Title:
Applied
Binary Code Obfuscation

Date:
January 21st 2009
Website:
http://www.astalavista.com

Author:
Nicolaou George
Mail:
ishtus<\at>astalavista<\d0t>com

Author:
Glafkos Charalambous
Mail:
glafkos<\at>astalavista<\d0t>com
Table of Contents
Introduction ..................................................................................................................... 3
Tools ............................................................................................................................... 3
Instruction Obfuscations ............................................................................................... 3
  EIP (the Instruction Pointer) .......................................................................................... 3
      return EIP; .................................................................................................................. 4
      CALL myself .............................................................................................................. 4
      “MOV EAX,EIP” ........................................................................................................ 5
Stack Based Obfuscation ............................................................................................... 5
  1. PUSH <value/register> ............................................................................................ 6
  2. POP <register/[memory]> ....................................................................................... 6
  3. CALL <address> ...................................................................................................... 7
  4. RETN ........................................................................................................................ 7
  5. MOV <register/[memory]>,<register/[memory]/value>* ........................................ 7
  6. JMP <address> ........................................................................................................ 8
Arithmetic and Logical Binary Obfuscation ................................................................. 8
  1. ADD <register/memory>, <value> ............................................................................ 9
  2. SUB <register/[memory]>, <value> ........................................................................ 10
  3. ADD <register/[memory]>,1 .................................................................................. 10
  4. SUB <register/[memory]>,1 .................................................................................. 10
  5. MOV <register>, 0 .................................................................................................. 11
  6. NOP ....................................................................................................................... 11
  7. NOT<register/[memory]> ...................................................................................... 12
  8. CMP <register>,0 ................................................................................................... 12
  9. NEG <register/[memory]> ..................................................................................... 12
 10. MOV <register>,<value> ........................................................................................ 13
Additional Obfuscations ............................................................................................... 13
      SAHF/LAHF ............................................................................................................. 13
      Polymorphism and Self Modifying Code ................................................................. 13
Example Software ....................................................................................................... 14
      Program Analysis .................................................................................................... 14
      Source Code ........................................................................................................... 14
      User Interface ......................................................................................................... 16
      Assembled Code ..................................................................................................... 16
      General Obfuscation approach .............................................................................. 18
Obfuscation Index Table ............................................................................................. 21
Introduction
An obfuscated code is the one that is hard (but not impossible) to read and understand. Sometimes corporate developers, programmers and malware coders for security reasons, intentionally obfuscate their software in an attempt to delay reverse engineering or confuse antivirus engines from identifying malicious behaviors. Nowadays, obfuscation is often applied to object oriented cross-platform programming languages like Java, .NET (C#, VB), Perl, Ruby, Python and PHP. That is because their code can be easily decompiled and examined making them vulnerable to reverse engineering. On the other hand, obfuscating binary code is not as easy as encrypting object or function names as it is done in programming languages mentioned above. In this case, the code is altered by using a variety of transformations, for instance self modifying code, stack operations or even splitting the factors of simple mathematical functions. Moreover, binary obfuscation is also used to defeat automated network traffic analyzers such like Intrusion Detection and Prevention Systems. In other words, binary code obfuscation is the technique of altering the original code structure and maintaining its original functionality. In the next pages of this paper we will explore the theory and practice of binary code obfuscation as well as a number of various techniques that can be used.

Tools
The tools used in this paper are the following:
- OllyDBG [http://www.ollydbg.de/]
- WinAsm Studio [http://www.winasm.net/]

Instruction Obfuscations
Obfuscation techniques aim to replace the instructions of a binary file or any other executable code with ones that have equivalent functionalities. Sometimes the size of those instructions is of no concern, since the main goal is to render the file or network traffic unreadable and undetectable. Therefore, obfuscated programs are most likely to be bigger than the original ones.

We can distinguish the methods used in two main categories for the purposes of this paper:
- The ones that use stack operations as a mean of obfuscation (Stack Based Obfuscation)
- The ones that use logical and arithmetical and logical instructions as a mean of obfuscation (Arithmetic and Logical Based Obfuscation)

EIP (the Instruction Pointer)
For the purposes of some obfuscation techniques we will need to retrieve the value of EIP. However we cannot directly reference it (wouldn’t life be easier if we could do so? I think not) thus we will have to find a way around. There are several ways of doing so but I will explain the most commonly used. Note that, the value of EIP is pushed into the stack each time you use the CALL instruction and we can use that to our advantage.
return EIP;

The simplest way to retrieve the value of EIP is to create a function that reads EIP and returns it.

