

تأليف: مهند سليمان

الكتاب لم يكتمل بعد

ملاحظة

بعض الاكود الموجودة في الكتاب مقتبسة من برامج مفتوحة المصدر

هل فكرت يوم كيف تقوم ببناء برنامج تصحيح يضاهي ollydbg إذا كنت من الذين يردون صنع برامجهم بأنفسهم فهذا الكتاب سيأخذ بيدك خطوة بخطوة .

لماذا يمتاز olly عن برامج التقيح رغم ان المبداء واحد؟

السبب هو قوة التحليل التي يمتاز بها فيجعل فهم البرنامج في غاية الوضوح وايضا دعم واسع لتعريفات دوال API ومعرفه البارمترات الممررة لها . إذا ليس من المستحيل والصعب جدا بناء برنامج يتفوق عليه خاصة إذا كان مقتوح المصدر تخيل عدد الناس الذين سوف يصفون افكارهم مقترحاتهم زيادتهم بالتاكيد في النهاية سينتج برنامج يتفوق على olly بكل بساطة !

قبل البداية.

المتطلبات

نظام تشغيل XP sp1 or sp2

لغة البرمجة .

الفيجوال بيسك (استخدمت البيسك لان الجميع يستطيع التعامل معها والحصول على أكبر فائدة لجميع المبرمجين)

فهم ولو بسيط لكيفية التعامل مع API

البداية .

لماذا أقوم بكتابة برنامج تصحيح ويوجد لدي ollydbg؟

هل ollydbg يلبي كل احتياجاتك هل خطرت لك فكرة ولم تستطيع عملها لان ollydbg لتتوفر به هذه الميزة، هل فكرت ماذا ستستفيد من كتابة برنامج تصحيح خاص بك ! كل هذه الامور أو أكثر تستطيع الاجابة عنها بنفسك بعد الانتهاء من عمل برنامجك الخاص .

لتصحيح أي برنامج يجب ادخله لحلقة التصحيح

يجب أن يكون البرنامج المراد تصحيحه فرع من برنامجنا الذي يعتبر أب له . وهناك طريقتين

الطريقة الأولى.

شغل الفيجوال بيسك و أضف له commonDialog وسمه op

```
op.ShowOpen
```

```
PID = Shell(op.FileName)
```

الدالة **shell** لتشغيل البرنامج والحصول على رقم العملية وجعله مرتبطا ببرنامجنا الرئيسي

ميزة هذه الطريقة انك تنقح البرنامج وهو في الذاكرة تنفع للبرامج المشفرة .

بعد ذلك نعرف العملية على أنها قابلة للتصحيح كيف

بواسطة الدالة (PID) **DebugActiveProcess** وهي تطلب بارمتر واحد وهو رقم العملية

تعريف الدالة في الفيچوال بيسك (لتنسى وضعها في الموديول)

```
Declare Function DebugActiveProcess Lib "kernel32" (ByVal dwProcessId As Long) As Long
```

بعد ذلك ندخل البرنامج لحققة التصحيح

هذه هي الحلقة

```
DO  
loop
```

الآن ستقوم بتعبئة الحلقة

```
DO  
A = WaitForDebugEvent(debugp, 3)  
loop
```

هذه الدالة من اسمها تقوم بانتظار الأحداث سياسية ، اقتصادية ، اجتماعية لكن للأسف تقوم بانتظار أحداث التصحيح

ونستقبلها في التركيب debugp

أحداث التصحيح

سيداتي وسادتي جانا خبر عاجل بان البرنامج X قد قام

باستثناء (اسرائيل من التفيش عن الأسلحة النووية)

إنشاء عملية (إرهابية) في الذاكرة

أنشاء فرع من عملية (النفط مقابل الغذاء) في الذاكرة

وهكذا إلي آخر الأحداث

تعريف الدالة

```
Declare Function WaitForDebugEvent Lib "kernel32.dll" (ByRef lpDebugEvent As DEBUG_EVENT, ByVal dwMilliseconds As Long) As Boolean
```

