Exploitation in the ‘New’ Win32 Environment

Basics of DEP / Stack Protection Evasion in Windows XP SP2/Windows 2003

Walter Pearce
Computer Security Consultant
IOActive Inc.

Prologue

Over the past three years, there has been a significant shift in security architecture and priority throughout the industry. Consequently, the research and development areas of the industry have drastically changed as well; In a nutshell, leaving much to be desired. Only five years ago, an abundance of buffer overflow articles, worms, exploits, and advances in general were the norm. Everyone knew the basics when it came to writing an exploit, and everyone wrote a paper on it.

Today, to put it lightly, that is no longer the case. Although Unix based exploitation remains largely the same (with the exception of StackGuard and the like); Win32 auditing, exploitation and research has become far, far more complex. With the release of Windows XP SP2 and Windows 2003, everything moved to a new level. The sun had set for the 'simple' core system exploits with the advent of DEP (Data Execution Protection) and the implementation of a host of new security measures within the new compilers, not even considering the .NET Framework and the implications this has had on development as a whole.

This paper is meant to focus on these changes in architecture made to prevent exploitation of win32 processes, and how to break them. After a long search for articles, papers, examples, and other resources covering this area, I found the internet rather lacking. It does not seem (in my eyes) to be a well covered area. To be more specific, the only easily locatable and detailed writing on this specific subject I was able to find was David Litchfield's *"Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server."* ***(Reference 1)***. I certainly may have missed other articles on the subject matter, and I apologize in advance for any repetitiveness in this article with others (many of the topics covered here are re-explanations of David Litchfield's paper).

Due to this, I felt it appropriate to re-iterate the things I have learned in the general areas of Win32 exploitation, and go over in detail the techniques to evade stack protection in Windows XP SP2 and Windows 2003.

What You Need

Lets not forget none of this happens out of thin air (snicker snicker?), and I've compiled a list of the applications and general knowledge I'll assume you have (and that I use in my examples and explanations) to make it through this eccentric piece of text alive.

Our Toys:

- A **Win32 C/C++ Compiler** (I'm using Visual Studio .NET 2005, so all examples will be as such)
- **OllyDbg** or an equivalently bad-looking debugger that serves its purpose that you are so heart-warmingly dedicated to. (SoftIce or WinDbg may do, but don't expect instructions on how to use them)

Optional Goodies which are cool to have anyways:

- **The Win32 NASM Assembler**
- **IDA Pro** always comes in handy for disassembling an application (Then again, if you have this either its an illegal copy or you probably know more about all this than me)
- 1(or more) **Brain**(s)
- Microsoft's **Vadump.exe** Utility

Perfect. Now you have everything applications wise you'll need to follow my wonderful step-by-step explanations. Next, here are some basic topics that I **will** be covering, but not in any sort of real detail past self-reinforcement. I recommend you read all the references at the end of this paper if you require any 'freshening up' in these areas.

- Understanding of Assembly in the Win32 Environment
- General memory stack layout and structure
- How buffer overflows work (If you don't know this, I don't think you should be here)
- SEH Overwrite Exploitation
- Anything else that I can't think of you'll obviously notice

I'll re-iterate, references are available at the bottom for all these topics and anything else I thought was good pertaining to the subject at hand.

Explanation of Win32 Buffer Overflows

Back to Basics

The basics of performing successful buffer overflows in Windows applications is practically the same as with Unix based overflows. There are only a few slight differences in Windows rather than Unix when it comes to the basics. I only mention this because I (and I would think most anyone that looked into security at all), began learning buffer overflows and the like in Unix. I know I did, and I believe it makes sense making such a relationship for anyone out there who may be in the same boat as I was. So, Let's dive right in shall we?

