The cross-page overwrite and it's application in heap overflows

By Greg Hoglund

ABSTRACT

Buffer overflows remain the backbone of subversive code attacks. As stack protection finds more use with compilers, such as Microsoft's .NET package, the heap will gain more attention. Heap overflows are certainly not a new topic, but very little has been done to effectively describe how heap overflows translate into control of the instruction pointer. In this paper I discuss a specific technique that can be used directly with heap overflows to gain control of the instruction pointer.

Introduction

I wrote this paper after working for a time with the Abyss Webserver heap overflow as provided within Dave Aitel's CANVAS product. The technique that Dave used is interesting at the very least. Although more reliable methods exist for taking advantage of a heap overflow on Windows[1] I think the cross-page overwrite deserves some documentation.

The heap manager

Heap memory is interlaced with heap control structures. Within the heap lie user controlled buffers – buffers that are subject to overflows and off-by-one errors that are no different from a stack overflow. However, on the heap, you will not find a saved instruction pointer to overwrite. In fact, with an attack against the heap you don't have the convenience of simply injecting a return address. With a heap overflow you get something totally different – you can write a single 32 bit value to any 32 bit address of your choosing.

With a heap overflow you can write a single 32 bit value to a single 32 bit address of your choosing.

Your attack begins by overwriting a heap control structure. The modified heap control structure is processed by the heap manager code and results in user supplied values to be used in a register move operation such as:

```
mov [edx], ecx
```

where `edx` and `ecx` are both user controlled. The exact nature registers used in the move may differ. Regardless, if you don't know your assembly, – this instruction means

[1] Replacing a function pointer used for `RtlEnterCriticalSection` is a popular choice and is used in CANVAS for the Cold Fusion overflow

"move the 32 bit value stored in `ecx` to the address referenced by `edx`". In other words, `edx` has the target address and `ecx` has the value we are going to put there. It is not worth disassembling all the logic that leads to this problem – but simply worth noting the locations from which these two register values are obtained. A little testing in a debugger can usually reveal where in your string these values are obtained. The heap overflow will look something very similar to that illustrated in Figure 1. A heap structure is overwritten and the locations marked as 'CCCC' and 'DDDD' are used to obtain the values in `ecx` and `edx` respectively.

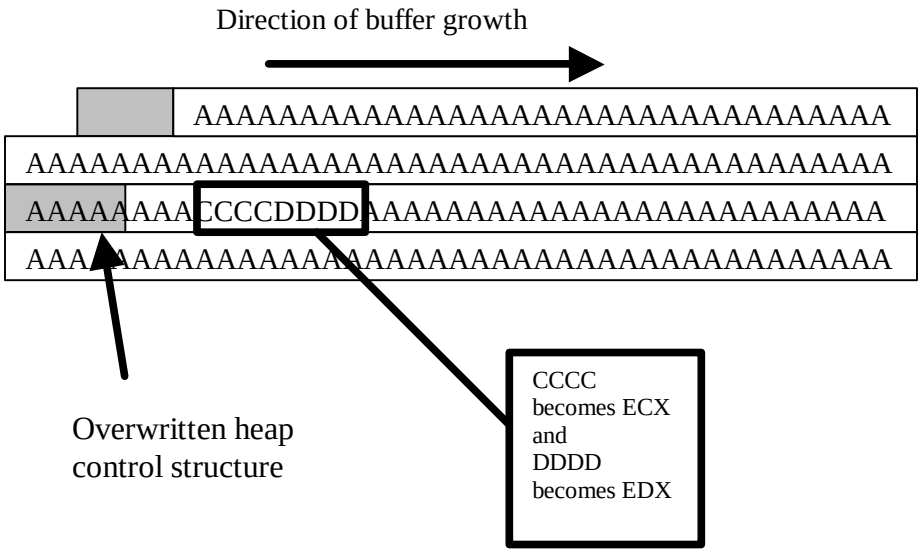

Figure 1

The two questions

Think about this – you must find a location in memory that you want to overwrite. Then, you must figure some value that you want to put in this location. Such an operation is certainly powerful, but how do you leverage this into control of the instruction pointer? You only control TWO things as the attacker – the address you wish to overwrite, and the value you are going to place there. By choosing these two numbers carefully, you can get control of the instruction pointer.

The exception handler

When you overwrite the heap, chances are that you are going to cause the program to crash. Not surprising since have corrupted important memory. Remember that you only get one chance to overwrite some 32 bit value in memory. If the program is going to

crash, the natural choice is to overwrite an exception handler for the current thread. The exception handler will almost certainly be called if the thread crashes.

