Further Evasion in the Forgotten Corners of MS-XLS

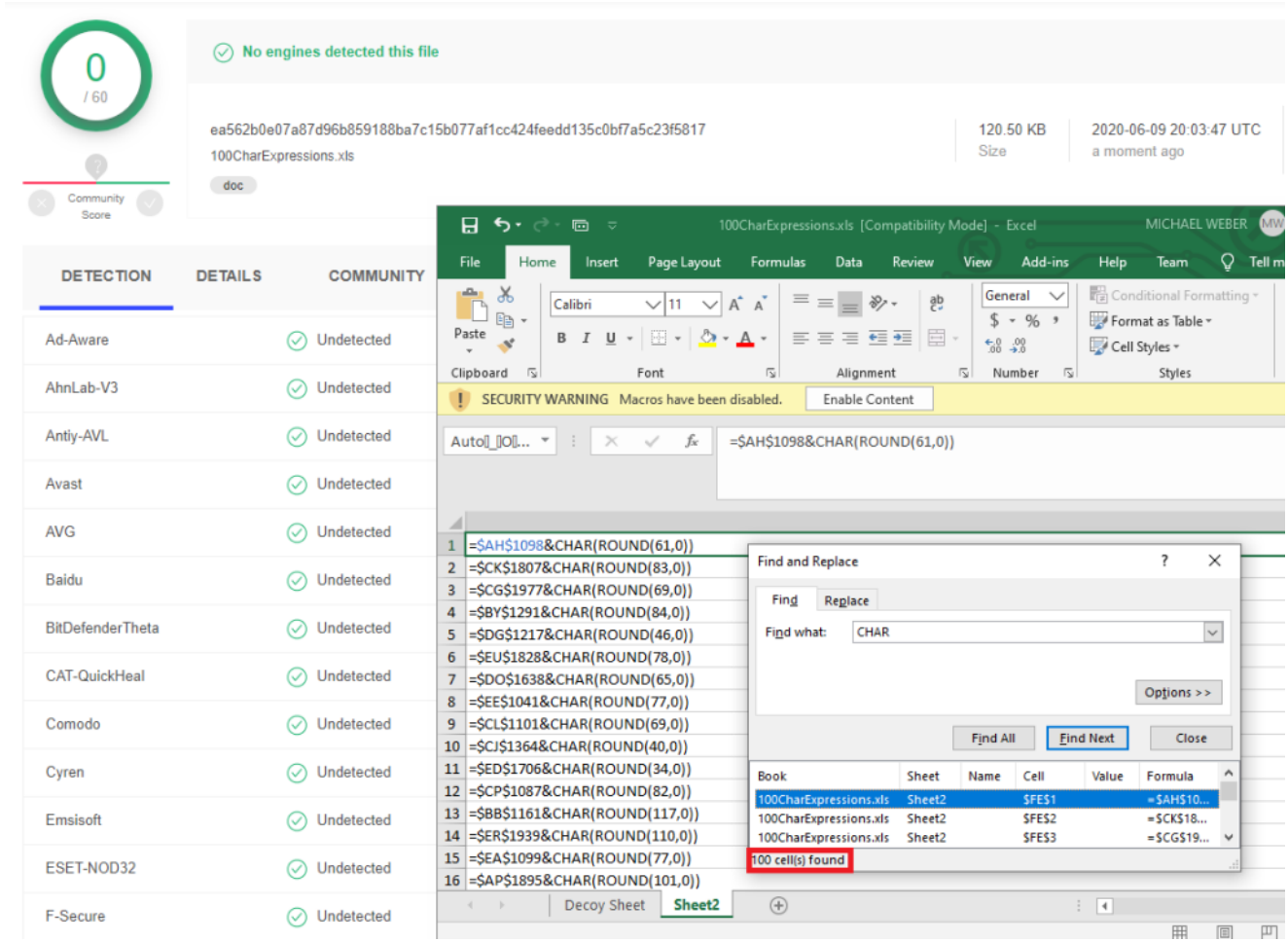
malware.pizza/2020/06/19/further-evasion-in-the-forgotten-corners-of-ms-xls/

