# Analysis of the WebViewFolderIcon ActiveX Integer Overflow (setSlice) CVE-2006-3730



# Table of Contents

# Introduction

This document contains details on CVE-2006-3730, a client-side remote code execution vulnerability in Windows Shell that is exposed by the WebViewFolderIcon ActiveX control. The flaw exists due to an integer overflow inside of a function designed to perform memory management for a heap-based dynamic structure array (DSA). Exploit code, released publicly over two weeks before the Microsoft patch, utilized the vulnerability to write data into unintended areas of the process' heap and ultimately gain control of the program's execution.

Also known as MS-06-057, this vulnerability is responsible for large-scale infection and compromise of Windows systems. Attackers are distributing the exploit using conventional methods such as embedding the code, or hyperlink references to the code, in email messages or web pages. Although the payload is variable, most exploits seen in the wild are designed to download and install additional malware on the target.

The forthcoming analysis serves to extend knowledge on this particular vulnerability and show how it is being exploited.

# Reverse Engineering the Vulnerable Code

## *Overview*

To provide this analysis, it was necessary to reverse engineer the relevant functions within Microsoft's WebView (webvw.dll) and Common Controls Library (comctl32.dll). These DLLs are loaded by Internet Explorer and are both involved in the processing of WebViewFolderIcon ActiveX objects. The pseudo-code presented in this document is not an exact duplicate of the original source, however it is representative of the program's behavior and the variables are named according to their observed usage.

## *Significant Data Components*

To understand the vulnerability and how it is exploited, one must also reconstruct the WebViewFolderIcon class members, DSA elements, and VARIANT structures. This section will present the data components, the relationship between them, and describe how they are used by Internet Explorer.

## *The WebViewFolderIcon Class*

The WebView library, webvw.dll, declares a class by the name of WebViewFolderIcon. Each time a new ActiveX object of this type is created; Internet Explorer allocates 424 bytes on the process heap to store it, and then calls the class' constructor to initialize its member values. Every member of this class is not required to understand the vulnerability, however it should be noted that the class contains a pointer to another class (*AmbientFont*) which contains pointers to virtual functions.

The WebFolderViewIcon class' destructor attempts to lookup and call the virtual functions by first de-referencing the AmbientFont class pointer. So if this value, *pAmbient*, is overwritten to point at an address where user-supplied data could be staged (such as the process heap), an attacker can gain control of program execution when the WebFolderViewIcon object is programmatically deleted or when the variable goes out of scope.

```
class AmbientFont
{
public:
	virtual void ClearClass(AmbientFont *);
	virtual void ClearClassAndObject(AmbientFont *, HGDIOBJ);
};

class CWebFolderViewIcon
{
public:
	CWebFolderViewIcon();
	~CWebFolderViewIcon();
	 int SetSlice(int, tagVARIANT, tagVARIANT, tagVARIANT);
private:
	void _ClearLabel(void);
	void _ClearAmbientFont(void);
	HDSA	HDSA;
	HGDIOBJ	hObject;
	AmbientFont *pAmbient;
};
```

## VARIANT and VARIANTARG Structures

VARIANT and VARIANTARG are 16-byte structures defined in OAIDL.H. Their purpose is to provide interoperability between weak and strong-typed languages. A VARIANT is simply a wrapper around the desired data, with a word-sized value (*vt*) that indicates which type of data it contains. This allows programmers of strong-typed languages, such as C or C++, to send/receive arbitrary data types across function boundaries without generating compile-time errors. Upon evaluating *vt*, the receiving function will know to interpret the VARIANT's value as an integer, string pointer, or other data type.

This is important because the WebViewFolderIcon setSlice() method accepts four parameters, three of which are VARIANT structures. The values for these structures originate from the user-supplied input and if the last VARIANT's data type is neither integer (VT_I4) nor wide-character pointer (VT_BSTR), then setSlice() will return before calling DSA_SetItem() and the vulnerability will not trigger.

## Dynamic Structure Arrays (DSA)

DSAs provide similar functionality to vectors. A DSA can contain items of any size and can be dynamically adjusted to insert or remove items. When a WebViewFolderIcon object is created, the constructor calls DSA_Create() and stores a handle to the DSA object in one of its class members, *HDSA*. Then, when the object's setSlice() method is invoked, a call is made to DSA_SetItem(). This function's purpose is to set the contents of the specified DSA item using values that originate from the user-supplied arguments to setSlice(). One of these arguments is the zero-based index of the item to set. It is this value that can trigger an integer overflow and cause the DSA_SetItem() function to mis-calculate the number of bytes to allocate for the adjustment, and also the memory address to begin writing the item data.