البار متر الأول يستقبل الأحداث وهو عبارة عن تركيب

البار متر الثاني مدة الانتظار

وتعيد قيمة 1 إذا وجود حدث

Public Type DEBUG_EVENT

dwDebugEventCode As Long ' رقم الحدث

dwProcessId As Long ' مقبض العملية للبرنامج المنفتح

dwThreadId As Long ' مقبض فرع العملية

DATA(20) As Long ' مساحة زائد لإحداث التنقيح الاخرى

End Type

التركيب DEBUG

نكمل باقي الحلقة

Public Const EXCEPTION_DEBUG_EVENT = 1

Public Const CREATE_THREAD_DEBUG_EVENT = 2

Public Const CREATE_PROCESS_DEBUG_EVENT = 3

Public Const EXIT_THREAD_DEBUG_EVENT = 4

Public Const EXIT_PROCESS_DEBUG_EVENT = 5

Public Const LOAD_DLL_DEBUG_EVENT = 6

Public Const UNLOAD_DLL_DEBUG_EVENT = 7

Public Const OUTPUT_DEBUG_STRING_EVENT = 8

Dim debugp as DEBUG_EVENT

DO

A = WaitForDebugEvent(debugp, 3)

If A <> 0 Then

Select Case debugp.dwDebugEventCode

Case CREATE_PROCESS_DEBUG_EVENT ' مهم للحصول على مقبض العملية

Case EXCEPTION_DEBUG_EVENT ' الاستثناءات أهم شيء نقطة الإيقاف

Case CREATE_THREAD_DEBUG_EVENT ' الإعلام إنشاء فرع آخر

Case EXIT_THREAD_DEBUG_EVENT ' الإعلام بإنهاء فرع من العملية

Case EXIT_PROCESS_DEBUG_EVENT ' الإعلام بإنهاء العملية

Exit DO

Case LOAD_DLL_DEBUG_EVENT ' تحميل مكتبة

Case UNLOAD_DLL_DEBUG_EVENT ' إنهاء تحميل مكتبة

Case OUTPUT_DEBUG_STRING_EVENT ' إظهار نص أثناء التنقيح

DoEvents

ContinueDebugEvent(debugp.dwProcessId, debugp.dwThreadId, DBG_CONTINUE)

Loop

End if

بعد الانتهاء من شرح
الأحداث سنتحدث عنها
أو انقر عليها

هكذا كونا الحلقة الرئيسة لبرنامج التصحيح بعد ذلك نقوم بتشريح كل حدث

الحدث الأول

CREATE_PROCESS_DEBUG_EVENT

للعلم هنا توجه مبرمجي الفيچوال بيسك مشكلة وهي إن التركيب union التابع للتركيب DEBUG_EVENT غير مدعوم مباشرة لاحظ المساحة الزائدة

مساحة زائدة لإحداث التتبع الأخرى ' DATA(20) As Long

لذلك يتوجب علينا إنشاء تركيب جديد ونسميه CREATE_PROCESS_DEBUG_INFO

Public Type CREATE_PROCESS_DEBUG_INFO

hFile As Long

hProcess As Long

hThread As Long

lpBaseOfImage As Long

dwDebugInfoFileOffset As Long

nDebugInfoSize As Long

lpThreadLocalBase As Long

lpStartAddress As Long

lpImageName As Long

fUnicode As Integer

End Type

ونعلن عن متغير جديد

Public CREATEPROCESSINFO As CREATE_PROCESS_DEBUG_INFO

من الأفضل إنشاء موديول جديد ووضع المتغيرات به

أهم شيء نستفيد من هذا التركيب مقبض العملية وفرعها (Thread) والحجم الوهمي للصورة (BaseOfImage)

المشكلة التي لدينا الآن هي كيف نربط هذا التركيب بالتركيب DEBUG_EVENT

الحل يكمن عن طريق دالة API وهي CopyMemory

Case CREATE_PROCESS_DEBUG_EVENT

CopyMemory CREATEPROCESSINFO, debugp.DATA(0), Len(CREATEPROCESSINFO)

