Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows platform

David Litchfield [davidl@ngssoftware.com]
30th September 2005

An NGSSoftware Insight Security Research (NISR) Publication

Buffer Underruns and Stack Protection

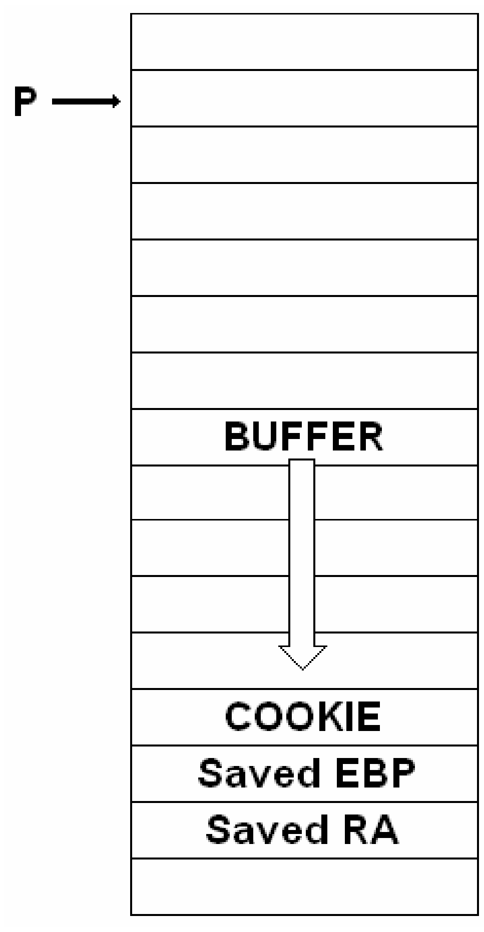
Starting with Windows 2003 Server, Microsoft introduced a number of Exploitation Prevention Mechanisms (XPMs) into their software. Over time these XPMs were refined as weaknesses were discovered [1][2] and more XPMs were introduced. Today the XPMs have been added to Windows XP Service Pack 2 and Windows 2003 Service Pack1 and include protection of the base pointer and saved return addresses by use of a security cookie or canary on the stack, variable re-ordering, parameter saving, NX/DEP, software DEP and Safe SEH. XPMs are realized with a combination of architectural changes to the OS, hardware capabilities and modifications to the Microsoft compilers by inserting procedure prologues and epilogues to potentially dangerous functions, the latter commonly known as "GS", named after the flags used to turn on "stack protection". Whilst there are recognized improvements that can be made to XPMs relating to the heap, in most cases where code still contains a stack based overflow, the current incarnations of the stack related XPMs make it extremely difficult, if not impossible to exploit. This is true of buffer overrun vulnerabilities; however, this is not true of buffer underrun vulnerabilities. Consider the following code:

Code listing 1

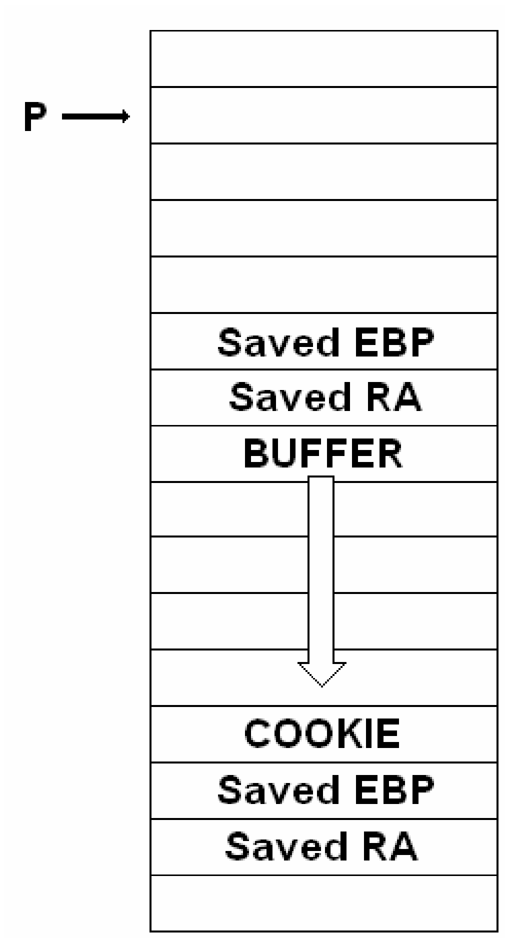
```
#include <windows.h>

int foo(char *str);


int main(int argc, char *argv[])
{
      foo(argv[1]);
      return 0;
}

int foo(char *str)
{
 int padding = 0;
 int i=0;
 char *p=NULL;
 char buffer[33]="";

 // Ooops!
 padding = 32 - strlen(str);

 for( i = 0; i < padding; ++i )
      buffer[i] = '0';

 // Ooops, again!
 p = &buffer[ padding ];
 lstrcpy( p , str );
 printf("%s\n",buffer);
 return 0;
}
```

