Windows Heap Overflow Exploitation

Exploiting a custom heap under Windows 7.

Author : Souhail Hammou

Blog : http://rce4fun.blogspot.com/

Hi , In this article I will be talking about exploiting a custom heap : which is a big chunk of memory allocated by the usermode application using **VirtualAlloc** for example . The application will then work on managing 'heap' block allocations and frees (in the allocated chunk) in a custom way with complete ignorance of the Windows's heap manager. This method gives the software much more control over its custom heap, but it can result in security flaws if the manager doesn't do it's job properly , we'll see that in detail later.

The vulnerability that we'll exploit together today is a 'heap' overflow vulnerability that's occuring in a custom heap built by the application. The vulnerable software is : **ZipItFast 3.0** and we'll be exploiting it today and gain code execution under Windows 7 . ASLR , DEP , SafeSEH aren't enabled by default in the application which makes it even more reliable to us . Even though , there's still some painful surprises waiting for us ...

Please refer to C/C++ courses about singly and doubly linked lists for better understanding of this paper's material.

To see an implementation of a custom heap manager in C/C++ please refer to my previous blog post : http://rce4fun.blogspot.com/2014/01/creating-and-using-your-own-heap-manager.html

Heap Manager Source code : http://pastebin.com/2LgcByyC

Let's just start :

The Exploit :

I've actually got the POC under exploit-db , you can check it right here :
http://www.exploit-db.com/exploits/17512/

Oh , and there's also a full exploit here :
http://www.exploit-db.com/exploits/19776/

Unfortunately , you won't learn much from the full exploitation since it will work only on Windows XP SP1. Why ? simply because it's using a technique that consists overwriting the vectored exception handler node that exists in a static address under windows XP SP1. Briefly , all you have to do is find a pointer to your shellcode (buffer) in the stack. Then take the stack address which points to your pointer and after that substract 0x8 from that address and then perform the overwrite. When an exception is raised , the vectored exception handlers will be dispatched before any handler from the SEH chain, and your shellcode will be called using a CALL DWORD PTR DS: [ESI + 0x8] (ESI = stack pointer to the pointer to your buffer - 0x8). You can google the _VECTORED_EXCEPTION_NODE and check its elements.

And why wouldn't this work under later versions of Windows ? Simply because Microsoft got aware of the use of this technique and now **EncodePointer** is used to encode the pointer to the handler whenever a new handler is created by the application, and then **DecodePointer** is called to decode the pointer before the handler is invoked.

Okay, let's start building our exploit now from scratch. The POC creates a ZIP file with the largest possible file name , let's try it :

N.B : If you want to do some tests , execute the software from command line as follows :

```
Cmd :> C:\blabla\ZipItFast\ZipItFast.exe C:\blabla\exploit.zip
```

Let's try executing it now :

```
00401C6F . 8950 04      MOV DWORD PTR DS:[EAX+4],EDX
00401C72 . 5B           POP EBX
00401C73 . C3           RETN                            EDX = entry->Blink ; [EDX] = entry->Blink->Flink
00401C74 > 8B00         MOV EAX,DWORD PTR DS:[EAX]      EAX = ptr to the FreeList entry for our block ; [EAX] = entry->Flink
00401C76 . 8902         MOV DWORD PTR DS:[EDX],EAX      List->Blink->Flink = List->Flink (EAX)
00401C78 . 8950 04      MOV DWORD PTR DS:[EAX+4],EDX    List->Flink->Blink = List->Blink (EDX)
00401C7B > 5B           POP EBX
00401C7C . C3           RETN
```

An access violation happens at 0x00401C76 trying to access an invalid pointer (0x41414141) in our case. Let's see the registers :

```
EAX 41414141
ECX 41414141
EDX 41414141
EBX 41414245
ESP 0018F4F0 ASCII "EBAA4$@"
EBP 0018F514
ESI 01FB2460
EDI 01FB2564 ASCII "AAAAAAAAAAAAAAAAAAAAAAAA
```

Basically the FreeList used in this software is a circular doubly linked lists similar to Windows's . The circular doubly linked list head is in the .bss section at address 0x00560478 and its flink and blink pointers are poiting to head when the custom heap manager is initialized by the software.