Appendix 1 - Vuln1.c

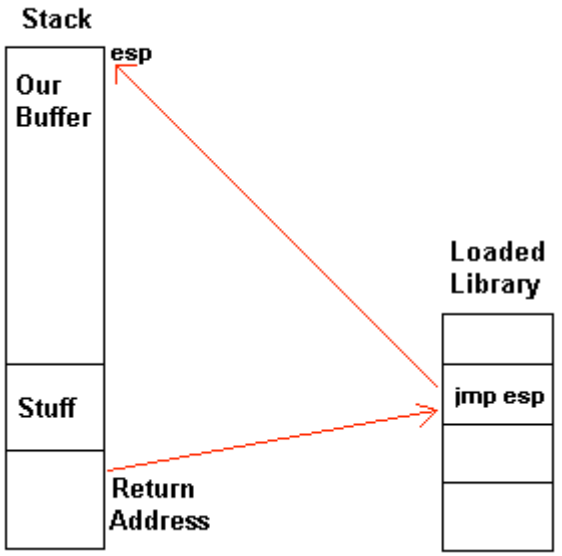
```
#include <stdio.h>

int main(int argc, char *argv[]) {
        char buffer[200];

        strcpy(buffer, argv[1]);

        return(0);
}
```

Obviously this example is pretty simple. I just want to point out a few differences. In Unix, an exploit for this would be rather simple, regardless of which approach you took. The simplest and generally most understood method in Unix is:

- Overflow the buffer and overwrite the routine address of the routine with an address within our buffer
- Upon the functioning calling the RET opcode, EIP is switched to our supplied address, thus rendering control of execution flow to us (Our goal, yippee)
- Within the buffer, we have executable shellcode that is then ran and gives us our shell, connectback, whatever we want.

Yes, yes, I know. This is all very basic. However, within Windows the approach is slightly different for the sake of simplicity. Rather than overwriting EIP with an address within our buffer, it Is simpler to point to an address of code within a loaded library that is static. Meaning, we point to an external library that will execute our shellcode for us. Let's see a pretty picture of this.

Figure 1 - illustration of compromised execution path



Above, in my excellent paint.exe representation of what I just previously explained, you can see the flow of execution with the red lines. The Return Address points to the address of the 'JMP ESP' instruction inside a loaded library (Which are only compile-dependant, so addresses are static across boots and versions). The ESP register, for my picture, happens to be the beginning of our buffer. When this JMP opcode is executed, it bounces back to our buffer, thus beginning execution on our code in a rather elegant way. This method of returning to loaded libraries and jumping back allows for a much larger range of attacks; be it tiny payloads by calling already loaded and located functions, or as you'll see later, to more devious means.

Note however, that for the sake of this example ESP was set to the beginning of our buffer. The majority of the time, this sadly is

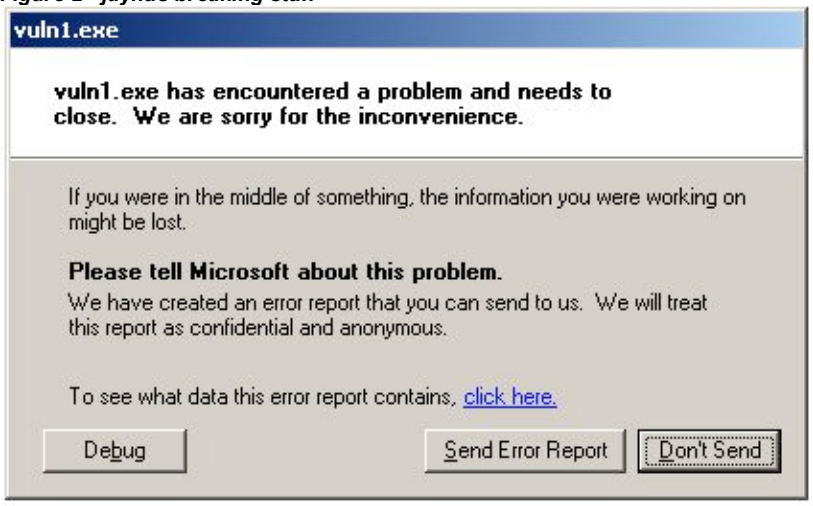
not the case. It may take a certain amount of data manipulation and search to find the correct opcode in combination with the right register to be able to get back somewhere in our buffer. This will be covered in further detail later on.

