Engineering Heap Overflow Exploits with JavaScript

Mark Daniel
Jake Honoroff
Charlie Miller

Independent Security Evaluators
www.securityevaluators.com

September 8, 2008

A version of this paper first appeared in the Proceedings of the USENIX Workshop on Offensive Technologies 2008 (WOOT '08).

Abstract

This paper presents a new technique for exploiting heap overflows in JavaScript interpreters. Briefly, given a heap overflow, JavaScript commands can be used to insure that a function pointer is reliably present for smashing, just after the overflown buffer. A case study serves to highlight the technique: the Safari exploit that the authors used to win the 2008 CanSecWest Pwn2Own contest.

1 Introduction

Many buffer and integer overflow vulnerabilities allow for a somewhat arbitrary set of values to be written at a relative offset to a pointer on the heap. Unfortunately for the attacker, often the data following the pointer is unpredictable, making exploitation difficult and unreliable. The most ideal heap overflow, in terms of full attacker control over the quantity and values of overflow bytes, can be virtually unexploitable if nothing interesting and predictable is waiting to be overwritten.

Thanks to safe unlinking protections, the heap metadata structures are often no longer a viable target for overflows. Currently, application specific data is usually needed as an overflow target, where normal program flow results in the calling of a function pointer that has been overwritten with an attacker supplied shellcode address. However, such exploits are in no way guaranteed to be reliable. It must be the case that pointers yet to be accessed are sitting on the heap after the overflown buffer, and no other critical data or unmapped memory lies in between, the smashing of which would result in a premature crash. Such ideal circumstances can certainly be rare for an arbitrary application vulnerability.

However, given access to a client-side scripting language such as JavaScript, an attacker may be able to create these ideal circumstances for vulnerabilities in applications like web browsers. In [2], Sotirov describes how to use JavaScript allocations in Internet Explorer to allow for attacker control over the target heap. In this paper we describe a new technique, inspired by his *Heap Feng Shui*, that can be used to reliably position function pointers for later smashing with a heap overflow.

This paper contains a description of the technique followed by an account of its application to a WebKit vulnerability discovered by the authors and used to win the 2008 CanSecWest Pwn2Own contest.

2 Technique

2.1 Context

Broadly, this technique can be used to develop client side exploits against web browsers that use JavaScript. In this context, the attacker crafts a web page containing (among other things) JavaScript commands, and induces the victim to browse the page. Using particular JavaScript commands, the attacker influences the state of the heap in the victim's browser process to arrange for a successful attack.

More specifically, the technique is applicable when one has a heap overflow in hand. In what follows, we refer to the buffer that is overflown as the *vulnerable buffer*. This heap overflow must have the property that both allocation of the vulnerable buffer and the overflow itself must be triggerable within the JavaScript interpreter. In particular, this technique will not apply in situations where the vulnerable buffer has already been allocated before the JavaScript interpreter has been instantiated.

We also assume that shellcode is available and a mechanism for loading it into memory has already been found. This is trivial with JavaScript – just load it into a big string.

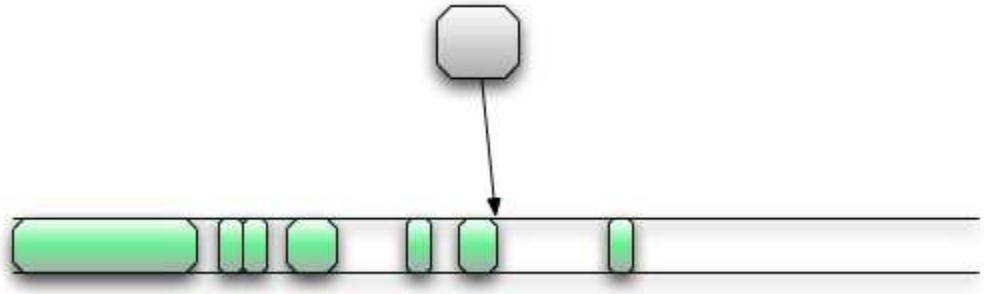

Figure 1: Fragmented heap.