June 19, 2020

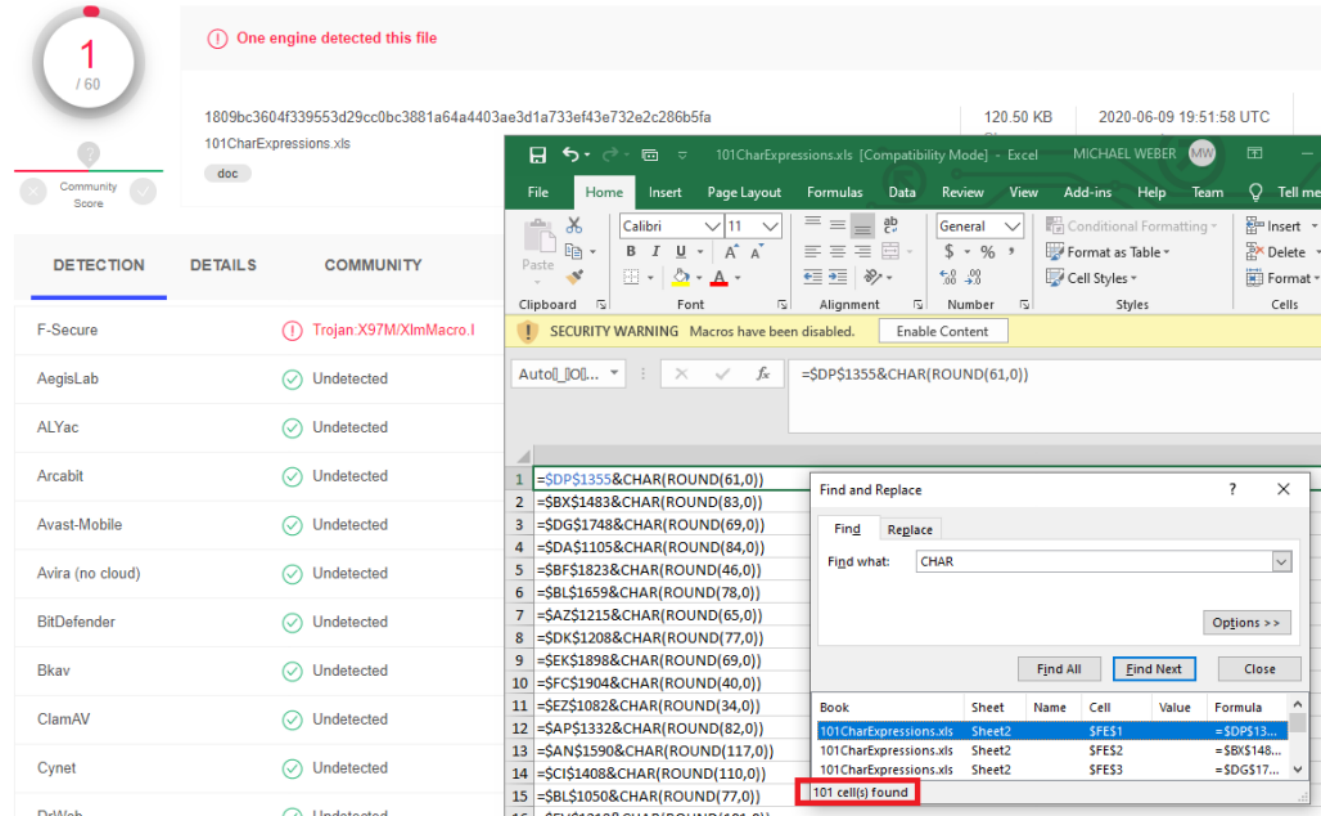
It's been a few weeks since my last discussion[1] of Excel 4.0 macro shenanigans and the space continues to change. LastLine published a great report[2] which summarized the progression of weaponized macros from February through May. The good folks at InQuest have continued[3] identifying[4] malicious[5] macro documents[6]. @DissectMalware's excellent XLMMacroDeobfuscator[7] has massively expanded its range of macro emulation, and FortyNorth Security released EXCELntDonut[8], a tool for converting Donut[9] shellcode into multi-architecture Excel 4.0 macros.

Over the past few weeks I've also started seeing some of the files generated by my tool Macrome[10] begin to trigger detections on VirusTotal[11]. This is exactly the sort of thing I want to see – besides the fact that it implies that AV is getting better signal on this attack vector, it also provides an opportunity to improve my tool and take better guesses about what direction attackers will pivot in the future. I'm a big believer in a @Mattifestation's approach to detection engineering[12] and detection from AV helps move the iterative development of tooling further along.

After realizing that some of my samples were being detected, I took several documents that had been generated during testing and submitted each of them to VirusTotal – only the larger documents appeared to be matching virus signatures. I did a quick binary search of the document sizes between what was detected on VirusTotal and what wasn't and discovered that if a document had greater than 100 **CHAR** invocations, then it was considered malicious.



