : Instruction Set Reference.

"Performs no operation. This instruction is a one-byte instruction that takes up space in the instruction stream but does not affect the machine context, except the EIP register.
The NOP instruction is an alias mnemonic for the XCHG EAX, EAX instruction."

This byte is injected into serial generation/checking procedures by amateur hackers where the protection mechanism is weak. This is known as bit-hacking. Sadly enough, bit-hacking STILL works for defeating plenty of today's Commercial Applications. Guess they never realised the importance for code-security.

While writing programs in Assembly Language, if you use Forward Referencing in a few situations or use a wrong Jump Instruction to jump to certain addresses, chances are quite bright that the Assembler will fill in some bytes with the NOP instruction.
As a result, having the presence of the 0x90 Instruction in your code is considered bad programming.

But again, I see no reason why the 0x90 Byte is present in Visual BASIC. Removing such entries will reduce the executable size drastically.

Programs like VBDE rely on such Procedure Signatures to identify where a procedure is present.

[/XTRA]

Let us start by analyzing the procedure in portions.
First the Procedure opens the Stack Frame. Then it allocates 12 Bytes on the stack for the Destructor and other variables. (We shall see the Destructor in detail after a short while.)
Then it allocates Dynamic Resources and calls the Zombie_AddRef Function.

What does the Zombie_AddRef Function do? It Takes the Object Reference.
In this function the parent object (in this case Form) is passed as a parameter and uses AddRef to increment reference count of the object (instantiation).
Since COM objects are responsible for their lifetime, the resources they use are allocated until the reference count is 0, when it reaches 0 the objects enter zombie state & can be deallocated to free resources.
Refer COM object management documentation for more detailed information.

Right after the call of the Zombie_AddRef Function there are MOV instructions which assigns values to many variables. That follows a reference to the "Ssup" string followed by a call to the rtcMsgbox procedure.

Why does it seem so wierd? Shouldn't it simply call the rtcMsgbox Function?

Let us find out why in a little more interesting manner.
Intuition tells us that no matter what the function does, it will end up calling the MessageBoxA or the MessageBoxW Function. So let's set a breakpoint on the MessageBoxA and MessageBoxW Functions.

To do that, start OllyDebug and load the Executable file by pressing F3.
After the program is loaded, press Alt+E to open the Executable Modules Window.
Double click USER32.DLL to open the disassembled listing of the User32.dll file.
From there press Ctrl+N to open the Imports/Exports Window. Then Scroll over till you see the MessageBoxA and MessageBoxW Functions. Click them one at a time and press F2 to set a breakpoint.

Now press F9 to run the program. The Application should open. Click the CommandButton. Now instead of the Debugger halting at a breakpoint of MessageBox, the MessageBox comes up without any halt to the Debugger.

Why does this happen? Does this mean that rtcMsgBox has a seperate copy of the MessageBox code within itself? Though it seems like a possible reason, it is unlikely to happen as Microsoft Developers built the Windows API so that they could be reused for performance. So that means that some API Function is called which displays the MessageBox.
So let us try another experiment. In the same Imports/Exports Section of User32.dll we see 2 more MessageBox functions which are MessageBoxIndirectA and

MessageBoxIndirectW. Let's try setting a breakpoint on both these Messages.

After the breakpoint is set, press F9, and click the Command Button.
This time, the Debugger halts at the MessageBoxIndirectA function.
Interesting isn't it? All Visual BASIC Applications which use the Msgbox()
Function are actually calls to MessageBoxIndirectA and not MessageBox as thought.

This is an important characteristic. So the Next time you set a breakpoint on
the MessageBox function and the debugger halts at a breakpoint, you can be
pretty sure that someone has used the MessageBox() API Directly by consulting
the API Text Viewer for the VB Declaration.

Let us now see the prototype of the MessageBoxIndirect() API Function.

```
Private Declare Function MessageBoxIndirect Lib "user32" Alias
"MessageBoxIndirectA" (lpMsgBoxParams As MSGBOXPARAMS) As Long
```

Only One Parameter? So then how is the Message Body and Title passed to the
Function? For that we'll need to see the declaration of the MSGBOXPARAMS
Structure.

```
Private Type MSGBOXPARAMS
        cbSize As Long
        hwndOwner As Long
        hInstance As Long
        lpszText As String
        lpszCaption As String
        dwStyle As Long
        lpszIcon As String
        dwContextHelpId As Long
        lpfnMsgBoxCallback As Long
        dwLanguageId As Long
End Type
```

This suggests that the required parameters are assigned to variables and the
reference to that object is passed to that function.
So That suggests that the many MOV instructions found before the rtcMsgbox call
are used to initialise the MSGBOXPARAMS Structure.