SEH Unveiled

SEH, or Structured Exception Handling, has become a rather useful tool in performing exploits in the windows environment. It now only allows for more attack vectors, but also cross-platform and more reliable exploits in general (Uh oh!). Exceptions allow us even further flexibility in exploiting applications, and should be understood before you try to make use of it.

Within the Win32 Framework, exceptions are thrown whenever an error or illegal instruction occurs, be it thrown from the user or the operating system itself. Programmatically, when an exception is thrown, the application has a chance to catch the exception and deal with it, and thus allow the program to continue execution. If no user-defined exception handlers are defined, than the operating system takes over, catching the exception, killing the process and giving you that wonderful 'problem' window. Send Error Report indeed, Dr. Watson (hah?).

Figure 2 - jaynus breaking stuff



Now, seeing this window is *usually* a good thing in our field, oddly enough. But its time to understand what exactly causes this voodoo magic window to appear.

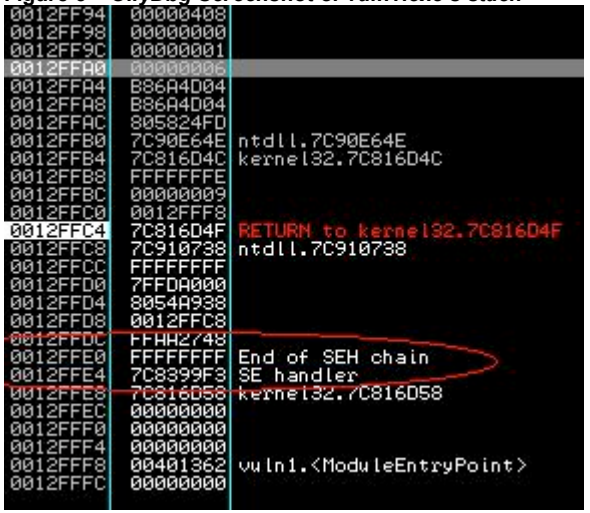
Inside any given Win32 process, the active stack now contains pointers for these new-fangled exception handlers. These are used and referenced by the system on the event of an illegal operation. By default, these pointers point to system handlers, thus giving us our error message. The references are good old struct's, defined as follows.

Appendix 2 - _EXCEPTION_REGISTRATION Structure

```
typedef struct _EXCEPTION_REGISTRATION {
       _EXCEPTION_REGISTRATION *next;
       PEXCEPTION_HANDLER  handler;
} EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;
```

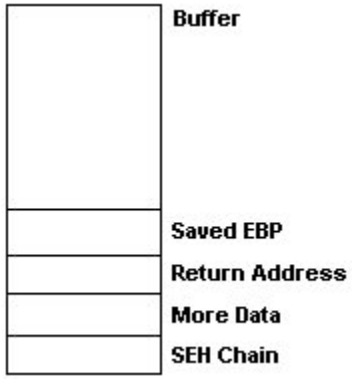
The exception handlers for any given process are always organized in a linked list 'chain'. That is, each record contains a pointer to the next record (_EXCEPTION_REGISTRATION *next), the terminating records pointer containing 0xFFFFFFFF.

Figure 3 – ***OllyDbg Screenshot of vuln1.exe's stack***



The chain of handlers is always on the bottom of the stack, conveniently enough labeled for us by OllyDbg. In the actual stack, this data is stored in 16-byte succession and ending with when the pointer is terminating. In Figure 3, you can see the OllyDbg output of the stack, with the labeled 'End of SEH chain' at 0012ffe0. So, we can see directly below that terminator in the next 8 bytes at 0012ffe4, the default handler for our vuln1.c executable is a pointer to 7c8399f3, which so happens to reside in the loaded address space of kernel32.dll and is the default exception handler for this compilation. Of course, this is within an application with no defined exception handlers, thus the compiled defaults set in. In other circumstances, the exception chain will grow longer, containing more handlers for different operations.