In the foo function, there's a buffer underrun vulnerability. The strlen() function gets the length of the string and subtracts this from 32. If the string is longer than 32 bytes then the integer “padding” goes negative. Thus when we get the address of buffer[padding] then, the address is outside of the buffer - at an address lower on the stack than buffer - in other words a buffer underrun.

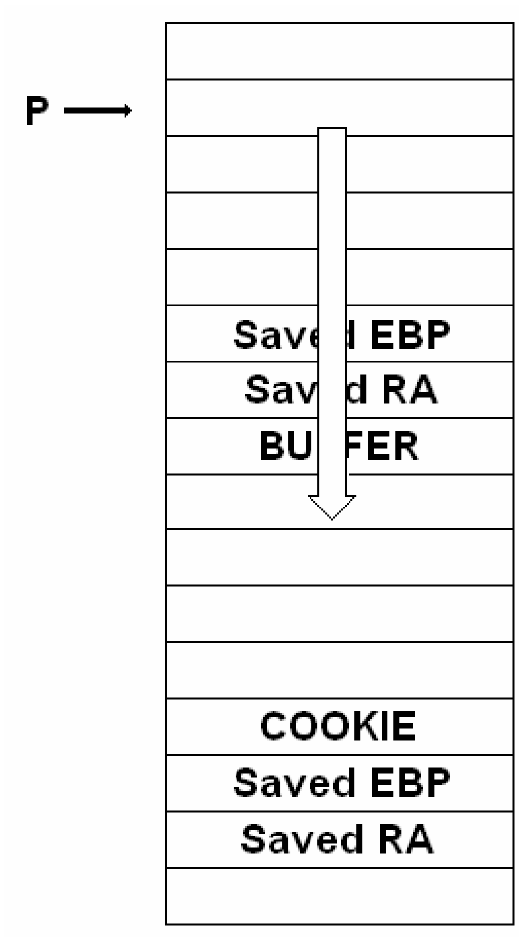
Before the lstrcpy() function is called our stack looks like this:



Our pointer p points to an address outside of buffer at an address less than buffer. Note our saved base pointer and saved return address are protected by the cookie. Once the lstrcpy function has been called our stack frame looks like this:



Note that as the lstrcpy function contains no buffers of its own it has no need for a cookie to protect the saved base pointer or return address. When we start copying data from the source (str) to the destination (p) look what happens, however:



The act of copying from the source to the destination overwrites the saved base pointer and, crucially, the saved return address. As such, when lstrcpy has finished and returns it does so to a location of the attacker's choosing.

While buffer underruns are not common they do exist, a good example being in MIT Kerberos [3].

In terms of exploitation this is mitigated by DEP. Or is it? An extremely simple way of defeating DEP is a return into LoadLibrary

Return-into-LoadLibrary

Return-into-libc attacks are a well known method of defeating non-exec XPMs [4],[5]. The idea is to overwrite the saved return address with the address of system() (or another useful function) and set the stack in such a way that it will spawn a shell on entry [6]. In more complex return-to-libc attacks it's possible to chain together a number of functions to achieve the goal. As far as Windows is concerned, one method of defeating DEP that has been discussed already is to return to VirtualAlloc(to allocate executable and writable memory), then lstrcat the shellcode to the newly allocated memory, then return to the shellcode [7]. A simpler solution is to return to LoadLibraryA. LoadLibrary takes as its first and only parameter a pointer to the name of the library to load. Once loaded, any code in the DllMain() function will be executed.

As LoadLibrary will accept a UNC path as a parameter this can be used for remote exploitation. What exacerbates the problem is the WebDAV redirector: if the host in the UNC path cannot be contacted over TCP 139 or 445 it will attempt to connect over the web on TCP port 80. Thus, if a firewall prevents outbound client connections on TCP 139/445 but not port 80, then the attacked system will download the library from a web server and load it. [Note: To prevent this behaviour disable the WebClient service.]