At this point, EAX is equal to the value of EIP (OFFSET+05)

### MASM (WinAsm) Code Sample:

;;;;Code Omitted

```asm
getIP PROC
    pop EAX ;Place the value of EIP from the stack inside EAX
    push EAX ;Push the value back to the stack so we won’t affect the program flow
    retn
getIP EndP
;;;;Code Omitted
```

invoke getIP ;After executing this instruction, EAX will contain the value of EIP at this point

### CALL myself

Another way to get the value of EIP is to use a CALL function that calls itself

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bytes</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSET+00</td>
<td>E8 00 00 00 00</td>
<td>CALL &lt;MYSELF&gt;</td>
</tr>
<tr>
<td>OFFSET+05</td>
<td>58</td>
<td>POP EAX</td>
</tr>
</tbody>
</table>

### MASM (WinAsm) Code:

;;;;Code Omitted

```asm
db 0e8h ;We hard-code the actual bytes inside our code
dd 00000000h
pop eax ;After executing this instruction, EAX will contain the value of EIP at this point
```

### Assembled code:

```
08410000h 01 7D 0C 1101 CALL <MYSELF>
08410002h 75 54 POP EAX
08410004h 01 7D 10 E203 CALL <MYSELF> POP EAX
08410006h 75 56 POP EAX
```

TEXT ""
“MOV EAX, EIP”

MASM (WinAsm) Code Sample 2:

```
:Code Omitted
mov eax,$ ; After executing this instruction, EAX will contain the value of EIP at this point, the offset of the next instruction therefore will be EAX+5
```

Sample 2 assembled code:

![Sample 2 assembled code](image)

**Stack Based Obfuscation**

In this category, we will explore most techniques that use the stack as a mean of obfuscation. The stack is an abstract data structure based on the principle of FIFO (First In First Out). In our case the stack is similar to the following figure:

![Stack diagram](image)

The ESP (Stack Pointer) points to the first object inside the stack. As you can see the second item is located at ESP+4, the third at ESP+8 and so on. Once a POP instruction is executed, the first item is taken out the stack, and then the stack pointer (ESP) points at the second item. Once a PUSH instruction is executed then the pushed value is added into the stack and the stack pointer (ESP) points to it.

Stack based obfuscation manipulates the stack or references to it in order to succeed certain transformations of instructions.
1. PUSH <value/register>

We have explained above how the PUSH instruction works. What we need to do is find a way to replace this instruction with a different one that has the same functionality.

We need an instruction or a set of instructions that:
- Stores the value or the value of a register in the stack
- Makes ESP point to that value

Since we can directly affect the ESP register and reference to the stack meaning that we are able to write data to it we could easily replace the PUSH instruction with:

```
MOV DWORD PTR SS:[ESP-4],<value/register>
SUB ESP,4
```

or

```
SUB ESP,4
MOV DWORD PTR SS:[ESP],<value/register>
```

Still we can obfuscate the PUSH <value/register> instruction by storing the value to be pushed in another register or memory address

```
MOV <register>[memory],<value/register>
PUSH <register>[memory>
```

2. POP <register/[memory]>

The same goes with the POP instruction. We have the stack register (ESP) currently pointing at the value we would like to retrieve from the stack and therefore we can replace POP with the following set of instructions:

We store the value currently pointed by ESP into any register then balance the Stack

```
MOV <register>,DWORD[ESP]
ADD ESP,4
```

Additionally, instead of directly POPing the value to the desired register (or memory address) we first POP it in a predefined memory address and copy it over to our register

```
POP <[memory]>
MOV <register>,<[memory]>
```
3. CALL <address>

The CALL instruction is used for jumping into functions or system calls. In the processors level, the address of the next instruction is pushed into the stack for retrieval. After the function is finished executing and the RETN instruction is called, the value of the instruction pointer (EIP) changes and points to the address at the top of the stack.

Code:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Instruction</th>
<th>← OFFSET+05 is pushed into the stack and EIP changes to FUNCTION+XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSET+00</td>
<td>CALL &lt;PROCEDURE&gt; &lt;INSTRUCTION&gt;</td>
<td></td>
</tr>
<tr>
<td>OFFSET+05</td>
<td>CALL &lt;INSTRUCTION&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Function x:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Instruction</th>
<th>← The first value is POPed from the stack and placed into EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION+00</td>
<td>&lt;INSTRUCTION&gt;</td>
<td></td>
</tr>
<tr>
<td>FUNCTION+XX</td>
<td>&lt;INSTRUCTION&gt;</td>
<td></td>
</tr>
<tr>
<td>FUNCTION+XX</td>
<td>RETN</td>
<td></td>
</tr>
</tbody>
</table>

We can replace this instruction with:

```
PUSH EIP+X : where X is the number of bytes until the next instruction (in the case above that is EIP+05)
JMP <address> : where address is the offset of the function (in the case above that is FUNCTION+00)
```

(Note that: the example above can be combined with the PUSH obfuscations explained in section 1)

Another set of instructions we can use to replace the CALL instruction is:

```
MOV <[memory]/register>,<address>
CALL <[memory]/register>
```

4. RETN

The RETN instruction is used to return from a function or a system call as explained above in section 3.

We can easily replace this instruction with:

```
POP <register/[memory]>
JMP <register/[memory]>
```

(Note that: the example above can be combined with the PUSH and POP obfuscations explained in section 1 and 2)

5. MOV <register/[memory]>,<register/[memory]/value>*

*Note that MOV <[memory]>,<[memory]> is not a valid instruction but you can still use the obfuscation set of instructions provided below to achieve it.