Figure 4 – Illustration of SEH Chain location on the stack



All that mumbo-jumbo aside, it is perfectly possible to abuse this handler for a multitude of purposes, the most obvious being taking control of the execution flow of the program. If we can get any type of exception to be thrown (an illegal EIP from an overflow?) then this exception handler will be called. What is to happen if we overwrite the stack further than the buffer, past the return address, and onto the SEH chain? We can use the same jump back method discussed above, thus dropping the chain of execution into our buffer.

Returning with instructions inside loaded modules

As I mentioned before, I would come back to this jump back method from loaded libraries. Now I am. Finding these instructions somewhere inside all the loaded library modules could be a very monotonous task without some sort of automation. Using your debugger to search the loaded modules for the appropriate JMP, call, or any other series of instructions you need to call is one way of doing it, but usually only when something a bit more custom is required. Microsoft's Vadump.exe ***(Link 1)*** is also capable of serving this purpose, dumping the address space of the specified process PID. For a bit faster and user-friendly approach however, one can always search the Metasploit Opcode Database, which contains a database of the global Windows DLL's useful instructions across multiple versions.

As I said, we are not always lucky enough to have a register pointing directly at our buffer or shellcode. In these cases, we would need to return to more specific instructions to get to our actual shellcode. When the exception handler pointer or return address is being executed, let us say for instance the CPU Registers look like this:

Register	***Address***
EAX	00000000
ECX	0012FFB0
EDX	41414141
EBX	41414141
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C910738
EIP	41414141

As you can see above, this example is as if we have just overflowed our application. The EIP and EBX have been overwritten by our buffer data, and the rest of the registers are however the application left them. These two registers were overwritten with the execution of the return code, and we would have conventional control of the process via the EIP. However, let us pretend our stack looks like the following and we can go a step further with SEH usage.

Address	*Data*	
0012FFC4	00000000	<------- Address of ESP
0012FFC8	459c2fee	<------- Data
0012FFCC	41414141	<------- beginning of our buffer
0012FFC0	41414141	
0012FFC4	41414141	

This is a rather rudimentary example, but as you can see if we were to point our return address or exception handler at a JMP ESP instruction, it wouldn't do us any good because the actual address ESP references (0012FFC4) is not part of our stack. Therefore, we would want to use another instruction from our libraries that would give us an appropriate location. In this case, it would be ideal to find a CALL [ESP + 8], because our stack begins 8 bytes below the address ESP references. Simple enough, yes? Yes. But with the new protection mechanisms in Windows XP SP2 and Windows 2003, we have to change our methods, and sadly it does get a bit more complicated.

New Protection Mechanisms in Windows XP SP2 and Windows 2003 and Visual Studio .NET

Overview

With the advent of Windows 2003, a number of new mechanisms were created in an attempt to thwart security vulnerabilities and disable the attack vectors we use today. Although I applaud Microsoft for this attempt, it is flawed. DEP (Data Execution Protection) was implemented as a security measure in Windows 2003, Windows XP, and also is the same basic mechanisms behind Visual Studio .NET's /GS compilation flag. As it was so daftly said before me, "Currently the stack protection built into Windows 2003 *can* be defeated", all it takes is a lot of ingenuity and a new perspective on things. I'm getting ahead of myself however. We will start with the basics.