To confirm our doubt, let's compare the MOV instructions with the code found
before the MessageBoxIndirect function is called.

```
  mov     edx, [eax]
  mov     [ebp+hWnd.lpszText], ecx
  mov     ecx, [eax+8]
  mov     eax, [eax+0Ch]
  push    esi
  push    ebx
  test    ah, 40h
  mov     [ebp+hWnd.hInstance], edi
  mov     [ebp+hWnd.lpszIcon], edi
  mov     [ebp+hWnd.lpfnMsgBoxCallback], edi
  mov     [ebp+hWnd.cbSize], 28h
  mov     [ebp+hWnd.hwndOwner], edx
  mov     [ebp+hWnd.lpszCaption], ecx
  mov     [ebp+hWnd.dwStyle], eax
  mov     [ebp+hWnd.dwLanguageId], edi
  jz      short loc_734A6133
  mov     [ebp+hWnd.lpfnMsgBoxCallback], offset sub_734A6098

loc_734A6133:

  mov     esi, ds:MessageBoxIndirectA
```

```
  lea     eax, [ebp+hWnd]
  push    eax             ; LPMSGBOXPARAMSA
  call    esi ; MessageBoxIndirectA
```

Interesting to see that....Isn't it?

Next comes the __vbaFreeVarList Function. From its name we can see that it deallocates the address of a certain number of variables. This function actually does no work except call the __vbaFreeVar Function multiple number of times.

Let us see how both functions work.

__vbaFreeVar : Frees a Temporary Variable.

__vbaFreeVar accepts only 1 Argument, which is the address of the variable to be deleted. This argument is ALWAYS passed through ECX.
Uses the API Function __imp_SysFreeString()[Ordinal Number 6] from OLEAUT32.DLL that carries out the actual deallocation of a variable.

__vbaFreeVarList : Frees Temporary Variables.

Have a look at this Snippet:

```
                lea     ecx, [ebp+var_54]  ; Variable 1 stored in ecx
                lea     edx, [ebp+var_44]  ; Variable 2 => edx
                push    ecx                ; Variable 1 pushed
                lea     eax, [ebp+var_34]  ; Variable 3 => eax
                push    edx                ; Variable 2 pushed
                lea     ecx, [ebp+var 24]  ; Variable 4 => ecx
                push    eax                ; Variable 3 pushed
                push    ecx                ; Variable 4 pushed
                push    4                  ; Free 4 Temp. Variables
                call    ds:__vbaFreeVarList
```

The code is pretty easy to understand. This function frees temporary variables that are passed as arguments to it.Interestingly each memory location is 16 bytes wide.
This is an interesting function as it can accept variable arguments.
It's equivalent function call in C would be:
__vbaFreeVarList(4,&var_24,&var_34,&var_44,&var_54);

where its declaration would be:

```
int __vbaFreeVarList(int NUMBER_OF_VARIABLES_TO_FREE,<addresses of the vars>...)
{
   // CODE
}
```

If we analyse the code of __vbaFreeVarList, we actually find multiple calls of __vbaFreeVar. Have a look at the source code of __vbaFreeVarList.

```
                public __vbaFreeVarList
__vbaFreeVarList proc near

 arg_0          = dword ptr  4
 arg_4          = dword ptr  8
 arg_8          = dword ptr  0Ch

                mov     ecx, [esp+arg_4]     ; Address of Variable to delete
                push    esi                  ; Saves value of esi
                lea     esi, [esp+4+arg_8]   ; esi = Address of Next Variable
                call    __vbaFreeVar         ; function call
```

```
                mov     eax, [esp+arg_4]     ; eax gets value of num of
                                             ; variables to be freed
                cmp     eax, 1               ; If eax<=1 then
                jbe     short freed_all_vars ; jump to end of function.
                push    edi                  ; Value of edi is saved on stack
                lea     edi, [eax-1]         ; edi=eax-1

            ; THIS IS A WHILE LOOP :  while(edi){ /*CODE*/ }
 loop_start:
                mov     ecx, [esi]          ; ecx=Address of Variable to Delete
                add     esi, 4              ; esi = Address of Next Variable
                call      vbaFreeVar        ; function call
                dec     edi                 ; edi--;
                jnz     short loop_start    ; Jump to beginning of the loop if
                                            ; edi is not ZERO.
                                            ; Well Written Code.No need for a
                                            ; CMP Instruction as DEC
                                            ; Instruction affects the ZERO Flag
                pop     edi                 ; Value of edi is restored.

 freed_all_vars:
                pop     esi                 ; Value of esi is restored.
                retn                        ; Return to the calling function
ENGINE:7352009D __vbaFreeVarList endp
```