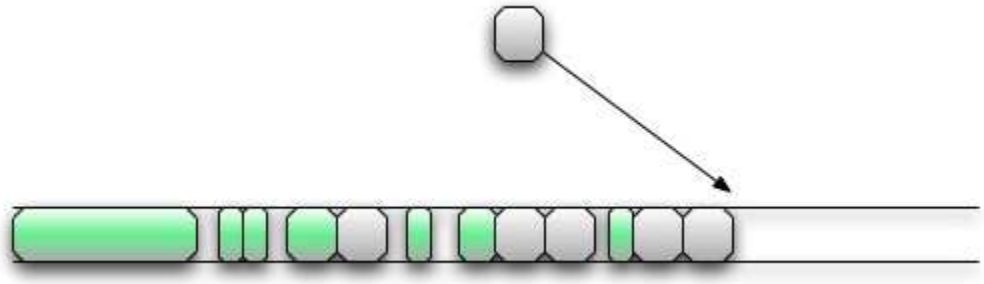

Figure 2: Defragmented heap. Future allocations end up adjacent.

2.2 Overview

Keep in mind that the goal is to control a buffer in the heap immediately following the vulnerable buffer. We accomplish this by arranging the heap so that all holes in it that are big enough to hold the vulnerable buffer are surrounded by buffers that we control.

The technique consists of five steps.

1. Defragment the heap.
2. Make holes in the heap.
3. Prepare the blocks around the holes.
4. Trigger allocation and overflow.
5. Trigger the jump to shellcode.

These steps are described in more detail in the rest of this section.

2.3 Defragment

The state of a process's heap depends on the history of allocations and deallocations that have occurred during the process's lifetime. Consequently, the state of the heap for a long running multithreaded process (e.g. a web browser) is unpredictable. Generally, such a heap is *fragmented* in that it has many holes of free memory. The presence of varying sized holes of free memory means that the addresses of successive allocations of similarly sized buffers are unlikely to have any reliable relationship. Figure 1 shows how an allocation in a fragmented heap might look. Since it is impossible to predict where these holes might occur in a heap that has been in use for some time, it is impossible to predict where the next allocation will occur.

However, if we have some control over the target application, we may be able to force the application to make many allocations of any given size. In particular, we can make many allocations that will essentially fill in all the holes. Once the holes are filled up, i.e. the heap is defragmented, allocations of similar size will, in general, be predictably close to each other, as in Figure 2.

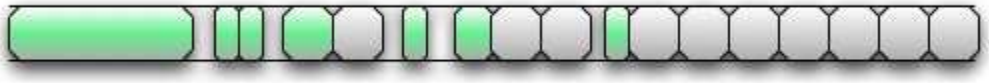

Figure 3: Defragmented heap with many allocations. We see a long line of same-sized buffers that we control.

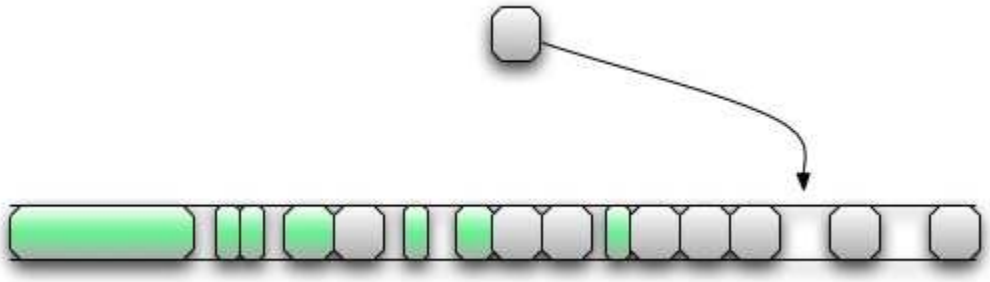

Figure 4: Controlled heap with every other buffer freed. The allocation of the vulnerable buffer ends up in one of the holes.