The MOV instruction moves the value of the right side operand to the left side operand

eg: LeftOPER = RightOPER

This can be replaced with:
PUSH <register/[memory]/value> ; This is the right side operand
POP <register/[memory]> ; This is the left side operand

(Note that: the example above can be combined with the PUSH and POP obfuscations explained in section 1 and 2. Also in order to POP to a memory address you are required to have the appropriate permissions)

OR in some cases:
LEA <register a>,DWORD[<register b>]

6. JMP <address>
One of the most common opcodes that are obfuscated is the JMP instruction. JMP (Jump) or any other conditional jumps (JE, JNZ, JO, JA, etc) are used for controlling program flow under certain conditions or unconditionally. Ways of obfuscating such instructions are:

PUSH <address>
RETN
(Note that: the example above can be combined with the PUSH and RETN obfuscations explained in section 1 and 4)

CALL <address>
(Optionally at <address>, in order to balance the stack and get rid of the return address from the stack)
PUSH <register>
(Or)
ADD ESP,4
(Note that: the example above can be combined with the CALL and POP obfuscations explained in section 3 and 2)

Conditional Jumps can be obfuscated by using the SAHF and LAHF instructions (See page 13)

Arithmetic and Logical Binary Obfuscation

The Logical and Arithmetical obfuscation can be considered as less technical than the Stack Based Obfuscation. It involves the use of simple mathematical and logical functions like ADD, SUB, AND, NOT, NEG, OR, XOR, TEST, SHL, LEA, etc.

Every mathematical function f() can be written in many ways. It can be simplified or expanded in any way you like. For example:

\[ f(y) = 5 \times 4 + 10 \]
Can be written as:
\[ f(y) = (2+3) \times (2+2) + 9 + 1 \]

Or:
\[ i,j= 0; \]
\[ \text{inc}(y) = x+1 \]
\[ f(y) = \text{inc}(\text{inc}(\text{inc}(\text{inc}(i)))) \times \text{inc}(\text{inc}(\text{inc}(\text{inc}(j)))) + \text{inc}(j+i) \]
Additionally, a Boolean expression (x OR y):

<table>
<thead>
<tr>
<th>y</th>
<th>x</th>
<th>y ∨ x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Can be expressed as NOT(NOT(x) AND NOT(y)):

<table>
<thead>
<tr>
<th>y</th>
<th>x</th>
<th>¬(¬y ∧ ¬x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

1. **ADD <register/memory>, <value>**

The ADD instruction is simple it adds the second operand to the first. Cases:

- You have a register that holds a value X (ADD EAX,<value>)
- You have a register that points to a memory offset that holds a value X (ADD [EAX],<value>)
- You have a direct memory offset that holds a value X (ADD [<offset>],<value>)

In elementary math that is presented as:

\[ y = y + x \]

This is equal to:

\[ y = y - (-x) \]

In low level programming that can be written as:

```
SUB y,-x
```

This can be replaced with:

```
SUB <register/memory>, -<value>
```

Or by using LEA:

```
LEA <register>, DWORD [<register>+<value>]
```

(Note that: you cannot directly reference to a memory offset using the LEA instruction. Although, if needed you could load the memory value to a register, apply this instruction and finally put it back. Note that DWORD could also be BYTE or WORD)
2. SUB <register/[memory]>, <value>

Similarly, the ADD instruction, the same cases apply. SUB subtracts the second operant from the first. Therefore functions like:

\[ y = y - x \]

Can be expressed as:

\[ y = y + (-x) \]

This can be replaced with:

ADD <register/memory>, <value>

Using LEA:

LEA <register>, DWORD[<register> - <value>]

(Note that: you cannot directly reference to a memory offset using the LEA instruction. Although, if needed you could load the memory value to a register, apply this instruction and write it back)

3. ADD <register,[memory]>,1

Adding the value one to a register or a memory location can be also expressed by \( \text{NEG}(\text{NOT}(\text{value})) \). Note that this affects CF and AF flags (Carry Flag, Auxiliary Flag)

This can be replaced with:

\[ \text{NOT} <\text{register}/[\text{memory}> \]
\[ \text{NEG} <\text{register}/[\text{memory}> \]

Using LEA:

LEA <register>, DWORD [<register>+1]

4. SUB <register/[memory]>,1

Subtracting from a register or a value in memory the value 1

This can be replaced with:

\[ \text{NEG} <\text{register}/[\text{memory}> \]
\[ \text{NOT} <\text{register}/[\text{memory}> \]

Using LEA:

LEA <register>, DWORD [<register>-1]
5. **MOV <register>, 0**
Changing the value of a register to zero (0). Note that this could be combined with section 5 of Stack Based obfuscations.