A “safe” document with exactly 100 **=CHAR()** expressions



A document that has one too many **=CHAR()** expressions

While my generated document had obfuscated the usage of the **CHAR** function, clearly there was a signature that could detect these alternate **CHAR** invocations. For reference, here is @DissectMalware's macro_sheet_obfuscated_char rule[13] that the generated document attempted to avoid:

```
rule macro_sheet_obfuscated_char
{
  meta:
    description = "Finding hidden/very-hidden macros with many CHAR functions"
    Author = "DissectMalware"
    Sample = "0e9ec7a974b87f4c16c842e648dd212f80349eecb4e636087770bc1748206c3b
(Zloader)"
  strings:
    $ole_marker = {D0 CF 11 E0 A1 B1 1A E1}
    $macro_sheet_h1 = {85 00 ?? ?? ?? ?? ?? ?? 01 01}
    $macro_sheet_h2 = {85 00 ?? ?? ?? ?? ?? ?? 02 01}
    $char_func = {06 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? 1E 3D  00 41 6F 00}
  condition:
    $ole_marker at 0 and 1 of ($macro_sheet_h*) and #char_func > 10
}
```

My previous blog post discussed how to break the longer signature for `$char_func`, but it didn't address what to do if the signature for the **CHAR** function were more reliable. In this case the signature was likely only the the three bytes of a **PtgFunc**[14] invocation with the **CHAR Ftab** value[15] (`41 6F 00`) but repeatedly occurring enough times to avoid false positives. This is likely the reason for the "high" minimum count requirement of 101+ instances versus the 11+ in the `macro_sheet_obfuscated_char` rule.

```
Formula[FE110]: CV1511&CHAR(ROUND(65,0))
00000000   6D 00 A0 00 0F 00 03 00  00 00 00 00 FF FF 21 00   m· ·········ÿÿ!·
00000010   00 00 00 00 18 00 44 E6  05 63 00 1F 00 00 00 00   ·····Dæ·c·····
00000020   00 40 50 40 1E 00 00 41  1B 00 41 6F 00 08         ·@P@···A··Ao··
```

An obfuscated invocation of CHAR(65) that triggered results on VirusTotal after 101+ instances were used

One "quick" hack to bypass this signature is to abuse the fact that **PtgFuncVar**[16] can be used instead of **PtgFunc** to invoke the **CHAR** function (`42 01 6F 00`). **PtgFuncVar** is largely identical to **PtgFunc** except for the fact that **PtgFuncVar** must also be provided with the number of arguments being passed into the called function. While **PtgFunc** is only used to call functions with a fixed number of arguments, there is nothing that stops us from invoking **PtgFuncVar** and providing the correct argument count. **PtgFunc(CHAR)** is identical to **PtgFuncVar(1,CHAR)**.

```
Formula[FE108]: CHAR(65)
00000000   6B 00 A0 00 0F 00 03 00  00 00 00 00 FF FF 21 00   k· ·········ÿÿ!·
00000010   00 00 00 00 07 00 1E 41  00 42 01 6F 00           ·······A·B·o·
```

Hex dump of a **FORMULA**[17]BIFF8 record using the alternate **PtgFuncVar(1,CHAR)** invocation

This is a nice signature evasion trick, but it ultimately is vulnerable to the same method of detection, just with a slightly different byte signature. Fundamentally, many tricks that macro sheets rely on in order to deobfuscate themselves will rely on invoking a handful of functions repeatedly. Large macro payloads can require invoking some form of **CHAR** and **FORMULA** hundreds of times – what will adversaries do once there are better signatures put into place for detecting suspiciously repeated usages of these functions?

Re-Enter the Subroutine

In normal programming, when we constantly call the same code over and over again, we write a function. Even in VBA macros, the idea of subroutines exist to allow for simple code-reuse. While the Excel 4.0 Macro Functions Reference[18] mentions the idea of Excel 4.0 macro subroutines several times – it never actually details how these can be created.

In practice, Excel 4.0 macro subroutines are really just a sequence of **RUN** and **RETURN** functions. A subroutine is invoked by calling the **RUN** function with an argument referencing the start cell of the sub-macro. Execution then starts at that cell and continues down the column until a **RETURN** function is invoked. The argument passed to **RETURN** is what the return value of the function will be. For example, if we wanted to create a subroutine that would eventually return the string “Hello World”, it would look something like this:

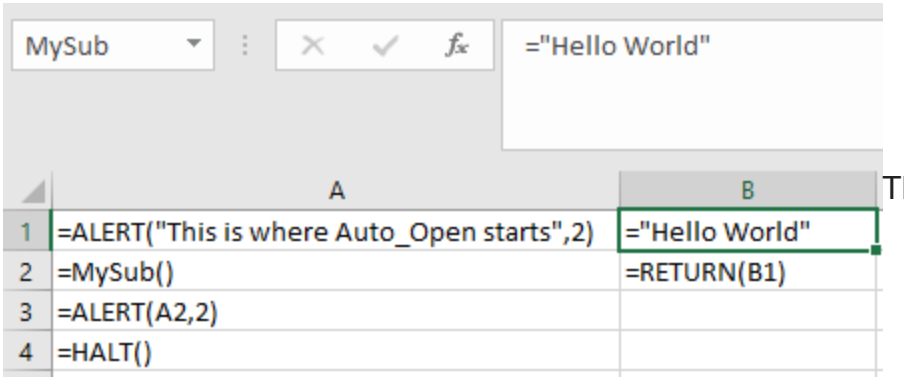
	A	B
1	=ALERT("This is where Auto_Open starts",2)	="Hello World"
2	=RUN(B1)	=RETURN(B1)
3	=ALERT(A2,2)	
4	=HALT()	