Stack Cookies

First and most importantly, a type of 'Cookie' (or canary, or whatever you want to call it) has been added to the stack. The cookie is an 8-byte unsigned int pseudo-randomly generated value put directly in front of the return address. When I say 'pseudo-random', don't think of guessing it. It's a virtual impossibility. If you're interested, see Appendix 2 on how these cookies are actually generated. This cookie is then saved in a secure version in the .data section of the executable upon execution; whenever a return address is called, this cookie is authenticated against this saved version. If these cookies do not match, then a security exception is thrown and the application is stopped.

Figure 5 – Illustration of Cookie location on the stack

This all boils down to one conclusion; when the buffer is overrun in an attempt to gain control of execution, this stack cookie is going to have to be overwritten on the way to the return address or the SEH chain, and the cookie will be checked before any execution of our code is done, and the process will be terminated. Surprisingly enough this comes with little affect to processing times and has certainly been well implemented by Microsoft.

When a cookie is all alone

David Litchfield best covered this topic in his paper in Reference 1, but I believe it is best to rehash from a different perspective. Let's take a deeper look into what happens when the cookie is not validated by its authorized sister in the .data section of the stack.

When the discrepancy is first detected, the system checks for a security handler in the .data section of the executable. In most

instances this is not defined, but if it is this handler processes first and then the exception is handled and nothing is given to the system. If, and most commonly, no handler is defined, the UnhandledExceptionFilter method is called. This eventually leads to the generic error message for all unhandled exceptions in win32 applications, where the ReportFault method is called and the window in Figure 2 is displayed.

Authentication Tables

In DEP protected executables, and executables compiled with the /GS flag, authenticated cookie values are saved in the .data section of the header, as well as authenticate saved addresses to the security handlers of the binary. That is, it saves a list of the pointers to registered exception handlers, and checks the address of the handler against this list before executing. If the address is not in the list, it does not execute it.

Checking for security handlers in practice

Appendix 4 – DeclaredSEH.c

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

int main(int argc, char *argv[]) {
	char buffer[200];