I also didn't check the full implementation of the FreeList and the free/allocate operations in this software to see if they're similar to Windows's (bitmap , block coalescing ...etc).
It's crucial also to know that in our case , the block is being unlinked from the FreeList because the manager had a 'request' to allocate a new block , and it was chosen as best for the allocation.

Let's get back to analysing the crash :
- First I would like to mention that we'll be calling the pointer to the Freelist Entry struct : "**entry**".

Registers at 0x00401C76 :
EAX = entry->Flink
EDX = entry->Blink
[EAX] = entry->Flink->Flink
[EAX+4] = entry->Flink->Blink (Next Block's Previous block)
[EDX] = entry->Blink->Flink (Previous Block's Next block)
[EDX+4] = entry->Blink->Blink

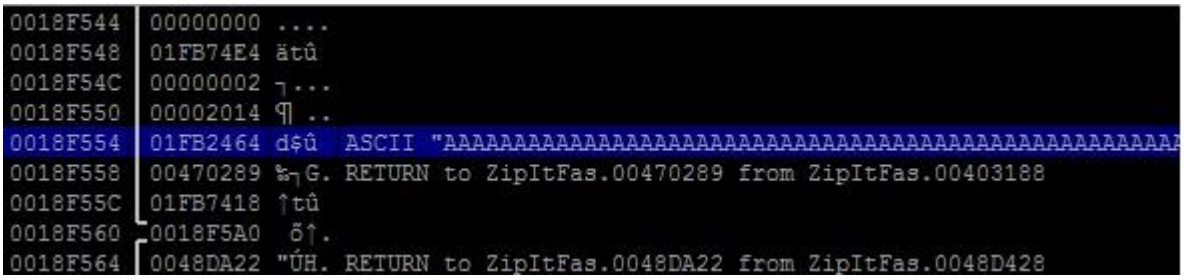
Logically speaking : **Next Block's Previous Block** and **Previous Block's Next Block** are nothing but the current block.
So the 2 instructions that do the block **unlinking** from the FreeList just :
- Set the previous freelist entry's flink to the block entry's flink.
- Set the next freelist entry's blink to the block entry's blink.

By doing so , the block doesn't belong to the freelist anymore and the function simply returns after that. So it'll be easy to guess what's happening here , the software allocates a static 'heap' block to store the name of the file and it would have best to allocate the block based on the filename length from the ZIP header (this could be a fix for the bug , but heap overflows might be found elsewhere , I'll propose a better method to fix ,but not fully, this bug later in this article).

Now , we know that we're writing past our heap block and thus overwriting the custom metadata of the next heap block (flink and blink pointers). So, We'll need to find a reliable way to exploit this bug ,

as the 2 unlinking instructions are the only available to us and we control both EAX and EDX. (if it's not possible in another case you can see if there are other close instructions that might help), you can think of overwriting the return address or the pointer to the structured exception handler as we have a stack that won't be rebased after reboot.
This might be a working solution in another case where your buffer is stored in a **static memory location**.

Under Windows 7 , it's not the case , **VirtualAlloc** allocates a chunk of memory with a different base in each program run. In addition , even if the address was static , the location of the freed block that we overwrite varies so in both cases we'll need to find a **pointer to our buffer**.
The best place to look is in the stack , remember that the software is trying to unlink (allocate) the block that follows the block where we've written the name , so likely all near pointers in the stack (current and previous stack frame) are poiting to the newly allocated block (pointer to metadata) . That's what we don't want because flink and blink pointers that we might set might not be valid opcodes and might cause exceptions , so all we need to do is try to find a pointer to the first character of the name and then figure out how to use this pointer to gain code execution , this pointer might be in previous stack frames.
And here is a pointer poiting to the beginning of our buffer : 3 stack frames away



Remember that 0x01FB2464 will certainly be something else when restarting the program , but the pointer 0x0018F554 is always static , even when restarting the machine.
A use for this pointer would be overwriting the vectored handler node with 0x0018f554 - 0x8 under XP SP1 , but it's not the case here **AT ALL** .