Hproc = CREATEPROCESSINFO.hProcess

ImgB= CREATEPROCESSINFO. lpBaseOfImage

تعريف الدالة

```
Public Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemoryReadProcessMemory" (Destination As Any, Source As Any, ByVal Length As Long)
```

بعد أن حصلنا على مقبض العملية نقوم بالقراءة من الذاكرة بواسطة دالة API **"ReadProcessMemory"**

تعريف الدالة

```
Declare Function ReadProcessMemory Lib "kernel32" (ByVal hProcess As Long, lpBaseAddress As Any, lpBuffer As Any, ByVal nSize As Long, lpNumberOfBytesWritten As Long) As Long
```

البارمتر الأول : مقبض العملية

البارمتر الثاني : العنوان الذي تريد أن يبدأ منه القراءة مثلا العنوان &H401000 مع ملاحظة يجب أن يكون العنوان بالترميز العشري

وهذا كود للتحويل من HEX to DEC

```

Public Function Hex2Dec(ByVal paraHexValue As String) As String

Dim LC As Long

Dim curHexValue As String

Dim curChar As String

Dim curValue As String

((curHexValue = UCase(Trim(paraHexValue

If Left(curHexValue, 2) = "&H" Then curHexValue = Mid(curHexValue, 3

For LC = 1 To Len(curHexValue

curChar = Mid(curHexValue, LC, 1

If InStr(1, "ABCDEF", curChar) <> 0 Then curChar = Asc(curChar) - 55

(((curValue = Val(curValue) + (Val(curChar) * (16 ^ (Len(curHexValue) - LC

Next LC

Hex2Dec = Val(curValue

End Function

```

البارمتر الثالث: متغير لتخزين البايتات المقروءة

تعريف المتغير

Dim buffer() as byte

البارمتر الرابع: عدد البايتات التي تريد قراءتها مثلا 100

Redim buffer(100) as byte

البارمتر الخامس: متغير يعد عدد البايتات المقروءة

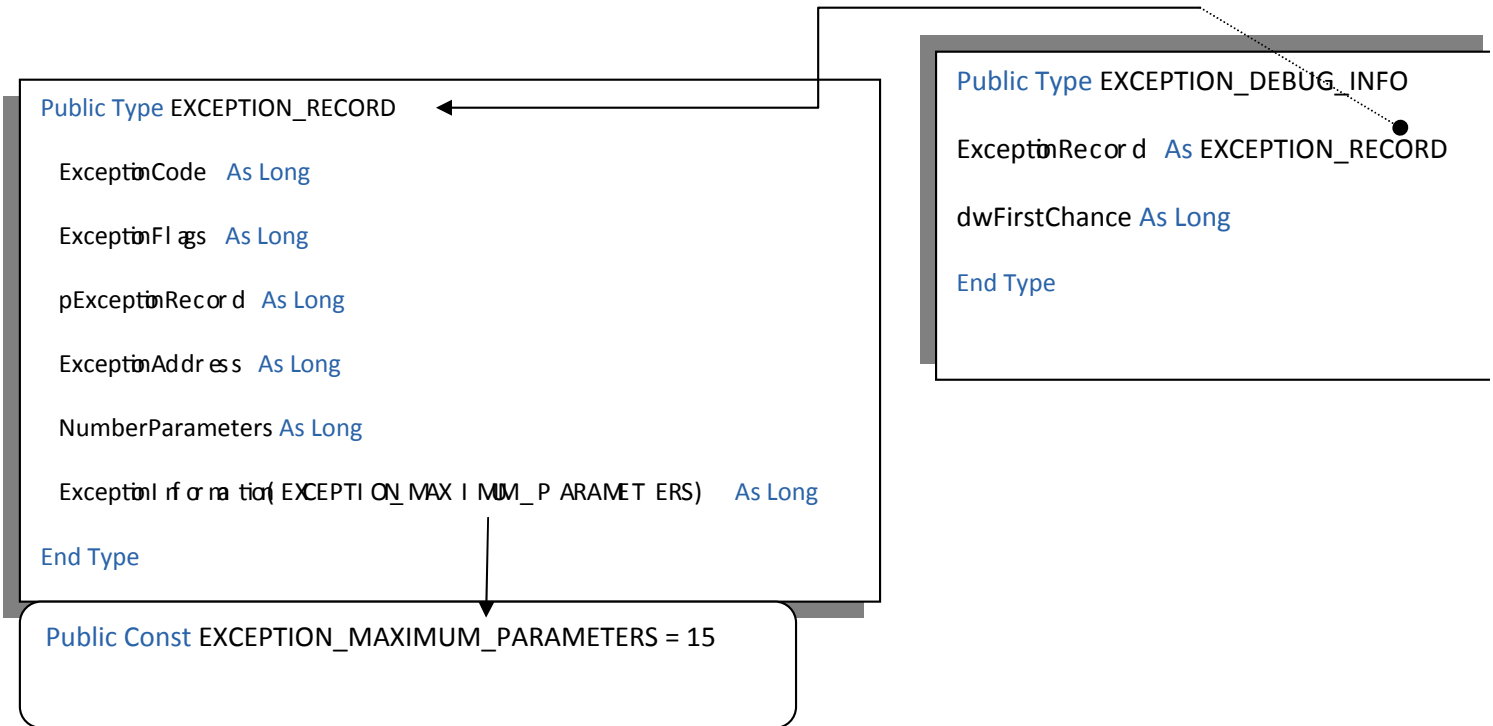
بعد أن قمنا بقراءة الذاكرة نقوم بعمل [Dissemble](#) لها انقر على الرابط لتعرف كيفية عمل [Dissemble](#)