	__try {
		strcpy(buffer, argv[1]);
	} __except(GetExceptionCode()) {
		printf("Exception Raised\n\n");
	}


	return 0;
}
```

Compiling the code snippet above, we can search the .data section of memory with OllyDbg for these authenticated handler addresses, being that we have defined an exception handler in our application. This only comes into play when an exception handler has been declared by the programmers, which is under normal circumstances a rare case. Nonetheless, knowing the mechanics behind the exceptions is a must to truly understand what is going on.

For example, compile this snippet using Visual Studio .NET. Open it for debugging in OllyDbg, without any arguments. We don't want any arguments because this will result in an exception being thrown inside strcpy because argv[1] is an empty pointer, therefore an access violation occurs.

Inside OllyDbg, open the memory window. At the bottom, there is the data block section of the main thread. In this section of memory, we can see at the very top is "(Pointer to SEH Chain)". This is the executables storage location for the address of the first item in the SEH Chain.

Now, this is only the default location upon memory load of the application. We want to see if this executable has its own declared exception handlers, and not just the default win32. If the executable has its own declared handlers, we can watch this data location for changes to see when it loads the SEH Chain, and whether the address is local to the process. Right click on the Pointer address, and set a 'Hardware, on write' breakpoint, and then run the application.

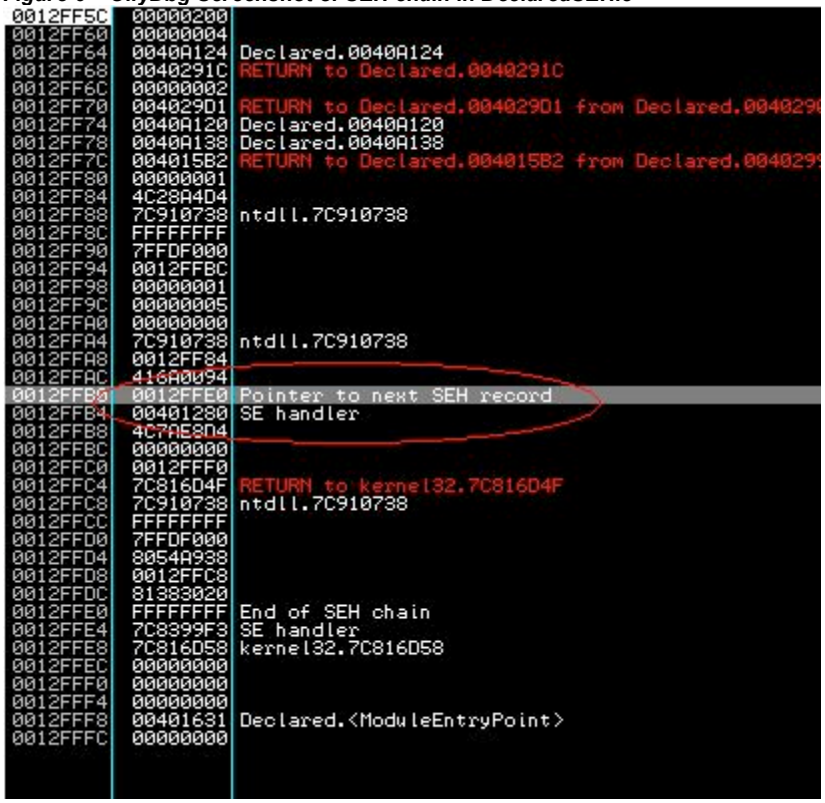
The application will break when the address in memory is written too many times; just continue running through it until we see our instruction code has broken out of the initial windows initialization and into our actual code block. In this instance, once the pointer to the SEH chain has changed to 0x0012FFB0, stop running the application. Go ahead and switch to the active stacks memory window and go to this address. As you can see in Figure 6, this address contains a _EXCEPTION_REGISTRATION instance, which when you follow each point to the next SEH record, the chain looks like this.

Register	*Data*	*Notes*
0012FFB0	0012FFE0	Pointer to next SEH record
0012FFB4	00401280	SE handler
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler

As you can see, the first registration, who's handler is at 0x00401280, is located within this applications address space.

Therefore, it's safe to assume this executable has its own defined handlers.

Figure 6 – OllyDbg Screenshot of SEH chain in DeclaredSEH.c



Now, I know everyone will hate me for this and anyone who has used OllyDbg would know what I skipped going through all this. Now that we know how SEH structures are built within the stack, OllyDbg does have an 'SEH Chain' view that displays all the current exception handlers for the running executable. Breaking ahead a bit will allow you to see the live SEH Chain upon execution of our code.

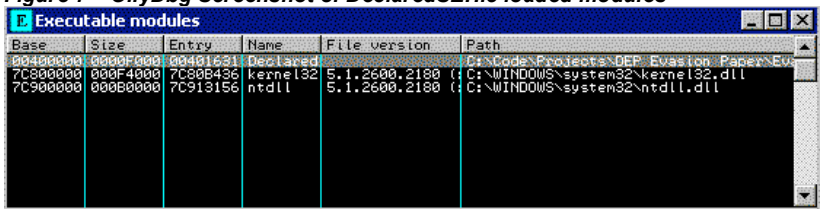
Vulnerabilities in DEP

Breaking stack protection using exception handlers

Ah ha! We haven't been going over all this exception handling for no reason. Abusing the way SEH operates turns out to be the one of the least time consuming ways to evade DEP and allow our arbitrary code to run.

The flaw allowing for this exists in the way the system checks the exception handler's pointer against the authoritative table. If the exception handlers address is outside the address range of any loaded module, than it is executed anyways. That is, if the address is within a module currently in memory, but that module is not loaded into the current executable, than the instructions at that address are executed.

Figure 7 – OllyDbg Screenshot of DeclaredSEH.c loaded modules



For example, let's take our DeclaredSEH.c and open it up in OllyDbg again. Opening the Executable Modules window, we can see not many modules are loaded in this application. Excluding kernel32 and ntdll, anyone familiar with win32 programming and come up with a list of modules off the top of their head that are sure to be loaded somewhere else besides within our target.

user32.dll, shell32.dll, gdi32.dll, ws2_32.dll, ws2help.dll, unicode.nls, advapi32.dll

And that is just to name a few. Needless to say, many of these system libraries that we can almost be certain of being loaded are available.

Loading Data Directly into the Heap

This method if attack is frankly very straightforward, very simple, and rather disappointing considering the time we can see Microsoft put into these protections. Upon authenticating security handler codes against the .data section, there is a small exclusion before the actual check occurs. If the address of the handler is within the heap, the address isn't authenticated, and execution flows just as if it was any other SEH handler redirect. This is highly application independent however, and strictly depends on 3 factors.

A. You must be able to get buffer data into the heap. Excluding shellcode evasion methods and search methods, this means being able to get a working shellcode into a heap buffer. In many cases, this is not possible.
B. You must be able to accurately predict the address this buffer will be placed at in the heap.
C. If the area in which you are deploying your payload on the heap is not executable, there is no way for your code to run.

Aside from these mitigating factors, it is completely possible to evade DEP completely by using the heap as your buffer location and just using the overwrite buffer to access the SEH.

Double Cookie Overwrite

Luckily for us, in certain scenarios we may actually be able to avoid all this DEP wonderfulness by just being able to assign our own stack cookies in the canary location and the authoritative table, in essence bypassing the whole purpose of this configuration to begin with.

Function Pointer Overwrites

This method is again one of our more circumstantial methods in which to gain control of execution. However, you would be surprised how many function pointers actual exist within an application. Our general objective here is to find a function pointer that we can say with a degree of certainty will be used after our payload has already been written to memory.

Appendix

Appendix 1 – vuln.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
	char buffer[200];
	strcpy(buffer, argv[1]);
	return(0);
}
```