As you can see, __vbaFreeVarList uses a while loop to free each variable one by one using the __vbaFreeVar Function.
Notice that the address of the variable to be freed is stored in ECX always.
You can disassemble the __vbaFreeVar Function to confirm that.

Now let us see what happens when after the MessageBox is shown.
This is the most interesting part.

After the MessageBox is displayed, a clean up code is executed that deallocates all the variables used in the entire procedure. Have a look at these statements in the Command1_Click() Code.

```
                push    offset continue_after_jump
                jmp     short fake_a_call_instr
 ; ---------------------------------------------------------------------------

 fake_a_call_instr:
                retn
 ; ---------------------------------------------------------------------------

 continue_after_jump:
                    ; code
```

This is the 'CALL-Simulation' instruction. If you recall, before a call function is executed, the processor pushes the location of the instructions which are supposed to receive control after execution of a function is over.
VB instead of issuing a call instruction simulates it using the push, jmp and retn instructions. It is small sections of code like this that reduce Visual BASIC's efficiency and performance.

Let us still see why this is done.

The CALL-Simulation Instruction calls the MSVBM60.Zombie_Release function.
This is the destructor code. Doesn't that remind you of something?
The instruction

```
    mov     [ebp+var_8], offset destructor
```
contains the offset of the destructor code. But is this so?
Double-click on the 'offset destructor' text and you'll land up here.

```
destructor      dd 80005h, offset loc_401A7E, 0, offset loc_401A85, 102825FFh
```

Hmm...this contains more offsets? By simply double-clicking the offsets you land up at the destructor code again. That's why the CALL simulation code is used so that the destructor code looks like its an inline function.

If you're more curious, you can also double-click the 'exception_handler' text to see where that leads to.

Well, after a long journey into the Command1_Click() Procedure, we're finally done analyzing it.

From this point onwards, I shall explain only the important section of code rather than explain such intricate details once again.

Let us proceed further.
This Time let us create a Visual BASIC Application using only a module.
We shall use the Main Subroutine.

Use this code:

```
Sub Main()
    MsgBox "Ssup"
End Sub
```

What you will realise that the Procedure code is an exact copy of the code we dealt with earlier. This means that Form Procedures and Module Procedures are treated alike. This also means that the Command Button code procedure had no chance of using any information of the Form Object.

Let's take another example.

VIII. STRING COMPARISON [STR1]

Create a form without any controls. The code in the form module is as follows:

```
Private Sub Form_Load()
    If "Sanchit" <> InputBox("ssup") Then
        MsgBox "wrong"
    Else
        MsgBox "Right"
    End If
End Sub
```