We emphasize that defragmentation is always with respect to a particular buffer size. In preparation for the exploit, we must defragment the heap with respect to the size of the vulnerable buffer. The size of this buffer will vary, but should be known. As shown in [2], arranging for defragmentation is simple in JavaScript. For example,

```
var bigdummy = new Array(1000);
for(i=0; i<1000; i++){
   bigdummy[i] = new Array(size);
}
```

In the code snippet above, each call to `new Array(size)` results in an allocation of `4*size+8` bytes on the heap.[1] This allocation corresponds to an `ArrayStorage` object consisting of an eight byte header followed by an array of `size` pointers. Initially, the entire allocation is zeroed out. A value of `size` should be chosen so that the resulting allocations are as close as in size as possible, and no smaller, than the size of the vulnerable buffer. The value of `1000` used above was determined empirically.

2.4 Make Holes

Remember, our goal is to arrange for control of the buffer that follows the vulnerable buffer. Assuming defragmentation worked, we should have a number of contiguous buffers at the end of the heap, all roughly the same size as the (yet to be allocated) vulnerable buffer (Figure 3). The next step is to free *every other* of these contiguous buffers, leaving alternating buffers and holes, all matching the size of the vulnerable buffer. The first step in achieving this is

```
for(i=900; i<1000; i+=2){
   delete(bigdummy[i]);
}
```

The lower limit in the `for` loop is based on the assumption that after `900` allocations in the defragment stage, we have reached the point where all subsequent allocations are occurring contiguously at the end of the heap.

Unfortunately, at this point, we have a problem. Simply deleting an object in JavaScript does not immediately result in the object's space on the heap being freed. Space on the heap is not relinquished until garbage collection occurs. Internet Explorer provides a `CollectGarbage()` method that immediately triggers garbage collection [2], but other implementations do not. In particular, WebKit does not. Accordingly, we digress with a discussion of garbage collection in WebKit.

Our inspection of the WebKit source code turned up three main events that can trigger garbage collection. To understand these events, one must be familiar with the way WebKit's JavaScript manages objects.

The implementation maintains two structures, `primaryHeap` and `numberHeap`, each of which is essentially an array of pointers to `CollectorBlock` objects. A `CollectorBlock` is a fixed–sized array of `Cell`s and each `Cell` can hold a `JSCell` object (or derivative). Every JavaScript object occupies a `Cell` in one of these heaps. Large objects (such as arrays and strings) occupy additional memory on the system heap. We refer to this additional system heap memory as *associated storage*.

Each `CollectorBlock` maintains a linked list of those `Cell`s that are free. When an allocation is requested, and no free `Cell`s are available in any existing `CollectorBlock`s, a new `CollectorBlock` is allocated.

All JavaScript objects deriving from `JSObject` are allocated on the `primaryHeap`. The `numberHeap` is reserved for `NumberImp` objects. Note that the latter do not correspond to JavaScript `Number` objects, but instead are typically short lived number objects corresponding to intermediate arithmetic computations.

When garbage collection is triggered, both heaps are examined for objects with zero reference counts, and such objects (and their associated storage if any) are freed.

We now return to the three events we found that trigger garbage collection. They are

1. Expiration of a dedicated garbage collection timer.

[1] Clearly, details such as this depend on the particular implementation of JavaScript. We used WebKit release 31114 for this investigation.