This can be replaced with:

- \( \text{AND } <\text{register}>, 0 \)
- Or:
  - \( \text{SHL } <\text{register}>, 30 \)
  - \( \text{SHL } <\text{register}>, 30 \)
  - (Note that: SHL can affect Z, S and O flags)
  - Or:
    - \( \text{XOR } <\text{register}>, <\text{register}> \)
    - Or:
      - \( \text{SUB } <\text{register}>, <\text{register}> \)
      - Or:
        - \( \text{LEA } <\text{register}>, \text{DWORD}[0] \)

Additionally, changing the value of a(n) 16bit or 8bit register could also be achieved by:

- 16bit (for AX):
  - \( \text{AND } <\text{register}>, \text{FFFF0000} \)

- 8bit (for AL):
  - \( \text{AND } <\text{register}>, \text{FFFF00} \)

6. **NOP**
No operation, no impact on the CPU and memory. It only changes the EIP. NOP is equivalent to the instruction XCHG (E)AX, (E)AX.

This can be replaced with:

- \( \text{AND } <\text{register}/[\text{memory}]>, \text{FFFFFFFF} (-1) \) (3 byte with register, 7 byte with memory)
- \( \text{SHL } <\text{register}>, 0 \) (3 byte)
- \( \text{SHR } <\text{register}>, 0 \) (3 byte)
- \( \text{MOV } <\text{register}>, <\text{register}> \) (2 byte)
- \( \text{XOR } <\text{register}/[\text{memory}]>, 0 \) (3 byte with register, 7 byte with memory (need write permissions), also affects P, A, Z flags)
SUB <register/[memory]>,0
(3 byte with register, 7 byte with memory)

OR <register>,<register>
(3 byte with register, 7 byte with memory (need write permissions), also affects P, A, Z flags)

XCHG <register>
(2 byte if any register other than EAX is used, 1 byte otherwise)

LEA <register>,[<register>]
(2 byte)

FNOP
(2 byte affects C0, C1, C2, C3)

Multi-byte sequence of NOP instructions (available on processors with model encoding)

<table>
<thead>
<tr>
<th>Length</th>
<th>Assembly</th>
<th>Byte Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>NOP</td>
<td>90h</td>
</tr>
<tr>
<td>2 bytes</td>
<td>66 NOP</td>
<td>66 90h</td>
</tr>
<tr>
<td>3 bytes</td>
<td>NOP DWORD ptr [EAX]</td>
<td>0F 1F 00h</td>
</tr>
<tr>
<td>4 bytes</td>
<td>NOP DWORD ptr [EAX + 00H]</td>
<td>0F 1F 40 00h</td>
</tr>
<tr>
<td>5 bytes</td>
<td>NOP DWORD ptr [EAX + EAX*1 + 00H]</td>
<td>0F 1F 44 00 00h</td>
</tr>
<tr>
<td>6 bytes</td>
<td>66 NOP DWORD ptr [EAX + EAX*1 + 00H]</td>
<td>66 0F 1F 44 00 00h</td>
</tr>
<tr>
<td>7 bytes</td>
<td>NOP DWORD ptr [EAX + 00000000H]</td>
<td>0F 1F 80 00 00 00 00h</td>
</tr>
<tr>
<td>8 bytes</td>
<td>NOP DWORD ptr [EAX + EAX*1 + 00000000H]</td>
<td>0F 1F 84 00 00 00 00 00h</td>
</tr>
<tr>
<td>9 bytes</td>
<td>66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]</td>
<td>66 0F 1F 84 00 00 00 00 00h</td>
</tr>
</tbody>
</table>

7. NOT<register/[memory]>
This can be replaced with:
XOR <register/[memory]>, -1

Or:
NEG <register/[memory]>
SUB <register/[memory]>, 1

8. CMP <register>, 0
This can be replaced with:
OR <register>,<register>

AND <register>,<register>

TEST <register>,<register>

9. NEG <register/[memory]>
This can be replaced with:
NOT <register/[memory]>
ADD <register/[memory]>, 1
10. MOV <register>,<value>

This can be replaced with:

LEA <register>,[<value>]

Additional Obfuscations

SAHF/LAHF

These instructions are responsible for loading and storing the CPU flags (EFLAGS) SF, ZF, PF, AF, and CF to the AH register and storing them back.

These instructions execute as described above in compatibility mode and legacy mode. They are valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

LAHF

 Loads flags to the corresponding bits in AH register as shown below:

<table>
<thead>
<tr>
<th>SF</th>
<th>ZF</th>
<th>**</th>
<th>AF</th>
<th>**</th>
<th>PF</th>
<th>**</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

** Reserved bits

SAHF

Stores the contents of AH register to flags.

An example code for using SAHF and LAHF instructions

XOR EAX, EAX
XOR EBX, EBX
MOV AL, 1 ; Set EAX = 00000001 and EBX = 00000000
CMP EAX, EBX ; Compare EAX with EBX => PF = 0 ZF = 0
LAHF ; Store Flags in AH, EAX = 00000201
TEST EBX, EBX ; PF = 1 ZF = 1
SAHF ; Load Flags from AH => PF = 0 ZF = 0
JNZ <address> ; Jump if Not Zero (if ZF is equal to 0)

Polymorphism and Self Modifying Code

Polymorphism techniques can be used to encrypt/decrypt code. You can either use a stub to decrypt the whole code and execute it or decrypt sections of the code needed for execution and then discard them (a type of Self Modifying Code). Polymorphism and Self Modifying Code do not fall under the scope of this paper.
Example Software
Program Name: getWAN
Md5sum: 1b73530132dd7f66feae5f29beb7a6c60
Compiler: MASM (WinAsm)