The resultant code is shown below (after stripping unimportant instructions)

```
Form_Load       proc near

                ; Variables and Arguments shown


        push    ebp
        mov     ebp, esp
        sub     esp, 0Ch
```

```
push    (offset vba_exception_handler+1)
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
sub     esp, 0F0h
push    ebx
push    esi
push    edi
mov     [ebp+var_C], esp
mov     [ebp+var_8], offset destructor
mov     eax, [ebp+arg_0]
mov     ecx, eax
and     ecx, 1
mov     [ebp+var_4], ecx
and     al, 0FEh
push    eax
mov     [ebp+arg_0], eax
mov     edx, [eax]
call    dword ptr [edx+4] ; Zombie_AddRef()
xor     eax, eax
mov     ebx, 80020004h
mov     edi, 0Ah

; code...

lea     edx, [ebp+var_98]
lea     ecx, [ebp+var_28]

; MOV [var],register Instructions

mov     [ebp+var 90], offset aSsup ; "ssup"
mov     [ebp+var_98], 8
call    ds:__vbaVarDup
lea     eax, [ebp+var_88]
push    offset aSanchit ; "Sanchit"

lea     ecx, [ebp+var_78]
push    eax
lea     edx, [ebp+var_68]
push    ecx
lea     eax, [ebp+var_58]
push    edx
lea     ecx, [ebp+var_48]
push    eax
lea     edx, [ebp+var_38]
push    ecx
lea     eax, [ebp+var_28]
push    edx
push    eax

call    ds:rtcInputBox
mov     edx, eax
lea     ecx, [ebp+var_18]
call    ds:__vbaStrMove
push    eax
call    ds:__vbaStrCmp
mov     esi, eax
lea     ecx, [ebp+var_18]
neg     esi
sbb     esi, esi
neg     esi
neg     esi
call    ds:__vbaFreeStr
lea     ecx, [ebp+var_88]
```

```
        lea     edx, [ebp+var_78]
        push    ecx
        lea     eax, [ebp+var_68]
        push    edx
        lea     ecx, [ebp+var_58]
        push    eax
        lea     edx, [ebp+var_48]
        push    ecx
        push    edx
        lea     eax, [ebp+var_38]
        lea     ecx, [ebp+var_28]
        push    eax
        push    ecx
        push    7
        call    ds:__vbaFreeVarList
        add     esp, 20h
        mov     [ebp+var_50], ebx
        test    si, si
        mov     [ebp+var_58], edi
        mov     [ebp+var_40], ebx
        mov     [ebp+var_48], edi
        mov     [ebp+var_30], ebx
        mov     [ebp+var_38], edi
        jz      short jump_if_right
        lea     edx, [ebp+var_98]
        lea     ecx, [ebp+var_28]
        mov     [ebp+var_90], offset aWrong ; "wrong"
        mov     [ebp+var_98], 8
        call    ds:__vbaVarDup
        lea     edx, [ebp+var_58]
        lea     eax, [ebp+var_48]
        push    edx
        lea     ecx, [ebp+var_38]
        push    eax
        push    ecx
        lea     edx, [ebp+var_28]
        push    0
        push    edx
        call    ds:rtcMsgBox
        lea     eax, [ebp+var_58]
        lea     ecx, [ebp+var_48]
        push    eax
        lea     edx, [ebp+var_38]
        push    ecx
        lea     eax, [ebp+var_28]
        push    edx
        push    eax
        jmp     short free_up_resources
 ; ---------------------------------------------------------------------------
jump_if_right:

        lea     edx, [ebp+var_98]
        lea     ecx, [ebp+var_28]
        mov     [ebp+var_90], offset aRight ; "Right"
        mov     [ebp+var_98], 8
        call    ds:__vbaVarDup
        lea     ecx, [ebp+var_58]
        lea     edx, [ebp+var_48]
        push    ecx
        lea     eax, [ebp+var_38]
        push    edx
        push    eax
        lea     ecx, [ebp+var_28]
        push    0
```

```
        push    ecx
        call    ds:rtcMsgBox
        lea     edx, [ebp+var_58]
        lea     eax, [ebp+var_48]
        push    edx
        lea     ecx, [ebp+var_38]
        push    eax
        lea     edx, [ebp+var_28]
        push    ecx
        push    edx

free_up_resources:

        push    4
        call    ds:__vbaFreeVarList
        add     esp, 14h
        mov     [ebp+var_4], 0
        push    offset call_emulate
        jmp     short jump_as_a_call
; ---------------------------------------------------------------------------
        lea     ecx, [ebp+var_18]
        call    ds:__vbaFreeStr
        lea     eax, [ebp+var_88]
        lea     ecx, [ebp+var_78]
        push    eax
        lea     edx, [ebp+var_68]
        push    ecx
        lea     eax, [ebp+var_58]
        push    edx
        lea     ecx, [ebp+var_48]
        push    eax
        lea     edx, [ebp+var_38]
        push    ecx
        lea     eax, [ebp+var_28]
        push    edx
        push    eax
        push    7
        call    ds:__vbaFreeVarList
        add     esp, 20h
        retn
; ---------------------------------------------------------------------------
jump_as_a_call:

        retn
; ---------------------------------------------------------------------------
call_emulate:

        mov     eax, [ebp+arg_0]
        push    eax
        mov     ecx, [eax]
        call    dword ptr [ecx+8] ; Zombie_Release()
        mov     eax, [ebp+var_4]
        mov     ecx, [ebp+var_14]
        pop     edi
        pop     esi
        mov     large fs:0, ecx
        pop     ebx
        mov     esp, ebp
        pop     ebp
        retn    4
Form_Load       endp
```