The nice thing about an exception handler is that it's controlled from a single 32 bit pointer. This means we can overwrite it easily using our heap overflow capability. Second, the location of the exception handler is very consistent – it's the first 32 bit entry in the thread control structure. The thread control structures live at memory locations that can be guessed (in the 0x7FFDxxxx range). So one of our questions – *where* to overwrite memory, has now been solved.

Now look closely at Figure 2. The **exception handler frame** is a data structure. The **exception handler pointer** that is stored at the beginning of the **thread control structure** points to the **exception handler frame**. This frame is located in stack memory.

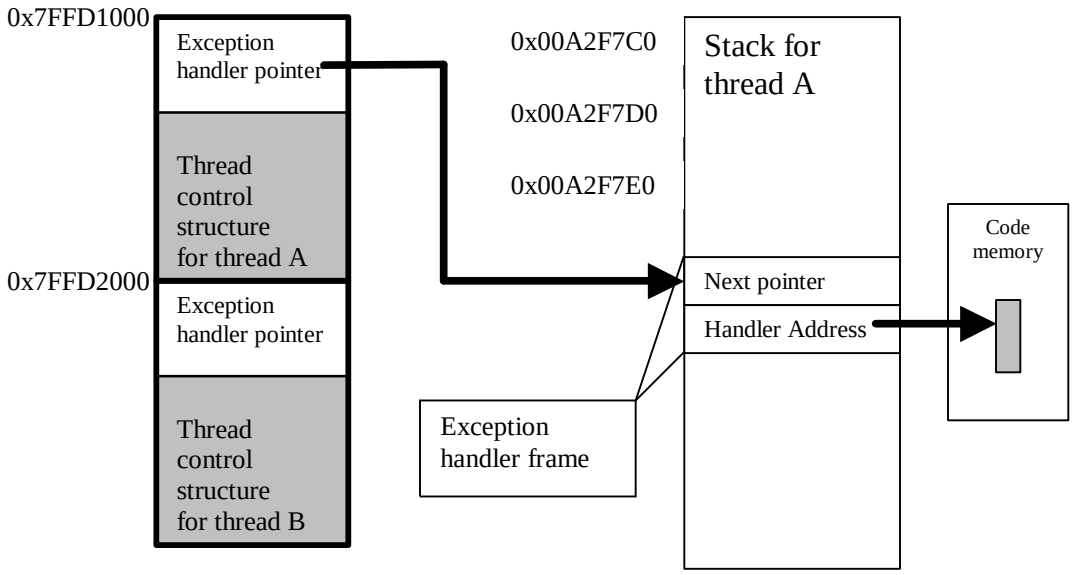

Figure 2

Look closely at the exception handler frame in Figure 2. The first 32 bits are a pointer to the next exception handler frame. For purposes of our attack we can ignore this value. Exception handlers are chained together so that if the current exception handler cannot handle the given exception, it can pass the exception on to the next handler. Since we are attacking the software, we don't want to pass any exceptions down the chain and thus we will never use this value. The second 32 bits represent the handler address. This value is very important to us. It points to code. If we can alter this value somehow to point into user controlled heap memory we can gain control of execution when an exception occurs.

A closer look at the stack

If you examine the stack you will find several pointers into the heap. This is typical when heap pointers are passed to subroutines or used as local variables. Chances are great that at least a few of these heap pointers are going to point into a user controlled buffer. See Figure 3 and note that pointer N3 points into a user controlled buffer. Depending on the program you are exploiting and the environment, these pointers may point directly to the beginning of your buffer, or perhaps somewhere in the middle of your buffer. By using a debugger[2] you will be able to determine exactly where in the buffer the pointer is referencing.