The class below describes the data types and corresponding functionality of each variable associated with a DSA. When the WebViewFolderIcon constructor calls DSA_Create(), it passes 16 and 2 as arguments, which are the desired values of cbItem and cItemGrow, respectively.

```
class DSA
{
public:
       int idxLastItem;      // Last index that the array can hold.
       HLOCAL hMemArrayData; // Handle to heap data.
       int cItems;           // Count of items.
       int cbItem;           // Size, in bytes, of the item.
       int cItemGrow;        // Number of items by which the array should be
                             // incremented, if the DSA needs to be enlarged.
};
typedef DSA *HDSA;
```

## Pseudo-code – The DSA_SetItem() Integer Overflow

The DSA_SetItem() function takes three parameters. The first is a handle to a DSA (*HDSA*) that contains the item to set. The second is a zero-based index of the item (*idxItem*), within the array, that the calling function wishes to set. The third is a pointer to the item (*pItem*) that will replace the specified item in the array. As briefly mentioned above, the calling function, setSlice(), derives the index and item data from the caller and the calling code may be implemented in java script.

The integer overflow occurs during the calculation of cItemsNew, a signed 32-bit integer that is supposed to store the new count of items in the DSA, once the adjustment has been completed. To arrive at this number, the value of cItemGrow (2) is added to the idxItem index, and then the result undergoes a

divide/multiply operation (for rounding). Expectedly, the new count of items is then multiplied by the size of each item (16) to derive the number of bytes required to store the DSA. This becomes the uBytes argument to ReAlloc(). The code then invokes a memmove() operation to set the item's contents at the requested offset.

Now, consider an example using 0x7FFFFFFE as the idxItem index. When cItemGrow is added to this value during the calculation of cItemsNew, it becomes 0x80000000. Multiplied by cbItem, this becomes zero, and ReAlloc() ends up allocating an inadequate number of bytes for the DSA. When memmove() is called, 0x7FFFFFFE is multiplied by cbItem and it becomes 0xFFFFFFE0, or -32 decimal. Instead of writing the item data at the proper location within an adequately-sized DSA, it begins to write the data 32 bytes behind the heap address returned by ReAlloc().

```
int DSA_SetItem(HDSA HDSA, int idxItem, void *pItem) {
	register HLOCAL hMem;
	register int cItemsNew;

	if (idxItem < 0) {
		return(INVALID_INDEX);
	}

	if (idxItem >= HDSA->idxLastItem) {
		if (idxItem + 1 > HDSA->cItems) {
			cItemsNew = ((idxItem + HDSA->cItemGrow) / HDSA->cItemGrow)
			  * HDSA->cItemGrow;

			hMem = ReAlloc(HDSA->hMemArrayData, cItemsNew * HDSA->cbItem);
			if (hMem == NULL) {
				return(0);
			}
			HDSA->hMemArrayData = hMem;
			HDSA->cItems = cItemsNew;
		}
		HDSA->idxLastItem = idxItem + 1;
	}

	memmove((void *)((idxItem * HDSA->cbItem) + (DWORD)HDSA->hMemArrayData),
	  pItem, HDSA->cbItem);

	return(1);
}
```

## *Pseudo-code – WebViewFolderIcon Class Destructor*

When an existing WebViewFolderIcon object is deleted or goes out of scope, its destructor method is called. As shown below, this function subsequently calls _ClearAmbientFont(). If the ActiveX object utilizes ambient fonts, the *pAmbient* class pointer will likely be a non-NULL value. In this case, the code de-references the pointer and attempts to call its ClearClassAndObject() and/or ClearClass() methods. As mentioned before, if *pAmbient* is overwritten to point at a heap address where user-supplied data exists, then the calls to ClearClassAndObject() and ClearClass() can be redirected.

```
void CWebFolderViewIcon::_ClearAmbientFont(void) {
	if (pAmbient == NULL) {
		if (hObject != NULL) {
			DeleteObject(hObject);
		}
	}
	else {
		if (hObject != NULL) {
			pAmbient->ClearClassAndObject(pAmbient, hObject);
		}
		pAmbient->ClearClass(pAmbient);
		pAmbient = NULL;
	}
	hObject = NULL;
	return;
}


CWebFolderViewIcon::~CWebFolderViewIcon(void) {
	_ClearLabel();
	_ClearAmbientFont();
	// Additional code snipped for brevity
}
```

# Triggering the Vulnerability for Code Execution

This section describes the multiple techniques that one may use to exploit the vulnerability for code execution.

## *Overview*

It has been discussed that if 0x7FFFFFFE is supplied as the first argument to setSlice(), then the memmove() operation within DSA_SetItem() will write data 32 bytes behind the DSA base address on the heap. We also have discussed the WebFolderViewIcon class and know that it contains a pointer to another class with virtual function tables. The idea is that if one can ensure that memory allocation for the DSA occurs directly after allocation for the class, then writing 32 bytes behind the DSA will overwrite some of the class members. We have also discussed that the data written is accepted from user-supplied input – the final three parameters to setSlice(), in particular.