بعد أن انتهينا من القراءة بقى الكتابة لكن لن نتكلم عنها الآن سنتحدث عنها عندما نريد وضع نقطة إيقاف
هكذا انتهينا من الحدث الأول

الحدث الثاني :

EXCEPTION_DEBUG_EVENT

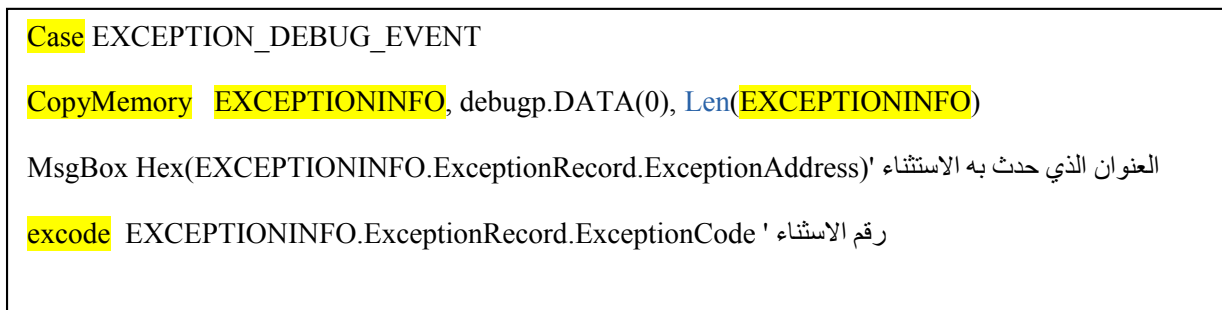
بنفس الفكرة الأولى ننشئ تركيب آخر



بعد أن وضعنا التعاريف السابقة في الموديل نعلن عن متغير جديد

```
Public EXCEPTIONINFO As EXCEPTION_DEBUG_INFO
```