Program Analysis

Source Code

getWAN.asm

.486
.model flat, stdcall
option casemap :none
include getWAN.inc

.code
start:
    invoke GetModuleHandle, NULL
    mov hInstance, eax
    invoke DialogBoxParam, hInstance, 101, 0, ADDR DlgProc, 0
    invoke ExitProcess, eax

DlgProc proc hWin :DWORD,
    uMsg :DWORD,
    wParam :DWORD,
    lParam :DWORD
    .if uMsg == WM_COMMAND
        .if wParam == GET
            invoke GetWANIP, addr Weburl
            invoke SetDlgItemText, hWin, RESULT, ADDR html
        .elseif wParam == EXIT
            invoke EndDialog, hWin, 0
    .endif
    .elseif uMsg == WM_CLOSE
        invoke EndDialog, hWin, 0
    .endif
    xor eax, eax
    ret
DlgProc endp

GetWANIP PROC url:DWORD
    local hInternet:DWORD
    local hURL:DWORD
    local hFile:DWORD
    local len:DWORD

    invoke InternetOpen, addr ipszAgent, INTERNET_OPEN_TYPE_PRECONFIG, 0, 0, 0
    mov hInternet, eax

    invoke InternetOpenUrl, hInternet, url, 0, 0, 0, 0
GetWANIP endp
mov hURL,eax

invoke InternetReadFile,hURL,addr html,sizeof html,addr len

invoke InternetCloseHandle,hURL
.if eax==0
    invoke InternetCloseHandle,hInternet
    Ret
.endif
Ret

GetWANIP EndP

GetWAN.inc

include windows.inc
include user32.inc
include kernel32.inc
include wsock32.inc
include wininet.inc

includelib user32.lib
includelib kernel32.lib
includelib wsock32.lib
includelib wininet.lib

DlgProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
GetWANIP PROTO :DWORD

GET equ 1001
EXIT equ 1002
RESULT equ 1003

.data
  Weburl db "http://www.whatismyip.com/automation/n09230945.asp",0
  ipszAgent db "Mozilla/6.9",0

.data?
  hInstance dd ?
  buffer db 256 dup(?)
  html db 256 dup(?)

;
getWAN.rc

; This Resource Script was generated by WinAsm Studio.
#define GET 1001
#define EXIT 1002
#define RESULT 1003

101 DIALOGEX 0,0,109,35
CAPTION "getWAN"
FONT 8,"Tahoma"
STYLE 0x80c80880
EXSTYLE 0x00000000
BEGIN
CONTROL "Get IP",GET,"Button",0x10000001,3,18,50,14,0x00000000
CONTROL "Exit",EXIT,"Button",0x10000000,56,18,50,14,0x00000000
CONTROL "",RESULT,"Edit",0x10000080,3,3,101,12,0x00000200
END