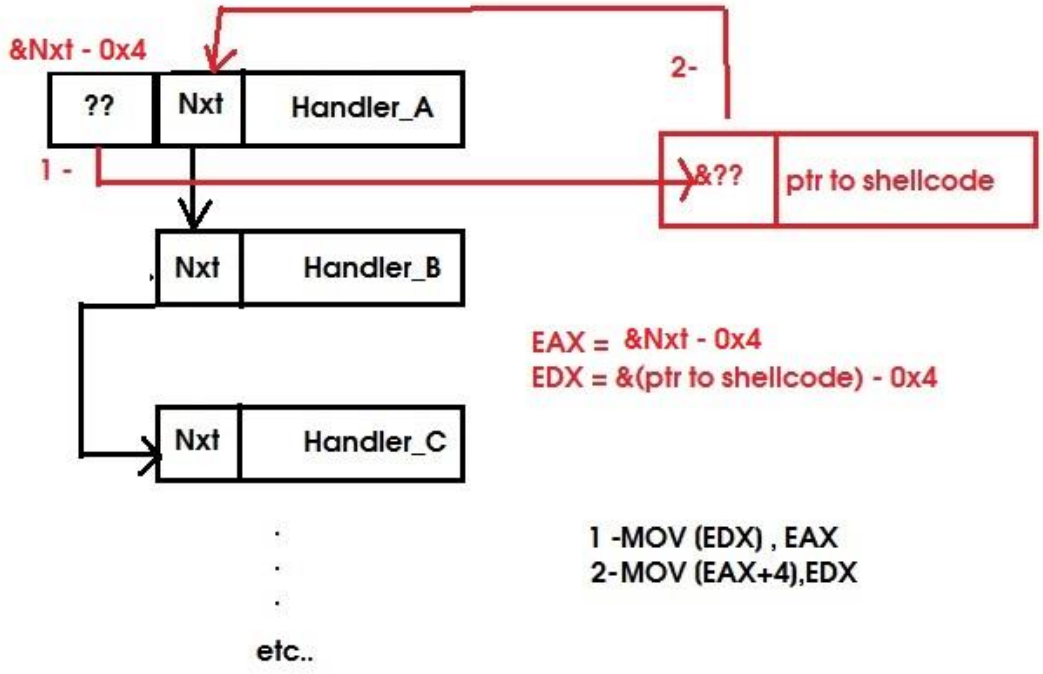
So when I was at this stage , I started thinking and thinking about a way that will help me redirect execution to my shellcode which is for sure at the address pointed by 0x0018F554 , and by using only what's available to me :
- Controlled registers : EAX and EDX.
- Stack pointer to a dynamic buffer pointer.
- 2 unlinking instructions.
- No stack rebase.

Exploiting the vulnerability and gaining code execution:

And Then I thought , why wouldn't I corrupt the SEH chain and create a **Fake** frame ? Because when trying to create a fake an SEH chain there are 3 things that you must know :

- SafeSEH and SEHOP are absent.
- Have a pointer to an exisiting SEH frame.
- Have a pointer to a pointer to the shellcode.

The pointer to the shellcode will be treated as the handler,and the value at ((ptr to ptr to shellcode)-0x4) will be treated as the pointer to the next SEH frame.
Let's illustrate the act of corrupting the chain : (with a silly illustration , sorry)



Let me explain :

we need to achieve our goal by using the 2 instructions :

MOV [EDX],EAX
MOV [EAX+4], EDX

We'll need 2 pointers and we control 2 registers , but which pointer give to which register ? This must not be a random choice because you might overwrite the pointer to the shellcode if you chose EAX as a pointer to your **fake SEH frame**.
So we'll need to do the reverse , but with precaution of overwriting anything critical.
In addition we actually don't care about the value of "next SEH frame" of our fake frame.