At first glance it would seem that, to do this successfully, the address of LoadLibraryA must be known beforehand as well as the location of the UNC path in memory. If address space layout randomization (ASLR) is employed then this would make knowing these values impossible and thus make returning to LoadLibraryA impossible.

Address Space Layout Randomization

As the term implies, address space layout randomization locates the stacks, heaps and base load addresses of libraries at random locations and there are third party tools available for Windows to allow this. WehnTrust [8] is a great example of such as tool. On Windows, the current problem with existing ASLR solutions is relocation of the main executable itself – whilst the location of everything else is randomized the main exe, not having a .reloc section, can still be found it the same place, though I'm reliably informed that the WehnTrust team have a few cool tricks up their sleeves that they're playing with. For the time being however, ASLR on Windows can be defeated in many cases.

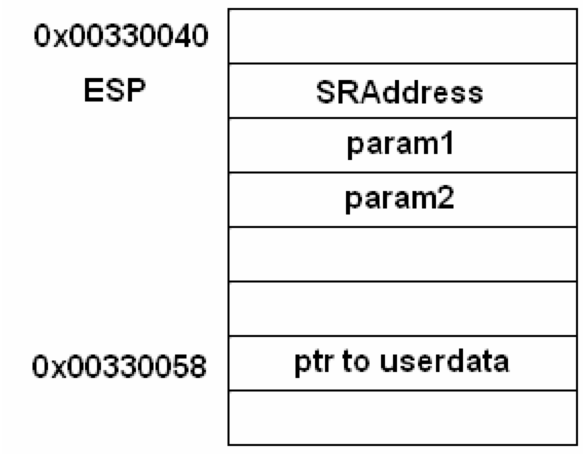
Rather than overwriting the saved return address on the stack with the address of LoadLibraryA in kernel32.dll, we can overwrite the saved return address with an address in the main executable that executes

```
        CALL DS:LoadLibraryA
```

With the location of the saved return address being at ESP, this would require that a pointer to the attacker's DLL be located at ESP + 4. Thus, when we return to this call instruction and it executes, the pointer to the name of the library to load is at the right location. For this to happen either a pointer to the library would need to have been the first parameter pushed onto the stack to the vulnerable function, or the pointer is written by the attacker. With ASLR the latter would not be a possibility and if, given the actual vulnerability, the former is not the case then a bit more work is required.

Firstly, the stack should be searched for a pointer to data controlled by the attacker. Once found, its offset from the location of the saved return address is then measured and for each DWORD a new return address is set – each new return address should point to an address in the main executable that executes a RET instruction – thus giving a chain of returns. With each RET, the stack pointer is indirectly adjusted – the equivalent of adding four to the ESP. Last in the chain is the return to the call LoadLibrary.

So assuming our stack looks like this before the real saved return address is overwritten…



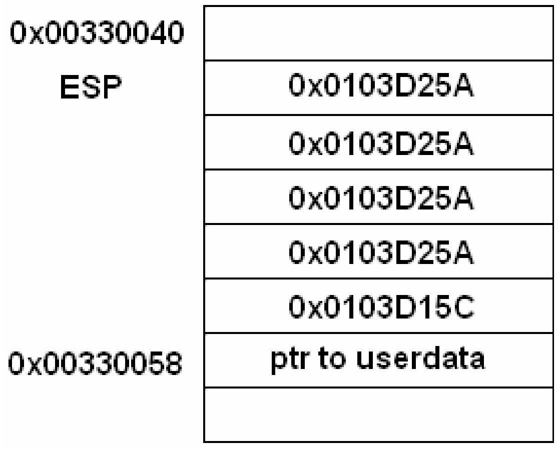
… and assume we're targeting winlogon.exe on Windows XP Service Pack 2. This has a base address of 0x01000000 and at address 0x0103D15C we can find the following instruction

```
        CALL DS:LoadLibraryA
```

At address 0x0103D25A we can find the instruction:

```
        RET
```

On overwriting the saved return address the attack would leave the stack in the following state:



When the vulnerable function returns it does so to 0x0103D25A. The instruction at this address is "`RET`" and when this executes we land back at 0x0103D25A. This continues three more times until we return to 0x0103D15C. The instruction at this address is the "`CALL DS:LoadLibraryA`" and when this executes the library pointer to by "ptr" is loaded.

If the return addresses used have a NULL then this technique may not be possible unless the overflow in Unicode in nature.