User Interface

Assembled Code

```
00401000  >/$  6A 00  PUSH 0                      ; /pModule = NULL
00401002  |.  E8 FB000000  CALL <JMP &kernel32.GetModuleHandleA>; \GetModuleHandleA
00401007  |.  A3 40304000  MOV DWORD PTR DS:[403040],EAX
0040100C  |.  6A 00  PUSH 0                      ; /Param = NULL
0040100E  |.  68 28104000  PUSH getWAN.00401028 ; /DlgProc = getWAN.00401028
00401013  |.  6A 00  PUSH 0                      ; /hOwner = NULL
00401015  |.  6A 65  PUSH 65                     ; /pTemplate = 65
00401017  |.  FF35 40304000  PUSH DWORD PTR DS:[403040] ; /hInst = NULL
0040101D  |.  E8 C8000000  CALL <JMP &user32.DialogBoxParamA> ; /DialogBoxParamA
00401022  |.  50  PUSH EAX                      ; /ExitCode
00401023  \.  E8 D4000000  CALL <JMP &kernel32.ExitProcess> ; \ExitProcess
00401028  /.  55  PUSH EBP
00401029  |.  8BEC  MOV EBP,ESP
0040102B  |.  817D 0C 11010> CMP DWORD PTR SS:[EBP+C],111
00401032  |.  75 3C  JNZ SHORT getWAN.00401070
00401034  |.  817D 10 E9030> CMP DWORD PTR SS:[EBP+10],3E9
0040103B  |.  75 1E  JNZ SHORT getWAN.0040105B
0040103D  |.  68 00304000  PUSH getWAN.00403000 ; /Arg1 = 00403000 ASCII
          "http://www.whatismyip.com/automation/n09230945.asp"
```

```
00401000  >/$  6A 00  PUSH 0                      ; /pModule = NULL
00401002  |.  E8 FB000000  CALL <JMP &kernel32.GetModuleHandleA>; \GetModuleHandleA
00401007  |.  A3 40304000  MOV DWORD PTR DS:[403040],EAX
0040100C  |.  6A 00  PUSH 0                      ; /Param = NULL
0040100E  |.  68 28104000  PUSH getWAN.00401028 ; /DlgProc = getWAN.00401028
00401013  |.  6A 00  PUSH 0                      ; /hOwner = NULL
00401015  |.  6A 65  PUSH 65                     ; /pTemplate = 65
00401017  |.  FF35 40304000  PUSH DWORD PTR DS:[403040] ; /hInst = NULL
0040101D  |.  E8 C8000000  CALL <JMP &user32.DialogBoxParamA> ; /DialogBoxParamA
00401022  |.  50  PUSH EAX                      ; /ExitCode
00401023  \.  E8 D4000000  CALL <JMP &kernel32.ExitProcess> ; \ExitProcess
00401028  /.  55  PUSH EBP
00401029  |.  8BEC  MOV EBP,ESP
0040102B  |.  817D 0C 11010> CMP DWORD PTR SS:[EBP+C],111
00401032  |.  75 3C  JNZ SHORT getWAN.00401070
00401034  |.  817D 10 E9030> CMP DWORD PTR SS:[EBP+10],3E9
0040103B  |.  75 1E  JNZ SHORT getWAN.0040105B
0040103D  |.  68 00304000  PUSH getWAN.00403000 ; /Arg1 = 00403000 ASCII
          "http://www.whatismyip.com/automation/n09230945.asp"
```
00401042 | E8 3F000000 CALL getWAN.00401086 ; \getWAN.00401086
00401047 | 68 44314000 PUSH getWAN.00403144 ; /Text = ""
0040104C | 68 EB030000 PUSH 3EB ; |ControlID = 3EB (1003.)
00401051 | FF75 08 PUSH DWORD PTR SS:[EBP+8] ; hWnd
00401054 | E8 9D000000 CALL <JMP.&user32.SetDlgItemTextA> ; SetDlgItemTextA
00401059 | EB 25 JMP SHORT getWAN.00401080
0040105B | 817D 10 EA030> CMP DWORD PTR SS:[EBP+10],3EA
00401062 | 75 1C JNZ SHORT getWAN.00401080
00401064 | 6A 00 PUSH 0 ; /Result = 0
00401066 | FF75 08 PUSH DWORD PTR SS:[EBP+8] ; hWnd
00401069 | E8 82000000 CALL <JMP.&user32.EndDialog> ; EndDialog
00401074 | FF75 FC PUSH DWORD PTR SS:[EBP-4]
00401076 | 6A 00 PUSH 0 ; /Result = 0
00401078 | FF75 08 PUSH DWORD PTR SS:[EBP+8] ; hWnd
0040107B | E8 70000000 CALL <JMP.&user32.EndDialog> ; EndDialog
00401080 | 33C0 XOR EAX,EAX
00401082 | C9 LEAVE
00401083 | C2 1000 RETN 10
00401086 | 55 PUSH EBP
00401087 | 8BEC MOV EBP,ESP
00401089 | 83C4 F0 ADD ESP,-10
0040108C | 6A 00 PUSH 0
0040108E | 6A 00 PUSH 0
00401090 | 6A 00 PUSH 0
00401092 | 6A 00 PUSH 0
00401094 | 6B 33304000 PUSH getWAN.00403033 ; ASCII "Mozilla/6.9"
00401099 | E8 70000000 CALL <JMP.&wininet.InternetOpenA>
0040109E | 8945 FC MOV DWORD PTR SS:[EBP-4],EAX
004010A1 | 6A 00 PUSH 0
004010A3 | 6A 00 PUSH 0
004010A5 | 6A 00 PUSH 0
004010A7 | 6A 00 PUSH 0
004010A9 | FF75 08 PUSH DWORD PTR SS:[EBP+8]
004010AC | FF75 FC PUSH DWORD PTR SS:[EBP-4]
004010AF | E8 60000000 CALL <JMP.&wininet.InternetOpenUrlA>
004010B4 | 8945 F8 MOV DWORD PTR SS:[EBP-8],EAX
004010B7 | 8D45 F0 LEA EAX,DWORD PTR SS:[EBP-10]
004010BA | 50 PUSH EAX
004010BB | 68 00010000 PUSH 100
004010C0 | 68 44314000 PUSH getWAN.00403144
004010C5 | FF75 F8 PUSH DWORD PTR SS:[EBP-8]
004010C8 | E8 4D000000 CALL <JMP.&wininet.InternetReadFile>
004010CD | FF75 F8 PUSH DWORD PTR SS:[EBP-8]
004010D0 | E8 33000000 CALL <JMP.&wininet.InternetCloseHandle>
004010D5 | 0BC0 OR EAX,EAX
004010D7 | 75 0C JNZ SHORT getWAN.004010E5
004010D9 | FF75 FC PUSH DWORD PTR SS:[EBP-4]
004010DC | E8 27000000 CALL <JMP.&wininet.InternetCloseHandle>
004010E1 | C9 LEAVE
004010E2 | C2 0400 RETN 4
General Obfuscation approach