ونضعه في موديل التعاريف



لاحظ الكود السابق وجود الاجراء **excode** والذي يتطلب بارمتر واحد وهو رقم الاستثناء

```
Public Const EXCEPTION_DEBUG_EVENT = 1

Public Const EXCEPTION_DATATYPE_MISALIGNMENT =
    &H80000002

Public Const EXCEPTION_SINGLE_STEP = &H80000004

Public Const EXCEPTION_ACCESS_VIOLATION =
    &HC00000005

Public Const EXCEPTION_BREAKPOINT = &H80000003

Public Const EXCEPTION_ARRAY_BOUNDS_EXCEEDED =
    &HC0000008C

Public Const EXCEPTION_FLT_DIVIDE_BY_ZERO =
    &HC0000008E

Public Const EXCEPTION_FLT_INVALID_OPERATION =
    &HC00000090

Public Const EXCEPTION_FLT_OVERFLOW = &HC00000091

Public Const EXCEPTION_INT_DIVIDE_BY_ZERO =
    &HC00000094

Public Const EXCEPTION_INT_OVERFLOW = &HC00000095

Public Const EXCEPTION_ILLEGAL_INSTRUCTION =
    &HC0000001D

Public Const EXCEPTION_PRIV_INSTRUCTION =
    &HC00000096
```

Sub excode(code As Long)

Select Case code

Case EXCEPTION_DEBUG_EVENT

Case EXCEPTION_DATATYPE_MISALIGNMENT

Case EXCEPTION_SINGLE_STEP ' مهم عند ما نتتبع الكود خطوة بخطوة '

Case EXCEPTION_ACCESS_VIOLATION

Case EXCEPTION_BREAKPOINT ' نقطة إيقاف '

Case EXCEPTION_ARRAY_BOUNDS_EXCEEDED

Case EXCEPTION_FLT_DIVIDE_BY_ZERO

Case EXCEPTION_FLT_INVALID_OPERATION

Case EXCEPTION_FLT_OVERFLOW

Case EXCEPTION_INT_DIVIDE_BY_ZERO

Case EXCEPTION_INT_OVERFLOW

Case EXCEPTION_ILLEGAL_INSTRUCTION

Case EXCEPTION_PRIV_INSTRUCTION

End Select

End Sub

أعتقد أن جميع الاستثناء واضحة لكننا سنركز على اثنين فقط

الحدث الثالث :

LOAD_DLL_DEBUG_EVENT

نفس السابق ننشئ تركيب

```
Public Type LOAD_DLL_DEBUG_INFO
    hFile As Long
    lpBaseOfDll As Long ' ا هم شيء عنوان المكتبة
    dwDebugInfoFileOffset As Long
    nDebugInfoSize As Long
    lpImageName As Long
    fUnicode As Integer
End Type
```

بعد ذلك نعلن عن متغير

```
Public LOADDLL As LOAD_DLL_DEBUG_INFO
```

ونكمل الحلقة

```
Case LOAD_DLL_DEBUG_EVENT
    CopyMemory LOADDLL, debugp.DATA(0), Len(LOADDLL)
    named = GetModuleNameFromHandle(LOADDLL.lpBaseOfDll)
```

لاحظ أن أضفنا دالة GetModuleNameFromHandle للحصول على اسم المكتبة عن طريق عنوانها في الذاكرة

```
Public Function GetModuleNameFromHandle(ByVal hModule As Long) As String
    Dim lLen As Long
    GetModuleNameFromHandle = Space(260)
    lLen = GetModuleFileNameEx(CREATEPROCESSINFO.hProcess, hModule, GetModuleNameFromHandle, 260)
    GetModuleNameFromHandle = Left(GetModuleNameFromHandle, lLen) ' تخلص من المسافات الزائدة
End Function
```

الدالة `GetModuleFileNameExA` هي دالة API تنبع المكتبة `psapi.dll`

تعريف الدالة :

```
Public Declare Function GetModuleFileNameEx Lib "psapi.dll" (ByVal hProcess As Long, ByVal hModule As Long, ByVal ModuleName As String, ByVal nSize As Long) As Long
```

البارمتر الأول : مقبض العملية وقد حصلنا عليه عن طريق `CREATEPROCESSINFO.hProcess`

البارمتر الثاني : عنوان المكتبة في الذاكرة وتم الحصول عليه بواسطة `LOADDLL.lpBaseOfDll`