So our main goal is to overwrite the "next SEH frame" pointer of an exisiting frame , to do so we need to have a pointer to our fake frame in one of the 2 registers. As [EAX+4] will overwrite the pointer to the buffer if used as a pointer to the fake SEH frame , we will use EDX instead. We must not also overwrite the original handler pointer because it will be first executed to try to handle the exception , if it fails , our fake handler (shellcode) will be invoked then.
So : **EDX = &(pointer to shellcode) - 0x4 = Pointer to Fake "Next SEH frame" element.**
EDX must reside in the next frame field of the original frame which is : [EAX+4].
And **EAX = SEH Frame - 0x4.**

Original Frame after overwite :
Pointer to next SEH : Fake Frame
Exception Handler : Valid Handler

Fake Frame :
Pointer to next SEH : (Original Frame) - 0x4 (we just don't care about this one)
Exception Handler : Pointer to shellcode

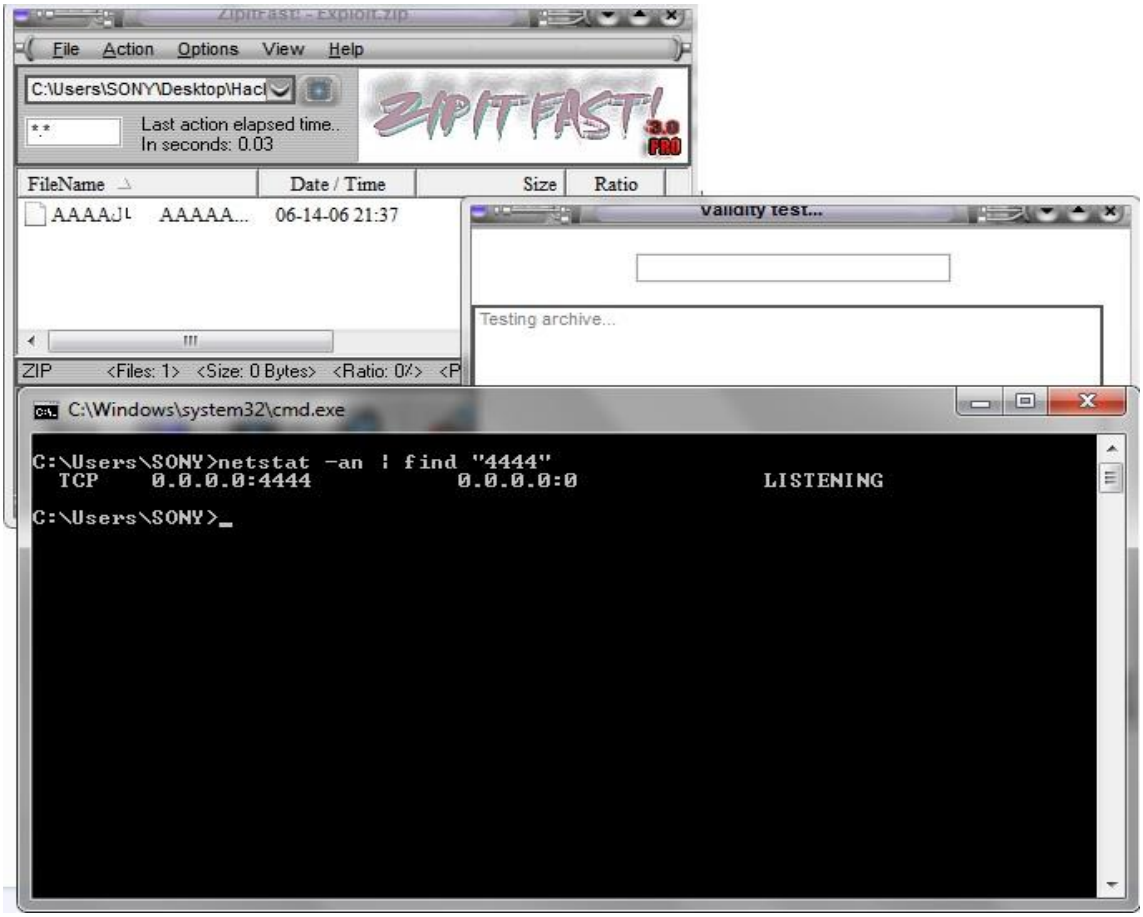
The SEH frame I chose is at : 0x0018F4B4

