Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit

I decided to write a quick document about the techniques I used to exploit Internet Explorer 8 on windows 7 with ASLR and DEP enabled.

The exploit consists of two parts.

The first part figures out where a certain .dll file is loaded in the current process followed by step 2 that uses the information gathered in step 1 to trigger an exploit that uses some ret2lib technique to disable DEP for our shellcode and then redirects the program flow to the shellcode.

I will not (and am not allowed to) give out the exact vulnerabilities that I used in the exploit, but I might disclose them someday when Microsoft has them patched. Yes, you read that correctly, them, I used 2 exploits to get the final code execution on W7, but that was partly to speed up the exploit.

Anyways, I'm writing this on the plane to Vancouver without access to the W7 VMs that I tested the exploit on, so I'll keep it vague. Also, I only had MS Word and MS Paint for the text and the images, so don't complain about the quality of the final document ☺

Part one: Evading ASLR

To get the address of a .dll file from the browsers memory I used a heap overflow to overwrite the \x00\x00 bytes at the end of an utf8 string so that I could read the vftable address of the object that I planted next to the string. This is of course the very short version of a research that took a few days to get everything stable, but it's an accurate description.

I found a heap overflow in IE8 that gave me control over both the size of the buffer and the amount of data that got written to the buffer, and some control over what data got written.

I then tried (with success) to set up the IE process heap as displayed on the next image. (Let me remind you that this image was created with ms paint while flying on a plane …)

The bright yellow area is our buffer that we can overflow.
The green area is our string, with the bright green area being the \x00\x00 bytes that mark the end of the string.
The red area is a C++ Object in memory with the vftable address as the first DWord as the bright red area.

This was not as easy as it is to write about it, and it took quite some tricks to get the heap layout the way I wanted it and in a way that the browser would survive a controlled buffer overflow and not crash

before I was able to use the information we got. Anyways, after a nice struggle with Internet Explorer I got the heap the way I wanted to, and quite reliable as well. I then triggered the overflow so it overwrote the \x00\x00 bytes of the string with the data from the buffer. The string does no longer end with \x00\x00. If we then read back the string with a JavaScript function, we suddenly have access to the data in the object since it will continue reading until it comes across a \x00\x00 sequence. So we got the address of the objects vftable in JavaScript. The vftable address tells us the base address of the .dll the object belongs to since it is located at a fixed offset from the base of the module. This information is something we can use to write a DEP evading exploit. If I had full control over the data that got written into the overflowing buffer, it would have been easy to overwrite the vftable address as well, but since I only had limited control (enough to ensure no \x00\x00 in the data), I could not reliable overwrite the vftable address with my data.

Anyways, part 1 end with the disclosure of the base address of a .dll loaded in our IE8 process.

Part 2: Evading DEP after an use-after-free vulnerability

A few months ago I wrote an exploit for IE8 that evaded perm-DEP on Windows XP by using a combination of heap spraying and return-to-lib. Well actually it's more a combination of heap spraying and fake object calls, but I have no idea if there is a name for that, but I doubt I'm the first to use it so I won't try to think of a clever way to name it.

When I was playing around with the Internet Explorer heap for an use-after-free vulnerability I found a while back, I noticed that the allocation of big heap chunks is rather predictable. Not the exact location, but the last 2 bytes are always the same. The blocks will eventually (maybe not the first few, but certainly everything above the 100 on XP) be allocated at 0xYYYY0020 Enough to allow us to spray the heap with a specific pattern that gives us a very high probability of knowing the start address of the pattern.

For example the following code will generate a heap filled with the same pattern over and over again. The pattern will start at 0xZZZZZY20 where Z can be anything (reasonable) and Y = 0/4/8/C since I used a 0x200 size pattern.

```
heap = new heapLib.ie(0x20000);

var heapspray = unescape("%u4141%u4242");

while(heapspray.length < 0x200) heapspray += unescape("%u4444");

var heapblock = heapspray;
while(heapblock.length < 0x40000) heapblock += heapblock;
finalspray = heapblock.substring(2, 0x40000 - 0x21);

for(var i = 0; i < 500; i++) {
   heap.alloc(finalspray);
}
```

This will result in allocations that look a bit like this:

Heap alloc size(0x7ffc0) allocated at 063d0020
Heap alloc size(0x7ffc0) allocated at 06450020
Heap alloc size(0x7ffc0) allocated at 064d0020
Heap alloc size(0x7ffc0) allocated at 06550020
Heap alloc size(0x7ffc0) allocated at 065d0020
Heap alloc size(0x7ffc0) allocated at 06650020
Heap alloc size(0x7ffc0) allocated at 066d0020

As you can see you get a rather contiguous block of filled memory. The best place to find the start of your pattern would be something like: 0x06442020

The pattern used in the above sample code in is quite useless of course, but the pattern you want depends in the vulnerability you're exploiting.

As you can see I used heaplib created by Alexander Sotirov as an easy way to allocate the strings.

This predictable behavior also works on Windows 7 although I needed to spray a bit more then on Windows XP. For XP I used 500 as a spray size, and then used 0x0a042020 as my start address, for Windows 7 I sprayed 900 times and used 0x16402020 as my start address.

The result is a heap where (if sprayed enough) we can predict the memory layout at an address that we choose. For XP I used 0x0a042020 (don't ask me why …). If we are not extremely unlucky that address should be the exact starting point of one of our heap patterns.