البارمتر الثالث: متغير للحصول على اسم ومسار المكتبة

البارمتر الرابع: يمرر طول المتغير في البارمتر الثالث

الدالة تعيد طول الاسم والمسار للمكتبة

الحدث الرابع :

`unLOAD_DLL_DEBUG_EVENT`

```
Public Type UNLOAD_DLL_DEBUG_INFO
```

```
lpBaseOfDll As Long ' عنوان المكتبة
```

```
End Type
```

```
Public UNLOADDLL As UNLOAD_DLL_DEBUG_INFO
```

باقي الحلقة

```
Case UNLOAD_DLL_DEBUG_EVENT
```

```
CopyMemory UNLOADDLL, debugp.DATA(0), Len(UNLOADDLL)
```

```
named = GetModuleNameFromHandle(UNLOADDLL.lpBaseOfDll)
```

الحدث الخامس :

EXIT_PROCESS_DEBUG_EVENT

واضح من اسمه انه ينفذ عندا انتهاء العملية التي هي قيد التصحيح

CASE EXIT_PROCESS_DEBUG_EVENT

Msgbox "process end"

Exit DO

End select

LOOP

هكذا تكون انتهت دورة التصحيح

الكود الذي كتبنا في الصفحات السابقة صالح لدورة واحد ولجعل البرنامج يدخل في دور لا نهائية علينا إضافة الدالة

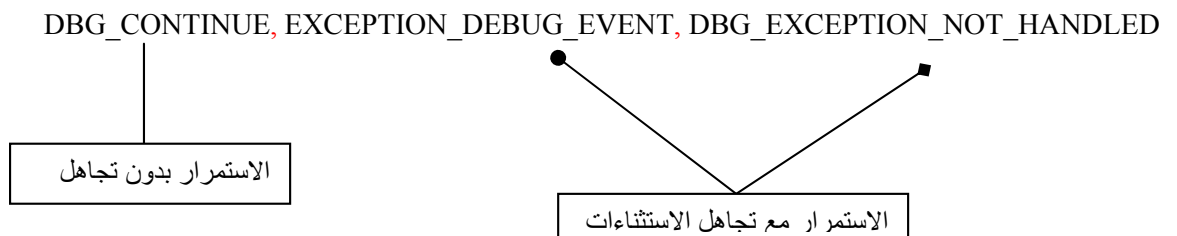
ContinueDebugEvent التي لها 3 بارمتر

```
Declare Function ContinueDebugEvent Lib "kernel32" (ByVal dwProcessId As Long, ByVal dwThreadId As Long,
ByVal dwContinueStatus As Long) As Long
```

البارمتر الاول مقبض العملية

البارمتر الثاني مقبض فرع العملية

البارمتر الثالث يحدد نوع الاستمرار



ContinueDebugEvent (debugp.dwProcessId, debugp.dwThreadId, DBG_CONTINUE)

بعد الانتهاء من المحور الأساسي لبرنامج المصحح الآن سندخل في التفاصيل التي تميز برامج الديبغر عن بعضها

فك التجميع

أدرجت مع الدرس الاداة Dissemble وهي تقوم بتحويل مصفوفة البايتات التي ممرت إليها إلى تعليمات إسمبلي وقد قمت ببنائها مستعمل أكواد من برنامج مفتوح المصدر أي يحق لك استعمالها في برامج تجارية

كيفية استعمال الاداة.

ضع على الفورم أداة listbox وسميها Ldis

ضع الاداة Dissemble على الفورم

1- تحديد العنوان الذي يبدأ به فك التجميع مثلا 400000

```
dissemble.BaseAddress = &H400000
```

2- تعبئة مصفوفة البايتات التي يمكن الحصول عليها عن طريق الدالة

Readfile or [ReadProcessMemory](#)

```
Dim dis(100) As Byte
```

```
Dim cu as long
```

```
Dim cmd as string
```

```
ReadProcessMemory(hprocess, ByVal adr, dis(1), 100, n)
```

```
DO
```

```
cmd = disassemble.DisAssemble(dis, Cu, Forward, 1, 0, IsError)
```

```
If IsError = 1 Then
```

```
Cu = Cu + 2
```

```
Else
```