The following assembly code is an example of how you could obfuscate the code.
```
MOV DWORD PTR SS:[ESP-4],3EB
SUB ESP,4
PUSH DWORD PTR SS:[EBP+8]
CALL getWAN_o.004010B2
ADD DWORD PTR SS:[ESP],6
JMP SHORT <JMP.&user32.SetDlgItemTextA>
ADD DWORD PTR SS:[ESP],7
JMP SHORT getWAN_o.00401126
MOV EAX,3EA
CMP DWORD PTR SS:[EBP+10],EAX
JNZ SHORT getWAN_o.00401126
PUSH 0
PUSH DWORD PTR SS:[EBP+8]
CALL getWAN_o.
ADD DWORD PTR SS:[ESP],7
JMP SHORT <JMP.&user32.EndDialog>
JMP SHORT getWAN_o.00401126
ADD EAX,-3DA
CMP DWORD PTR SS:[EBP+C],10
JNZ SHORT getWAN_o.00401126
PUSH 0
EAX
JMP SHORT getWAN_o.00401121
INT3
INT3
INT3
JMP DWORD PTR DS:[<&user32.DialogBoxPara>; user32.DialogBoxParamA]
JMP DWORD PTR DS:[<&user32.EndDialog>; user32.EndDialog]
JMP DWORD PTR DS:[<&user32.SetDlgItemText>; user32.SetDlgItemTextA]
JMP DWORD PTR DS:[<&kernel32.ExitProcess>; kernel32.ExitProcess]
JMP DWORD PTR DS:[<&wininet.InternetCloseHandle>; wininet.InternetCloseHandle]
JMP DWORD PTR DS:[<&wininet.InternetOpen>; wininet.InternetOpenA]
JMP DWORD PTR DS:[<&wininet.InternetOpenUrl>; wininet.InternetOpenUrlA]
JMP DWORD PTR SS:[EBP+8]
JMP DWORD PTR SS:[EBP+10],EAX
JMP DWORD PTR SS:[EBP+C],10
JMP DWORD PTR SS:[EBP+10],EAX
JMP DWORD PTR SS:[EBP+C],10
INT3
DB 00
PUSH 0
NOP
POP EAX
LEAVE
RETN 10
PUSH EBP
MOV EBP,ESP
ADD ESP,-10
PUSH 0
PUSH 0
PUSH 0
PUSH 0
ADD ESP,83C4F0
PUSH EBP,ESP
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH getWAN_o.00403033 ; ASCII "Mozilla/6.9"
```
E8 BFFFFFFFF CALL <JMP.&wininet.InternetOpenA>
8945 FC MOV DWORD PTR SS:[EBP-4],EAX
6A 00 PUSH 0
6A 00 PUSH 0
6A 00 PUSH 0
6A 00 PUSH 0
FF75 08 PUSH DWORD PTR SS:[EBP+8]
FF75 FC PUSH DWORD PTR SS:[EBP-4]
E8 AFFFFFFFF CALL <JMP.&wininet.InternetOpenUrlA>
8945 F8 MOV DWORD PTR SS:[EBP-8],EAX
8D45 F0 LEA EAX,DWORD PTR SS:[EBP-10]
50 PUSH EAX
68 00010000 PUSH 100
68 44314000 PUSH getWAN_o.00403144
FF75 F8 PUSH DWORD PTR SS:[EBP-8]
E8 9CFFFFFFF CALL <JMP.&wininet.InternetReadFile>
FF75 F8 PUSH DWORD PTR SS:[EBP-8]
E8 82FFFFFF CALL <JMP.&wininet.InternetCloseHandle>
0BC0 OR EAX,EAX
75 0C JNZ SHORT getWAN_o.00401196
FF75 FC PUSH DWORD PTR SS:[EBP-4]
E8 76FFFFFF CALL <JMP.&wininet.InternetCloseHandle>
C9 LEAVE
C2 0400 RETN 4
C9 LEAVE
C2 0400 RETN 4
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equal Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH &lt;value/register&gt;</td>
<td>MOV DWORD PTR SS:[ESP-4],&lt;value/register&gt;</td>
</tr>
<tr>
<td></td>
<td>SUB ESP,4</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>MOV &lt;register/[memory]&gt;,&lt;value/register&gt;</td>
</tr>
<tr>
<td></td>
<td>PUSH &lt;register/[memory]&gt;</td>
</tr>
<tr>
<td>POP &lt;register/[memory]&gt;</td>
<td>MOV &lt;register&gt;, DWORD [ESP]</td>
</tr>
<tr>
<td></td>
<td>ADD ESP,4</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>POP &lt;[memory]&gt;</td>
</tr>
<tr>
<td></td>
<td>MOV &lt;register&gt;,&lt;[memory]&gt;</td>
</tr>
<tr>
<td>CALL &lt;address&gt;</td>
<td>PUSH EIP + bytes to next instruction</td>
</tr>
<tr>
<td></td>
<td>JMP &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>MOV &lt;[memory]/register&gt;,&lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>CALL &lt;[memory]/register&gt;</td>
</tr>
<tr>
<td>JMP &lt;address&gt;</td>
<td>PUSH &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>RETN</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>CALL &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>At address:</td>
</tr>
<tr>
<td></td>
<td>POP &lt;register&gt; or ADD ESP,4</td>
</tr>
<tr>
<td>RETN</td>
<td>POP EAX</td>
</tr>
<tr>
<td></td>
<td>JMP EAX</td>
</tr>
<tr>
<td>SUB &lt;register/memory&gt;,&lt;value&gt;</td>
<td>ADD &lt;register&gt;, -&lt;value&gt;</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>LEA &lt;register&gt;, DWORD [&lt;register&gt;-&lt;value&gt;]</td>
</tr>
<tr>
<td>ADD &lt;register/memory&gt;,&lt;value&gt;</td>
<td>SUB &lt;register&gt;,-&lt;value&gt;</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>LEA &lt;register&gt;, DWORD [&lt;register&gt;+&lt;value&gt;]</td>
</tr>
<tr>
<td>PUSH &lt;xx&gt; POP &lt;register&gt;</td>
<td>MOV &lt;register&gt;,&lt;xx&gt;</td>
</tr>
<tr>
<td>ADD &lt;register/[memory]&gt;,1</td>
<td>NOT &lt;register/[memory]&gt;</td>
</tr>
<tr>
<td></td>
<td>NEG &lt;register/[memory]&gt; (affects C and A flags)</td>
</tr>
<tr>
<td></td>
<td>LEA &lt;register&gt;,DWORD[&lt;register&gt;+1]</td>
</tr>
<tr>
<td>SUB &lt;register/[memory]&gt;,1</td>
<td>NEG &lt;register/[memory]&gt;</td>
</tr>
<tr>
<td></td>
<td>NOT &lt;register/[memory]&gt; (affects C and A flags)</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>LEA &lt;register&gt;,DWORD[&lt;register&gt;-1]</td>
</tr>
<tr>
<td>***MOV &lt;register&gt;,0</td>
<td>AND &lt;register&gt;,0</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>SHL &lt;register&gt;,30</td>
</tr>
<tr>
<td></td>
<td>SHL &lt;register&gt;,30</td>
</tr>
</tbody>
</table>
(note that this affects the Z,S and O flags)