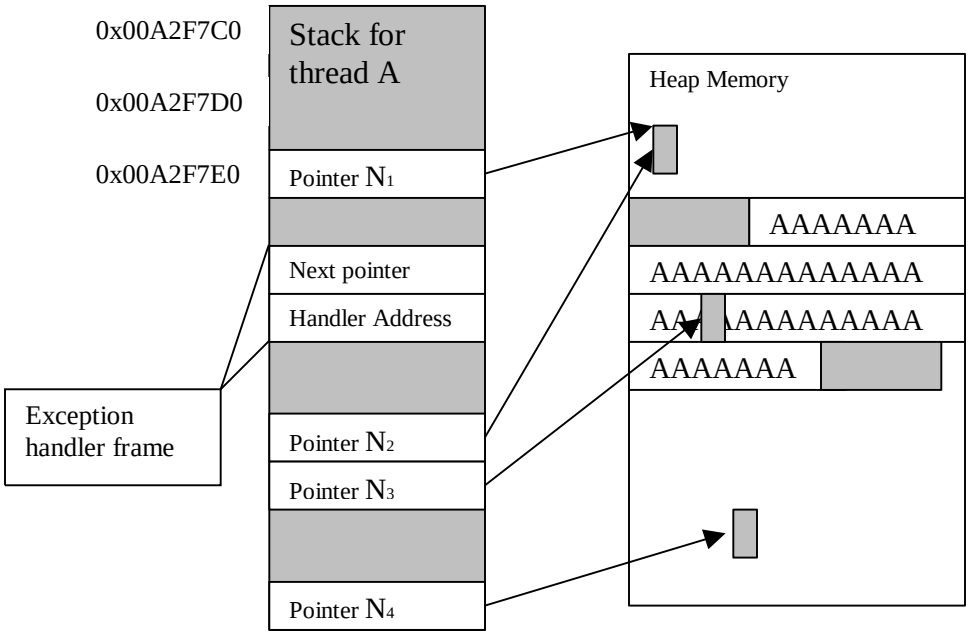

Figure 3

Changing the address of the Exception handler frame

The essence of our attack is the ability to change the address of the exception handler frame. This address is stored at the beginning of the thread control structure, as illustrated in Figure 4. In this example, we alter the exception handler pointer to reference the location just *above* pointer N3. Remember the exception handler frame structure – the first 32 bits are of no concern to us. Only the second 32 bits matter since this is the address of the handler function. In Figure 4 we can see that by changing the address of the exception handler frame we can fool the software into using pointer N3 as the handler function address. This is exactly what we want. This means that if and when

[2] my favorite is OllyDbg

an exception occurs, the system will jump to our user controlled buffer and start executing our injected code.

This leaves just one final problem – how do we change the address of the exception handler pointer?

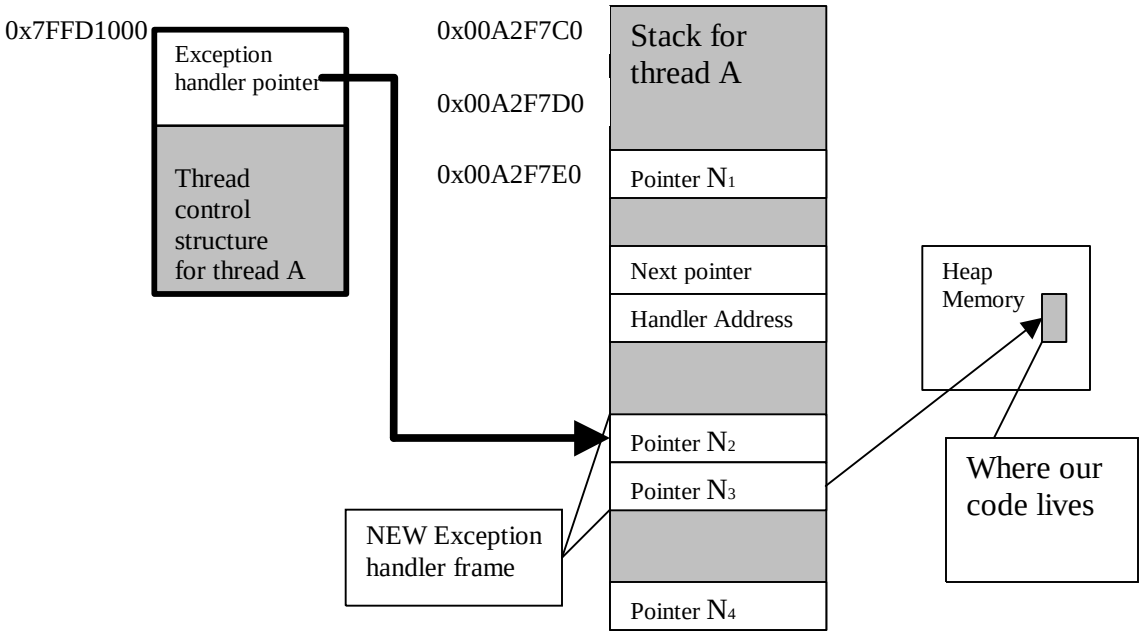

Figure 4

The cross-page overwrite

In a perfect world we would be able to simply alter the address of the exception handler pointer directly. But, look and note that the thread control structure lives at address 0x7FFD1000. Do you see the problem? The address contains a NULL byte. So, we cannot use this value in NULL terminated string operations. Look closer – what is the value we want to put into this address? It's a pointer to the stack. The stack lives in memory at 0x00A2Fxxx. Again we see a NULL byte. So – at first glance BOTH of the numbers we need to inject into our string contain NULL bytes. How do we get around this?