## *Writing Backward on the Heap*

The publicly available exploits all write backward on the heap. They litter the heap with repeated blocks of "spraySlide" bytes and shell code in hopes that at least one will land on the heapSprayToAddress location in memory.

```
var heapSprayToAddress = 0x05050505;
var infernalis_ = unescape("%u9090%u9090$war_code");
var heapBlockSize = 0x400000;
var payLoadSize = infernalis_.length * 2;
var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
var spraySlide = unescape("%u0505%u0505");
spraySlide = getSpraySlide(spraySlide,spraySlideSize);
heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;
memory = new Array();

for (i=0; i<heapBlocks; i++)
{
 memory[i] = spraySlide + infernalis_;
}
```

At this point, the exploit enters another loop that creates 128 WebViewFolderIcon objects, each which allocates memory for a DSA and then writes the three instances of heapSprayToAddress parameters 32 bytes behind the DSA's base address on the heap. This is basically a brute-force attempt to ensure that at least one of the memmove() calls overwrites the preceding object's members, in particular the pointer to *AmbientFont* and its virtual function pointers.

```
for ( i = 0 ; i < 128 ; i++)
{
 try{
  var tar = new ActiveXObject('WebViewFolderIcon.WebViewFolderIcon.1');
  tar.setSlice(0x7ffffffe, 0x05050505, 0x05050505,0x05050505 );
 }catch(e){}
}
```

During destruction of the objects, which happens when the variables go out of scope, the code attempts to call the AmbientFont object's virtual functions. If the brute-force is successful, the processor will be redirected to 0x05050505 and hit the spraySlide and eventually the shell code.

## *Writing Forward on the Heap*

It is also possible to surround a particular WebViewFolderIcon object and corresponding DSA with other objects of the same type, then write forward on the heap. The strategy for this technique is to create heap litter similar to the previous exploitation method so that a known address is allocated on the heap. After this has been accomplished, several hundred blocks sized 0x1A8 bytes should be allocated in order to ensure contiguous memory allocation of similarly-sized objects. Next, if the first call to setSlice() for a WebViewFolderIcon object uses an index such as 0x70000018, then the maximum index of the DSA will be set to 0x7000001A, without causing memory corruption. The attacker would then allocate a few hundred new WebViewFolderIcon objects.

Since the size of the memory allocated for the DSA (0x1A0 bytes) and the size of the WebViewFolderIcon object (0x1A8) are similar, and the previous steps ensure contiguous allocation of similarly sized blocks then the attacker can ensure that a WebViewFolderIcon object is placed right after the DSA memory. Now, given that the attacker can write at an arbitrary index into the DSA, regardless of whether the memory buffer is appropriately sized, the attacker can write forward from the DSA memory area and into a WebViewFolderIcon object's members. Similar to the previous exploitation method, the attacker would try to overwrite a virtual function table within a WebViewFolderIcon class, and then cause it to get de-allocated, thus redirecting execution to user-supplied data.

# Remediation and Defense

Before a vendor-supplied fix was available, the Microsoft-recommended defense tactics included setting the WebViewFolderIcon ActiveX kill-bit and/or configuring Internet Explorer to prompt before running ActiveX controls. Third party research groups also released patches that helped prevent exploitation by automating the kill-bit method or by patching the vulnerable DLL in memory.

Several signatures for intrusion detection/prevention systems have also been proposed, however each has its weaknesses. For example, a majority (sid 7985 – 7988) detect the WebViewFolderIcon CLSID or Unicode-encoded CLSID in a packet, however as both the proof-of-concept and public exploits show – the vulnerability can be triggered without including these values. The remaining rules (sid 8419 and 2003110) specify that, at least, the string “WebViewFolderIcon” must exist in the packet for detection. Additional criteria is included to prevent false positives, such as “0x7fffffff”, however once again, neither of these values are required for exploiting the vulnerability.

An attacker can simply create java script code that starts at wildly different values, and modify those values through several steps in order to arrive at the desired value. This method can be used to encode strings and integers such that an IDS would need to implement a java script engine in order to canonicalize the values so that it might come close to detecting whether or not the code is indeed an exploit.

Fortunately, Microsoft issued an update on October 10th that addresses the vulnerability. The following code shows how Microsoft’s patch resolved the integer mis-calculation. This was done by converting the variables to unsigned integers from signed integers and by inserting the following check before the ReAlloc() call:

```
cItemsNew = ((idxItem + HDSA->cItemGrow) / HDSA->cItemGrow) * HDSA->cItemGrow;
if (cItemsNew >= (0x7FFFFFFF / HDSA->cbItem)) { // New
       return(0);
}
hMem = ReAlloc(HDSA->hMemArrayData, cItemsNew * HDSA->cbItem);
```

These modifications are effective in preventing user-supplied values from negatively influencing the DSA_SetItem() memory management routines.