Appendix 2 – _EXCEPTION_REGISTRATION structure

```
typedef struct _EXCEPTION_REGISTRATION {
	_EXCEPTION_REGISTRATION *next;
	PEXCEPTION_HANDLER  handler;
} EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;
```

Appendix 3 – Stack Cookie Generation

(This code is from David Litchfield's paper, Reference 1)

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

int main(int argc, char *argv[]) {
	FILETIME ft;
	unsigned int Cookie=0;
	unsigned int tmp=0;
	unsigned int *ptr=0;
	LARGE_INTEGER perfcount;

	GetSystemTimeAsFileTime(&ft);
	Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;
	Cookie = Cookie ^ GetCurrentProcessId();
	Cookie = Cookie ^ GetCurrentThreadId();
	Cookie = Cookie ^ GetTickCount();
	QueryPerformanceCounter(&perfcount);

	ptr = (unsigned int)&perfcount;
	tmp = *(ptr+1) ^ *ptr;
	Cookie = Cookie ^ tmp;

	printf("Cookie: %.8X\n",Cookie);
	return 0;
}
```

Appendix 4 – DeclaredSEH.c

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

int main(int argc, char *argv[]) {
	char buffer[200];


	__try {
		strcpy(buffer, argv[1]);
	} __except(GetExceptionCode()) {
		printf("Exception Raised\n\n");
	}


	return 0;
}
```

References

1. Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server
   http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf
2. Phrack Issue 63, Article 15 - NT Shellcode Prevention Demystified
   http://www.phrack.org/show.php/phrack/5/phrack/28/show.php?p=63&a=15
3. Security Forest Wiki
   http://www.securityforest.com

Links

1. Microsoft's vadump.exe
   http://www.microsoft.com/windows2000/techinfo/reskit/tools/existing/vadump-o.asp
2. Metasploit Opcode Database
   http://metasploit.com/users/opcode/msfopcode.cgi