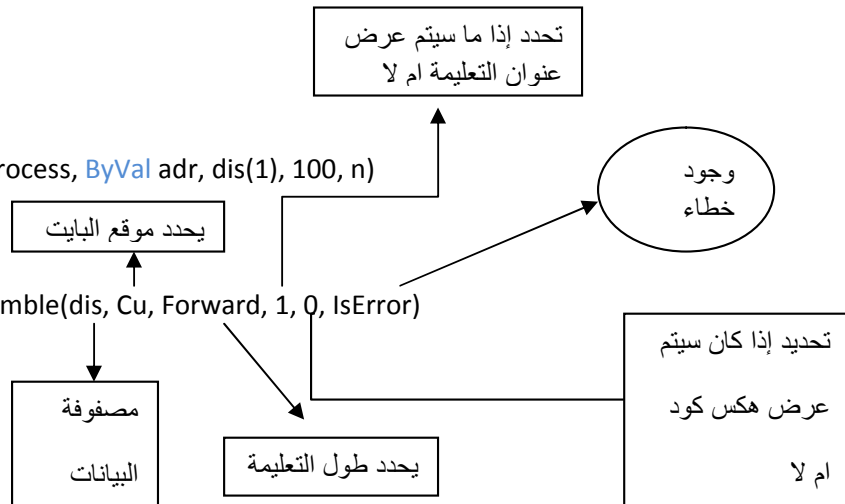
```
Cu = Cu + Forward
```

```
End If
```

```
Ldis.AddItem cmd
```

```
If cu > 100 then exit do
```

```
LOOP
```



بعد ذلك يتم تحميل الالة ليست بوكس بتعليمات الاسمبلي

يوجد مع المرفقات مثال يوضح السابق

التعامل مع المعالج

1-القراءة

بعد الانتهاء من واجهة البرنامج وهي تعليمات الاسمبلي بقي الحصول على محتويات الرجستري للمعالج كيف؟

عن طريق الدالة **GetThreadContext**

```
Public Declare Function GetThreadContext Lib "kernel32" (ByVal hThread As Long, lpContext As CONTEXT) As Long
```

نلاحظ من تعريف الدالة انها تتطلب 2 بارمتر

البارمتر الاول : مقبض فرع العملية ونعرف كيفية الحصول عليه **CREATEPROCESSINFO.hThread**

البارمتر الثاني : التركيب context وهو أهم شئ يعتبر كأنه المعالج الموجود بجهازك

تعريف التركيب context

Public Type CONTEXT

ContextFlags As Long ' أعلام المعالج

مسجلات التصحيح

Dr0 As Long

Dr1 As Long

Dr2 As Long

Dr3 As Long

Dr6 As Long

Dr7 As Long

FloatSave As FLOATING_SAVE_AREA ' لعرض الارقام الحقيقية

SegGs As Long

SegFs As Long

SegEs As Long

SegDs As Long

Edi As Long

Esi As Long

Ebx As Long

Edx As Long

Ecx As Long

Eax As Long

Ebp As Long

Eip As Long

SegCs As Long

EFlags As Long ' لعرض القيمة المسجلة للاعلام المعالج

Esp As Long

SegSs As Long

ExtendedRegisters(MAXIMUM_SUPPORTED_EXTENSION - 1) As Byte

End Type

Public Type FLOATING_SAVE_AREA

ControlWord As Long

StatusWord As Long

TagWord As Long

ErrorOffset As Long

ErrorSelector As Long

DataOffset As Long

DataSelector As Long

RegisterArea(SIZE_OF_80387_REGISTERS - 1) As Byte

Cr0NpxState As Long

End Type

80

512

نعلن عن المتغير reg

Public reg as context

بعد ذلك `GetThreadContext( CREATEPROCESSINFO.hThread ,reg)`

قبل الحصول على محتويات المعالج يجب تهيئة المتغير reg

```
reg.ContextFlags = CONTEXT_i486 Or CONTEXT_CONTROL Or CONTEXT_INTEGER Or CONTEXT_SEGMENTS Or CONTEXT_FLOATING_POINT
```