So : EAX = 0x0018F4B4 - 0x4 = **0x0018F4B0** and EDX =0x0018F554 - 0x4 = **0x0018F550**

Now all we need to do is calculate the length between the 1st character of the name and the flink and blink pointers , and then insert our pointers in the POC.

Inserting the shellcode :
The space between the starting address of the buffer and the heap overwritten metadata is not so large , so it's best to put an **unconditional jump at the starting of our buffer past the overwritten flink and blink pointers** and then put the shellcode just after the pointers. As we can calculate the , this won't cause any problem.
Final exploit here : http://pastebin.com/pKyQJicy

I chose a bind shellcode , which opens a connection to (0.0.0.0:4444). Let's try opening the ZIP file using ZipItFast and then check "netstat -an | find "4444" :



A Fix for this vulnerability ??

The method I stated before which consists on allocating the block based on the filename length from the ZIP headers can be valid only to fix the vulnerability in this case , but what if the attackers were also able to cause an overflow elsewhere in the software ?

The best way to fix the bug is that : when a block is about to be allocated and it's about to be unlinked from the Freelist the first thing that must be done is checking the validity of the doubly linked list , to do so : **safe unlinking** must be performed and which was introduced in later versions of Windows.

Safe unlinking is done the following way :

```
if ( entry->flink->blink != entry->blink->flink || entry->blink->flink != entry){
      //Fail , Freelist corrupted , exit process
}
else {
      //Unlink then return the block to the caller
}
```

Let's see how safe unlinking is done under Windows 7 :

The function is that we'll look at is : **RtlAllocateHeap** exported by ntdll

```
76EC3AB5  . 8D4E 08       LEA ECX,DWORD PTR DS:[ESI+8]              1-
76EC3AB8  . 8B39          MOV EDI,DWORD PTR DS:[ECX]                2-
76EC3ABA  . 897D B8       MOV DWORD PTR SS:[EBP-48],EDI             1 + 2 => Next = Entry->Flink
76EC3ABD  . 8B56 0C       MOV EDX,DWORD PTR DS:[ESI+C]
76EC3AC0  . 8955 98       MOV DWORD PTR SS:[EBP-68],EDX             Prev = Entry->Blink
76EC3AC3  . 8B12          MOV EDX,DWORD PTR DS:[EDX]                EDX = Prev->Flink => EDX = Entry->Blink->Flink
76EC3AC5  . 8B7F 04       MOV EDI,DWORD PTR DS:[EDI+4]              EDI = Next->Blink => EDX = Entry->Flink->Blink
76EC3AC8  . 3BD7          CMP EDX,EDI
76EC3ACA  . 0F85 4B240400 JNZ ntdll.76F05F1B
76EC3AD0  . 3BD1          CMP EDX,ECX
76EC3AD2  . 0F85 43240400 JNZ ntdll.76F05F1B                        if ( Prev->Flink == Next->Blink && Prev->Flink == Entry)
76EC3AD8  . 2958 78       SUB DWORD PTR DS:[EAX+78],EBX                Success;
76EC3ADB  . 8B80 B8000000 MOV EAX,DWORD PTR DS:[EAX+B8]
```

Even if this method looks secure , there is some research published online that provides weaknesses of this technique and how can it be bypassed. I also made sure to implement this technique in my custom heap manager (**Line 86**) , link above.

I hope you enjoyed this paper ... and make sure you check my blog for more cool stuff.

Best regards,

Souhail Hammou.