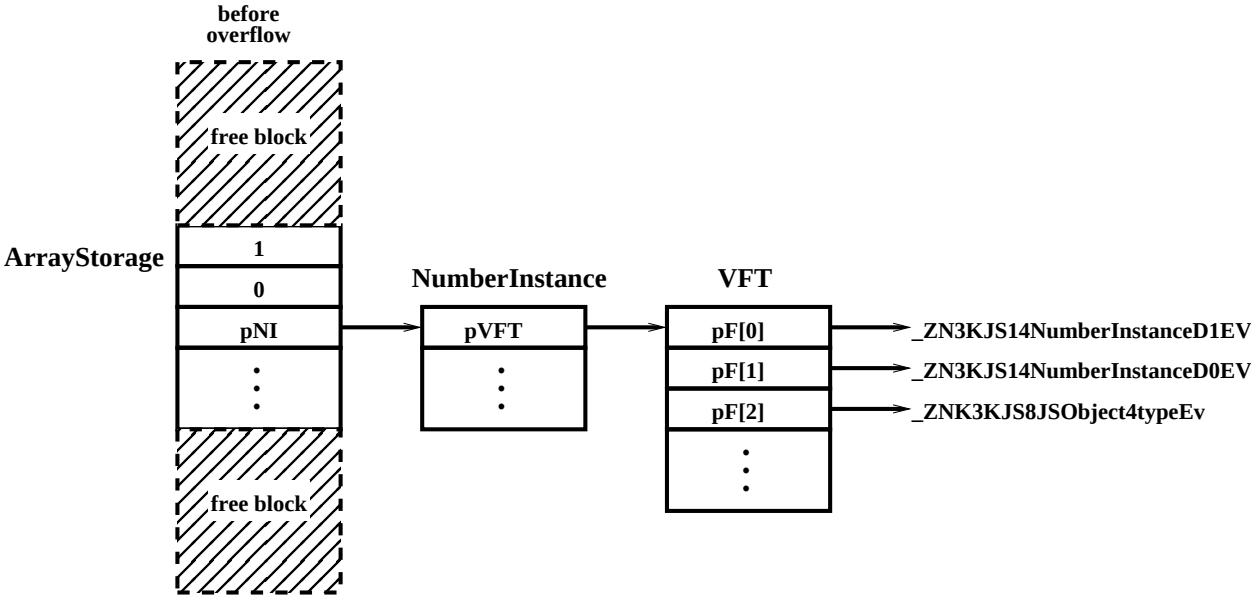

Figure 5: Details of an attacker controlled block just before the overflow is triggered.

2. An allocation request that occurs when all of the particular heap's `CollectorBlock`s are full.

3. Allocation of an object with sufficiently large associated storage (or a number of such objects).

The first of these events is not very useful, because JavaScript has no `sleep` routine, and any kind of sizeable delay in the script runs the risk of causing the "Slow Script" dialog to appear.

The last of these events depends on the number of objects in the `primaryHeap` and the sizes of their associated storage. Experiments suggest that the state of the `primaryHeap` varies greatly with the number of web pages open in the browser and the degree to which these pages use JavaScript. Consequently, triggering reliable garbage collection with this event is difficult.

On the other hand, the `numberHeap` appears to be relatively insensitive to these variations. Heuristically, the `numberHeap` maintains only one allocated `CollectorBlock`, even during significant web browsing of pages with JavaScript. Since the `numberHeap` `CollectorBlock` has 4062 `Cell`s, JavaScript code that allocates at least this many `NumberImp`s should trigger garbage collection. As an example, manipulation of `double`s is one operation that results in a `NumberImp` allocation from the `numberHeap`, so the following JavaScript code can be used for garbage collection triggering:

```
for(i=0; i<4100; i++){
  a = .5;
}
```

With the completion of this code, the heap should appear as in Figure 4, and we are ready for the next step.

2.5 Prepare the Blocks

This step is straightforward. We use the following JavaScript.