I'll explain next how to 'reliable' exploit an use-after-free when it's an easy one. The ones I usually find are the easy types, and with easy I mean: the freeing of the object happens on a different line of JS Code then the using. So we have plenty of time to refill the objects freed address with data that we like.

Say we have the following lines of JavaScript that trigger our hypothetical use-after-free situation:

```
var MyObject = new Object();            <Allocates the object>
var MyObjRef = MyObject.SomeProperty;   <Gets a reference to the object>
MyObject.CleanUp();                     <Frees the object without taking care of 'MyObjRef '>
alert(MyObjectRef.parent);              <Accesses the no longer existing object>
```

Now we want to fill the space left by our freed object with useful data before we call the lines that access the object again. Most times a use-after-free vulnerability will try to do something with the objects vftable. The vftable address is the first DWord of the objects allocated memory, so we need some type of heap spray that can reliable fill the freed memory with user controlled data at the first DWord.

The good thing is that Internet Explorer uses a heap that keeps track of recently freed memory so it can reuse them if a request for an allocation with approx the same size comes in again. If I'm correct it is called the LFHeap (Low Fragmentation I think) but what's in a name. This means that if we know the size of the freed object and we then allocate our own data with the exact same size, it should fill up the recently freed memory nicely.

Getting to know the size of the freed object is not that hard, just use breakpoints on heap alloc en heap free calls in ntdll.
Then we have to allocate the correct memory size again. For this I usually add some 'className' properties to an array of div elements I created earlier. The nice thing about that is that the className properties get allocated as strings with any size you wish, and without any heap overhead, so the first DWord of the allocation is user controlled data. The only drawback is the fact that you can't use \x00\0x00 in your data, but I haven't found that a problem yet.

So what we need to do:

1) Create some array to hold our div elements (var DivArray = new Array();)
2) Fill an array with say 50 objects. (DivArray.push(document.createElement('div');)
3) Run the JavaScript code until we free the object.
4) Add some classNames to the divs: (DivArray[i].className = unescape("%u4141%u4141……
5) Re-use the freed object.

What will happen next is most likely something like this:
move eax, [ecx] ecx = our object memory.
call [eax+0xYY] eax now holds 0x41414141

But how do we turn this into DEP evasion? Simple: We control %eax, we also know the layout of the heap on certain locations. What we do is: set %eax to the start of our heap pattern. Now let's assume that the virtual function called is located at 0x34. Given the fact that we know where our pattern start and we know where in our pattern we try to read our function call, we completely control where IE will call to. We can unfortunately not call directly into our heap spray. What we can do however is use already existing MS code to call 'VirtualProtect' and change the memory protect settings for our shellcode from READWRITE into READWRITEEXECUTE.

For example, let's assume we find a code sequence that looks a bit like this:

```
0x6ff02348 : mov ecx, eax
             call [eax+10]
```

and another that goes like:

```
0x6FF01234 : push [eax+70]
             push [eax+60]
             push [eax+50]
             push [eax+40]
             push [eax+34]
             push [eax+20]
             call [ecx+14]
             …..
             …..
             retn
```

if we then set our heap pattern so that

1) Pattern Start + 0x34 (The first call from the freed memory) points to (0x6FF02348)
2) Pattern Start + 0x10 points to the second address location (0x6FF01234)
3) Pattern Start + 0x14 points to VirtualProtect in kernel32 (0x7c801ad4)

4) Pattern Start + 0x20 points to our Pattern Start
5) Pattern Start + 0x30 = 0x200 (our pattern size)
6) Pattern Start + 0x40 = 0x40 (If I'm correct this is READWRITEEXCUTE)
7) Pattern Start + 0x50 = some address in our heap that we don't need

As you can see this will eventually call VirtualProtect and change the memory status into executable. And the funny thing is that we jump in the middle of the function and just push a lot of stuff on the stack. This will screw up the stack, and if the code hit the retn we will again control the eip. This will work great on Windows XP where we know the exact address of VirtualProtect. On Windows 7 we need to be just a little bit more creative.

How do we call VirtualProtect without knowing the exact location of that function since kernel32.dll gets randomized as well you might wonder? Well, a lot of .dll files in Internet Explorer contain the ATL library. The ATL library uses the VirtualProtect function somewhere. This means that the exact location of the VirtualProtect function in located in the import section of the .dll of which we know the location. Say we know the .dll is loaded at 0x6fff0000, we then know that (for example) 0x6fff1288 has the correct address of VirtualProtect. All we need now is to set %eax to 0x6fff1280 before 'call [eax+8] ' for example. This again can be done with creative code reuse and setting our pattern. When I write exploits using this technique I usually set my pattern to contain a steadily climbing range of numbers like:

```
var pattern = unescape("%u0000%u0001%u0002%u0003%u0004%u0005%u0006……..")
```

This way it is very easy to spot the exact location in your string that you need to edit next.

Basically all we do is string together some pieces of already existing code while using either call [] or jmp to jump to the next lines of assembly that we need until we are all set up for the call to VirtualProtect. Then we line up for the VirtualProtect call, give it the right parameters, and now our heap spray suddenly has EXECUTE rights. If the call to VirtualProtect pushed more parameters than expected, the return stack will be screwed and we end up where we want.

I just read most of it back and Agree that it's a bit of a lousy paper, skipping certain concepts and assuming prior knowledge, continuously switching from 'I' to 'we', but hey, you read it so far so maybe you liked it anyways ☺

Peter Vreugdenhil