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR PUSH 0</td>
</tr>
<tr>
<td>OR POP &lt;register&gt;</td>
</tr>
<tr>
<td>OR XOR &lt;register&gt;,&lt;register&gt;</td>
</tr>
<tr>
<td>OR SUB &lt;register&gt;,&lt;register&gt;</td>
</tr>
<tr>
<td>OR LEA &lt;register&gt;, DWORD [0]</td>
</tr>
</tbody>
</table>

| MOV <16bit register>,0 |
| AND <register>, FFFF0000 |
| AND <16bit register>, 0000 |

| NOP |
| AND <register/[memory]>, FFFFFFFF (-1) |
| OR SHL <register>, 0 (logical shift left) |
| OR SHR <register>, 0 (logical shift right) |
| OR MOV <register>, <register> |
| OR XOR <register/[memory]>, 0 |
| OR ADD <register/[memory]>, 0 |
| OR SUB <register/[memory]>, 0 |
| OR <register/[memory]>, 0 |
| OR PUSH <register/[memory]> |
| POP <register/[memory]> |
| OR XCHG <register> |
| OR FNOP |
| OR LEA <register>, [<register>] |
| OR FNOP |

| NOT <register/[memory]> |
| XOR <register/[memory]>, -1 |
| OR NEG <register/[memory]> |
| SUB <register/[memory]>, 1 |

<p>| MOV &lt;register a&gt;, &lt;register b&gt; |
| PUSH &lt;register a&gt; |
| POP &lt;register b&gt; |
| OR |</p>
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV DWORD [[memory] offset],&lt;register a&gt; MOV &lt;register b&gt;, DWORD [[memory] offset] OR LEA &lt;register a&gt;, DWORD [register b] OR MOV [memory]&lt;register a&gt; MOV &lt;register b&gt;,[memory]</td>
<td></td>
</tr>
<tr>
<td>CMP &lt;register&gt;,0</td>
<td>OR &lt;register&gt;,&lt;register&gt; OR AND &lt;register&gt;,&lt;register&gt; OR TEST &lt;register&gt;,&lt;register&gt;</td>
</tr>
<tr>
<td><strong>MOV &lt;register&gt;,&lt;value&gt;</strong></td>
<td>LEA &lt;register&gt;,,[&lt;value&gt;] OR PUSH &lt;value&gt; POP &lt;register&gt;</td>
</tr>
<tr>
<td>MOV [[memory]],[value/register]</td>
<td>PUSH &lt;value/register&gt; POP [[memory]]</td>
</tr>
<tr>
<td>MOV [[memory] a],[[memory] b] (NO such instruction)</td>
<td>PUSH [[memory] b&gt; POP [[memory] a&gt;</td>
</tr>
<tr>
<td>NEG &lt;register[[memory]]&gt;</td>
<td>NOT &lt;register[[memory]]&gt; ADD &lt;register[[memory]]&gt;,1</td>
</tr>
</tbody>
</table>

Greetings
I would like to say thank you to the people that helped me writing this paper. Special thanks fly out to Maria-Lena Demetriou for correcting and pointing out some grammatical errors :)