Let us first see the Prototype of the InputBox Function

```
Function InputBox(Prompt As String, Title As String, Default as String , _
```

```
        xpos As Long , ypos As Long, helpfile As String ,context As Long ) _
        As String
```

If we disassemble the MSVBM60.DLL File and scroll over to the rtcInputBox function, we can confirm this from the prototype that IDA provides us.

Most of the code is irrelevant to us but the important chunk of code of the function (which took me quite some time to find) is given below:

```
push    edi         ; Context Pushed
push    ebx         ; HelpFile Pushed
push    [ebp+arg_8] ; ypos Pushed
push    [ebp+arg_C] ; xpos Pushed
push    eax         ; Default pushed
push    ecx         ; Title Pushed
push    edx         ; Prompt pushed


call    sub_7349DC68    ; Function that causes Inputbox
                ; At this point EAX contains the entered string

push    [ebp+arg_4]     ; BSTR
mov     edi, eax
call    esi ; __imp_SysFreeString
push    [ebp+arg_0]     ; BSTR
call    esi ; __imp_SysFreeString
push    [ebp+var_4]     ; BSTR
call    esi ; __imp_SysFreeString
push    [ebp+var_1C]    ; BSTR
call    esi ;   imp SysFreeString
mov     eax, edi        ; Sets EAX to the Return Value
pop     edi
pop     esi
pop     ebx
leave                   ; Close Stack Frame
retn    1Ch             ; Returns from here
```

In the first portion of the code, the push instructions push all the seven parameters to the function that creates the InputBox Dialogbox. I know that with the current example that I'm disassembling, it's not quite possible to believe that all the push instructions stand for what I've mentioned. So what you can do is disassemble the following code given below and set a breakpoint on the PUSH instructions in the MSVBM60.DLL File using OllyDebug or SoftIce.

```
str1 = InputBox("This is prompt","This is Title","Default-Val",10,20, _
                                                "c:\help.hlp",1)
```

With this you can actually verify the contents of the push instruction to confirm what I've written.

Now, after the call instruction you can see a PUSH instruction pushing the contents of a variable. This is a parameter for the SysFreeString Function.

Next is a MOV instruction transferring the contents of the EAX register into EDI. EAX at this point of time contains the address of the String that we filled in the Text Box. This is done to save the value of EAX.

Then the SysFreeString Function is called. This function takes 1 argument which is the string that needs to be deallocated. This function does not return any value after execution.

Did you notice something stupid?

If the SysFreeString Function does not return any value, then why were the contents of EAX saved in EDI? And why is there a need to have a MOV EAX , EDI instruction?
What a waste of precious bytes.

This is deploring code generated by Visual C++ 6.0 Compiler (which was used to write the MSVBM60.DLL File) and since Visual BASIC programs use this routine, it results in slow, sluggish programs.
And this is just one function...imagine what would happen if we analyzed all of them?

Anyway, the original contents of the registers are restored, the stack frame is closed (with LEAVE) and the function returns after adding 0x1C bytes to ESP.

Now let's have a look at this code portion.

```
call    ds:rtcInputBox
mov     edx, eax
lea     ecx, [ebp+var_18]
call    ds:__vbaStrMove
push    eax
call    ds:__vbaStrCmp
mov     esi, eax
lea     ecx, [ebp+var_18]
neg     esi
sbb     esi, esi
neg     esi
neg     esi
call    ds:__vbaFreeStr
```

After the call of rtcInputBox, EAX contains the entered string in the Text Box of the InputBox dialog. The address of this string is moved to EDX.

Is this done to save the contents of the String for the compare function?
As obvious as it may seem, it is really not like that. Let us see.