The cross-page overwrite involves writing past the end of one page of memory and into the start of another. The entire purpose of this is to avoid using a NULL byte. See Figure 5. The exception handler pointer is located directly at the beginning of a page 0x7FFD2000. We see that the address of the exception handler is set to 0x00A2F7E0. Remember that little endian byte order causes this to look backwards in memory. Let us assume that for our exploit, we need the exception handler pointer to reference address 0x00A24141. This address obviously contains a NULL byte, as does the target address 0x7FFD2000.

By writing to address 0x7FFD10FE we can write over 2 bytes of the exception handler pointer. This results in writing over the 2 lower bytes of the address stored there. By only writing over two bytes we preserve the upper bytes 0x00A2. Thus, we aren't required to use a NULL byte for our new exception frame pointer – we simply leave the existing NULL byte in place! The resulting exception frame pointer is 0x00A24141. Furthermore, by writing to an address 2 bytes prior to the start of the page, our target address becomes 0x7FFD10FE – also avoiding a NULL byte. We have now solved both of our NULL byte problems.

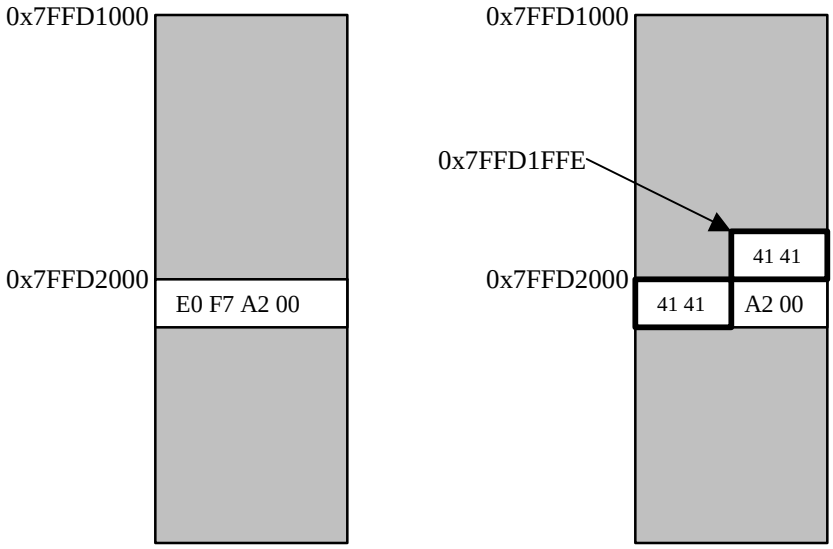

Figure 5

Dummy thread pages

The simple yet effective trick of writing over a page boundary assumes that *both pages exist*. This means that the process needs to be multi-threaded and that two separate threads must exist that are using two *adjacent* memory pages. This is not always going to be the case but there is more you can do.

In many instances, each socket connection to a server will get it's own thread. Using a debugger you should check to what threads exist when you make a connection. Note the location of the thread structure memory pages. You can observe this manually by looking at the FS register and many debuggers have a thread enumeration window. What you want to look for is a thread that has a page directly before the thread you are exploiting. If this isn't the case you will need to experiment with making multiple connections to the server. Try making two dummy connections in addition to your attacking connection. Make one connection before the attack and one after. Make sure

to pause your attack until all your dummy connections are in place. These dummy connections will hopefully ensure that new thread pages have been built up around the thread that your attacking. Once again, check with the debugger and make sure that a thread page *preceeds* the page you are attacking.

Conclusion

I have outlined a somewhat complex technique that allows you to convert a heap overflow into control of the instruction pointer. The technique requires modification of the exception handler pointer at the beginning of the thread control block. To avoid NULL bytes we use a trick known at the 'cross page' overwrite. This trick only works if there is a page mapped into memory directly preceeding the thread page we are attacking. By using multiple socket connections we can sometimes influence the thread pages and increase our chances of success.