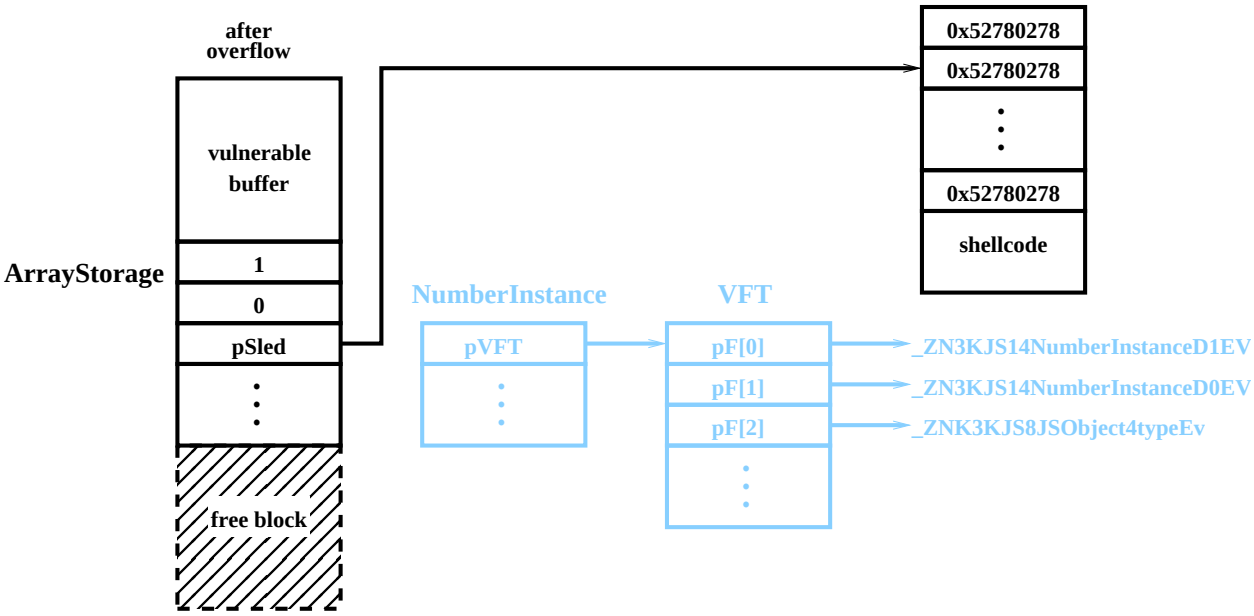

Figure 6: Details of an attacker controlled block just after the overflow is triggered.

```
for(i=901; i<1000; i+=2){
   bigdummy[i][0] = new Number(i);
}
```

The code `bigdummy[i][0] = new Number(i)`, creates a new `NumberInstance` object, and stores a pointer to this object in the `ArrayStorage` object corresponding to `bigdummy[i]`. Figure 5 depicts a portion of the heap after this JavaScript runs.

2.6 Trigger Allocation and Overflow

Now it is time to allocate the vulnerable buffer. If the previous steps have gone as expected, the allocation for the vulnerable buffer will end up in one of the holes that we created, and we are ready for the overflow. The object of the overflow is to overwrite the *pNI* pointer in the `ArrayStorage` object that follows the vulnerable buffer. The new value should be an address in the sled for the shellcode. Details about the sled will be discussed below, but for now, note that a typical NOP sled is not appropriate here. After allocation and overflow, the heap should look as depicted in Figure 6.

2.7 Trigger Jump to Shellcode

The jump to shellcode is executed by simply interacting with the `Number` objects created during preparation of the blocks above. More specifically, we need force a call to a virtual method of the underlying `NumberInstance` object in the JavaScript implementation. For the blocks that were not overwritten, execution is transferred to **((*pNI) + 4*k)* where *k* is the index of the method in the virtual function table

that is invoked. For the block that immediately follows the vulnerable buffer, execution is transferred to *((*pSled)+4*k)*. This double dereference of *pSled* is mildly irritating, but the case study that follows shows a simple way of dealing with it.