The __vbaStrMove function moves a String from one place in memory to another place. On Analysis of the function code it is found that this function accepts two arguments and returns one value as shown below:

PARAMETERS:

EDX : Source String
ECX : Destination

RETURNS : Source String. Sets EAX to EDI (which holds value of EDX)

Now we see that by setting the value of EAX to EDX, we are setting the entered string as a SOURCE string to be copied into another location. This is neat.

Now the Source String is pushed again and the __vbaStrCmp Function is called. This on first glance looks wierd again. It seems that the StrCmp Function accepts only one argument. Then what does it compare it with?

If you scroll above you will find a push instruction that pushes the address of the string "Sanchit" (push   offset aSanchit ; "Sanchit")

Such cases remind us that unlike code generated by Pascal and C/C++ compilers, Visual BASIC functions can have it's arguments pushed anywhere and not right before the function call.
Keep this thing in mind when you set out to disassemble your own Visual BASIC programs.

Now after the Comparing function returns its value via the EAX register, it is copied to ESI.

Then the LEA instruction is used to load the address of a variable in the ECX register. Now since we are aware of VB's tricks we know that this is a parameter to the __vbaFreeStr function. You should notice that the same variable which held the value of the entered string is now being passed to this function to deallocate it as its not required after the comparison has been done.

Now let's talk about this code fragment:

```
neg     esi
sbb     esi, esi
neg     esi
neg     esi
call    ds:__vbaFreeStr
```

The NEG statement's actual use is to change the sign of a number for example, from 3 to -3.

But this one has an indirect use. This Instruction affects many flags. But the one it is meant for is the Carry Flag (CF) which is used in the next SBB Instruction.If ESI is equal to 0 then CF is reset to 0 and otherwise is set to 1.

Now the SBB Instruction stands for Subtraction with Borrow. In this case it can be translated to this:
ESI = ESI - (ESI + CF);

This is where the previous NEG instruction comes into picture.

The next two NEG instructions are of no use. You can take this as another example to show why VB code is slow.
Following this is the __vbaFreeStr Function which deallocates space for a string variable.

Now you might wonder that if comparison has been performed, then why isn't there any JUMP instruction?
If you keep reading the code you will find the TEST and JUMP Instructions after the seven variables used have been cleared. These instructions are found together in C/C++ and Pascal programs but in Visual BASIC this isn't always the case.

That's why Visual BASIC programs take up more time for analysis compared to C and Pascal programs.

I'm not discussing the rest of the code as it has been covered in the previous section.

Now let's discuss a Hack-tip.
This type of String Comparision protection is used in many popular Shareware Applications.
Removing such protection isn't much of a job and I'll discuss a cleaner way to hack this protection.

Remember the two useless NEG instructions before the __vbaFreeStr Function call? Since these instructions are of no practical use, we can replace these two bytes with XOR ESI, ESI which consumes only 2 bytes. That way the TEST instruction would always result in ZERO making control jump always to the Msgbox("Right") code.

A lot of Hackers use such 'useless' code to implement such hacking techniques.

IX. CONCLUSION [END1]

This isn't exactly the end of the tutorial. I'd rather say that it is the end for now. This topic is vast and I'm trying to include explanation and analysis of all Visual BASIC functions. If I plan to release this tutorial with all functions inclusive, it's going to take a lot of time.
So i've decided to post this tutorial first and keep updating it every 15 to 20 days. You can check the LAST UPDATED Section in the beginning of the Tutorial to see how recently this tutorial has been updated.
Keep checking for updated versions every month.

I hope you've enjoyed my tutorial as I've put in a lot of hard work and time on writing this.

Since I haven't come across any Books or Articles on this subject, I don't quite know what exactly is expected from my tutorial. I would appreciate it if you could email me suggestions and comments on this tutorial.
I may not be able to reply to every email, but I do read each one of them.

Thanks.

X. IN THE NEXT UPDATE [NXTU]

In the next update of this tutorial, I plan to give an explanation of which function is called when a certain Visual BASIC function is called. I then analyze the function, it's parameters and return value, methods to improve the function and much more.

XI. REFERENCES [REFR]

-> MSDN (msdn.microsoft.com)

-> "VISUAL BASIC REVERSED - A decompiling approach" by Andrea Geddon.
LINK (might be dead) : http://www.uploading.com/?get=8XO86WZN

****
[EOF]