Whilst LoadLibraryA is used here as an example there could be other possible, "useful" functions. WinExec provides an interesting (though unlikely!) alternative. It too will accept a UNC path. There of course will be others. The point is that, unless the main executable is relocated, an attacker can still run code of their choosing; that said, having the protection provided by ASLR is better than not having it. An attacker needs to know you've got it to defeat it. Additionally, there will be vulnerabilities that can't be exploited using the techniques described here.

Protecting the saved return address more effectively.

It is necessary to protect the saved return address more effectively. Whilst a cookie or canary acts as a goal keeper in the case of stack based overflows we have seen it doesn't protect from stack based buffer underruns. Providing that there is *true* randomization of the stack location, one idea to protect the saved return address itself would be to XOR it with the address at which it can be found. See Code Listing 2

Code Listing 2

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

```
int foo();

#define PROLOG { \
                  __asm lea eax, dword ptr[esp] \
                  __asm xor dword ptr[esp],eax \
                  __asm push ebp \
                  __asm mov ebp, esp \
                  }

#define EPILOG { \
                  __asm mov esp, ebp \
                  __asm pop ebp \
                  __asm lea eax, dword ptr[esp] \
                  __asm xor dword ptr[esp],eax \
                  __asm ret \
                  }

int main()
{
      foo();
      return 0;
}

__declspec(naked) int foo()
{
      PROLOG;
      printf("hello");
      EPILOG;
}
```

In order to successfully attack this an attacker would need to know the address at which the saved return address is stored at and overwrite the saved return address with a their chosen replacement XORed with the stack address. The strength of this as a potential XPM relies on the location of the stack being random. With there being a finite number of locations where the base of the stack could possibly be located, and with the offset into the stack where the saved return address is stored being consistent, then this XPM could fall to a brute force attack. If the underrun existed in a system service that restarted on failure then this could provide an attacker with the opportunity they need: by keeping their values consistent, the law of diminishing returns suggests their attack would eventually succeed. The degree of likelihood depends on just how finite the number of possible stack locations is. If the implementation provides for 4096 possible locations then the attack has a good chance of success – 262144 possible locations and the chances are much slimmer.

During discussions with colleagues about this topic, several possible solutions were suggested: maintain a separate “stack” of saved return address on a heap allocated at process startup. On return these would be compared with the current return address; another is to XOR the saved return address with a global security cookie; another would be to maintain a copy of the most recent saved return address in the Thread Environment Block and previous saved return address chained on the stack by the current call. This is demonstrated with the following code:

Code Listing 3

```
#define PLOG { \
                  __asm mov eax, dword ptr fs:[0x18] \
                  __asm mov ecx, dword ptr[eax+80] \
                  __asm push ecx \
                  __asm mov ecx, dword ptr[ebp+4] \
                  __asm mov dword ptr[eax+80], ecx \
                  __asm xor ecx, ecx \
                  __asm xor eax, eax \
                  }

#define ELOG { \
                  __asm mov eax, dword ptr fs:[0x18] \
                  __asm mov ecx, dword ptr[eax+80] \
                  __asm cmp dword ptr[ebp+4],ecx \
                  __asm jne KillTheProcess \
                  __asm pop ecx \
                  __asm mov dword ptr[eax+80], ecx \
                        }
```

Doing this would wreak havoc with exception handling, however, in the event of an exception as the chain of return addresses is broken. This is true also of the stack of return addresses saved on the heap; neither is suitable, therefore.

Conclusion

Like everything in security the question really boils down to a risk assessment. What is the likelihood of there being enough buffer underruns in the code to make this a prescient issue? If there is only one such flaw does this mean we should protect each call in such a manner? It's a high price to pay if it were the case. Would this still be true if the next big worm targeted such a flaw and we *didn't* put the protection in place? Until this is answered the best thing to do is review the code of critical or high risk applications to find and remove such flaws. Typically one would expect such errors to be found in code that relates to encoding and decoding such as crypto, SNMP, asn.1, etc.

Finally, it's important to note that whilst there may occasionally be routes through to arbitrary code execution, having these protection mechanisms is far better than not having them. This paper should not be taken as a criticism of any of the exploit prevention mechanisms but rather a demonstration of how the possibilities for successful code execution exploits are rapidly disappearing.

References

[1] http://www.cigital.com/news/index.php?pg=art&artid=70
[2] http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf
[3] http://securityfocus.com/bid/7185
[4] http://archives.neohapsis.com/archives/bugtraq/1997_3/0281.html
[5] http://community.corest.com/~juliano/non-exec-stack-problems.html
[6] http://www.ngssoftware.com/papers/non-stack-bo-windows.pdf
[7] http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html
[8] http://www.wehnus.com