The following JavaScript forces a virtual function call for each `NumberInstance` object, and thereby triggers execution of the shellcode.

```
for(i=901; i<1000; i+=2){
   document.write(bigdummy[i][0] + "<br />");
}
```

3 Case Study

Our research into reliable JavaScript heap exploitation was motivated by a vulnerability we found in WebKit's PCRE (Perl-Compatible Regular Expression) parsing. It was an integer overflow that allowed for an arbitrary overflow size past a buffer holding a compiled regular expression that could be allocated to be any size up to 65535 bytes. However, the overflow occurred very soon after the buffer was allocated, such that we often ran up against unallocated memory during the overflow. In other instances, we overwrote important data, but what data was there changed between runs and seemed totally unpredictable. The technique described in Section 2 solved these problems for us and allowed for reliable exploitation.

First we needed to defragment the heap for a target size of approximately 4000 bytes. The following debugger output shows how the first several allocations are needed to defragment the heap (note the jumping around of allocation addresses):

```
Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$1 = 0x16278c78

Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$2 = 0x50d000

Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$3 = 0x510000

Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$4 = 0x16155000

Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$5 = 0x1647b000

Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$6 = 0x1650f000
```

```
Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$7 = 0x5ac000
```

By the time we've done almost 1000 allocations, the heap starts to look totally predictable, and all allocations end up adjacent:

```
Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$997 = 0x17164000

Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$998 = 0x17165000

Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$999 = 0x17166000

Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$1000 = 0x17167000

Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$1001 = 0x17168000

Breakpoint 3, 0x95850389 in
KJS::ArrayInstance::ArrayInstance ()
array buffer at$1002 = 0x17169000
```

After freeing up a bunch of holes at the end using the garbage collection technique described in Section 2.4, we see the vulnerable buffer landing in the last hole at `0x17168000`, right before data we control at `0x17169000`.

```
Breakpoint 2, 0x95846748 in
jsRegExpCompile ()
regex buffer at$1004 = 0x17168000
```

So we will now overflow our regex buffer into the `ArrayStorage` data. The bytes of the overflow have to be compiled regular expression bytes. Luckily, the regular expression character class construct allows for virtually arbitrary bytes in compiled form, as it compiles to 33 bytes, the final 32 of which constitute a 256-bit bit array where a bit being set to one means that that character is in the class. Thus we use the character class `[\x00\x59\x5c\x5e]` and arrange for it to land at the beginning of the `ArrayStorage` data, since the first 3 dwords of its compiled bit class are a non-zero dword, a zero dword, and an address in our heap spray, namely:

```
0x00000001 0x00000000 0x52000000
```

Finally, we use a specially crafted heap spray using a large string of the dword `0x52780278` followed by shellcode. We arrange the spray allocation so that this address is in the spray. This is so that it is self-referential, i.e. the dereferencing that occurs down the chain from the object pointer through the VFT stays in the spray. Then, when interpreted as instructions once execution begins, it becomes the conditional jumps:

```
78 02:   js +0x2
78 52:   js +0x52
```

which are an effective NOP no matter what the jump condition value is: if the condition is true, a jump of two is taken to the beginning of the next instruction, and if the condition is false, it is false for the jump of `0x52` as well. This has the nice property of meaning the most significant byte when determining the heap spray address is completely unused in the sledding, as well as being 4-byte aligned (a spray using the unconditional jump opcode `0xeb` is not, and this can be essential in some exploits).

4 Conclusions

The technique described in this paper allowed for reliable exploitation of a buffer overflow that initially had no predictable and interesting data to overwrite. While some attacker control is necessary, such as allocation size, overflow size, and overflow data, this technique should be applicable to other browser vulnerabilities when the attacker has access to JavaScript. We suspect that similar techniques may be applicable given access to other client-side scripting languages.

References

[1] Flake, H. *Third Generation Exploitation*. Blackhat Briefings Windows 2002.

[2] Sotirov, A. *Heap Feng Shui in JavaScript*. Blackhat Europe 2007.