بعد استدعاء الدالة السابقة يتم ملء التركيب reg بمسجلات المعالج للعملية التي تحت التنفيذ

يوجد مشكلة وهي ان التركيب context لا يحتوي مباشر على الاعلام z,c,p

يمكن الحصول عليها عن طريق reg.EFlags الذي يحتوي على رقم يدل على حالة جميع الاعلام

TC=reg.EFlags And 1&

TP= (reg.EFlags And 4&) / 4&

TA=(reg.EFlags And 16&) / 16&

TZ= (reg.EFlags And 64&) / 64&

TS=(reg.EFlags And 128&) / 128&

TT=(reg.EFlags And 256&) / 256&

TD= (reg.EFlags And 1024&) / 1024&

TO= (reg.EFlags And 2048&) / 2048&

2-الكتابة

تتم الكتابة عن طريق الدالة **SetThreadContext**

```
Public Declare Function SetThreadContext Lib "kernel32" (ByVal hThread As Long, lpContext As CONTEXT) As Long
```

البارمتر مثل الدالة السابقة

معظم استخدامات هذه الدالة جعل المعالج ستمر خطوة خطوة

كتابة نقطة توقف Breakpoint

من المعروف ان المقاطعة رقم 3 هي التي تجبر المعالج على التوقف (int 3) وان الهكسي كود المناظر لها

هو CC او 204 بالنظام العشري

كيف يمكن إضافة المقاطعة 3 للبرنامج المصحح ؟

عن طريق الكتابة للبرنامج في الذاكرة بواسطة الدالة **WriteProcessMemory**

```
Declare Function WriteProcessMemory Lib "kernel32" (ByVal hProcess As Long, lpBaseAddress As Any, lpBuffer As Any, ByVal nSize As Long, lpNumberOfBytesWritten As Long) As Long
```

نلاحظ ان البارامترات المرسله للدالة يشبه البارامترات المرسله للدالة **ReadProcessMemory**

قبل كتابة نقطة التوقف يجب القراءة من العنوان الذي نريد الكتابة إليه وذلك لإرجاع القيمة الأصلية

```
c = ReadProcessMemory(hprocess, ByVal adr, orgb, 1, n)
```

```
r = WriteProcessMemory(hprocess, ByVal adr, &HCC, 1, ByVal 0&)
```

ثم نقوم بالكتابة للعنوان المراد التوقف عنده وذلك بكتابة الرقم &Hcc

عندما يمر المعالج على العنوان الذي كتابنا به نقطة التوقف يحدث استثناء وهو من نوع [breakpoint](#) في حدث EXCEPTION_BREAKPOINT نقوم بإرجاع البايث الأصلي وإرجاع المعالج خطوة للخلف

```
Case EXCEPTION_BREAKPOINT  
  
r = WriteProcessMemory(hprocess, ByVal adr, orgb, 1, ByVal 0&)  
  
setPmov CREATEPROCESSINFO.hThread
```

كيف يمكن ان ارجع خطوة للوراء ؟

من المعروف ان العلم TF هو المسئول عن جعل المعالج ينفذ خطوة خطوة إذا كان TF=1 كيف

```
Sub setPmov(thr As Long)  
  
reg.EFlags = &H100 ' TF=1 لجعل  
  
للرجوع خطوة واحدة ' 1 - reg.Eip = reg.Eip  
  
c = SetThreadContext(thr, reg)  
  
End Sub
```

وذلك بوضع القيمة &H100 في reg.EFlags وبالنظام الثنائي 100000000 ومرتبة العلم TF هي 9 والخانة رقم

9 تحتوي على 1 فيصبح TF=1

سنكمل في الجزء الثاني باقي الافكار وسنتكلم عن تحليل الكود ومعظم ميزات

ollydbg