A simple example of an Excel 4.0 macro subroutine – it will eventually pop up an alert saying “Hello World”

Excel actually even aliases the **RUN** command by letting users specify a cell reference or cell name and invoke it directly by appending **()** to the invocation as seen below:

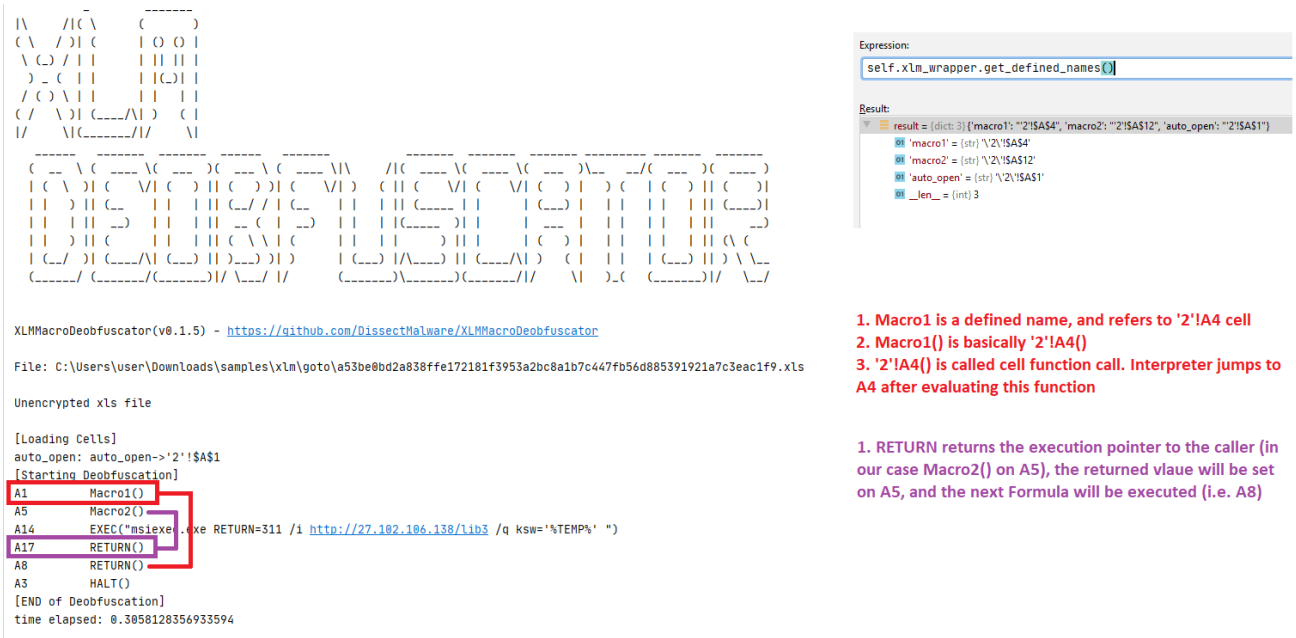
	A	B
1	=ALERT("This is where Auto_Open starts",2)	="Hello World"
2	=B1()	=RETURN(B1)
3	=ALERT(A2,2)	
4	=HALT()	

This is functionally identical to the previous Macro sheet



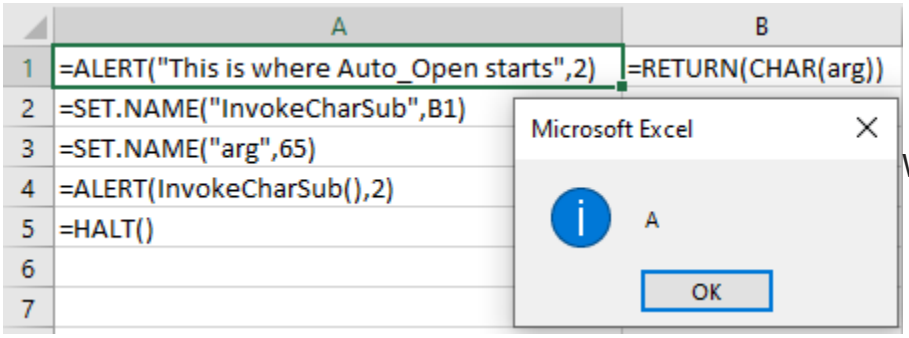
This is also the same, except B1 has been named MySub

It's not a very common way to see macros used right now, but malware authors are clearly already aware of this[19] as can be seen from a sample shared by @JohnLaTwc and analyzed by @DissectMalware:



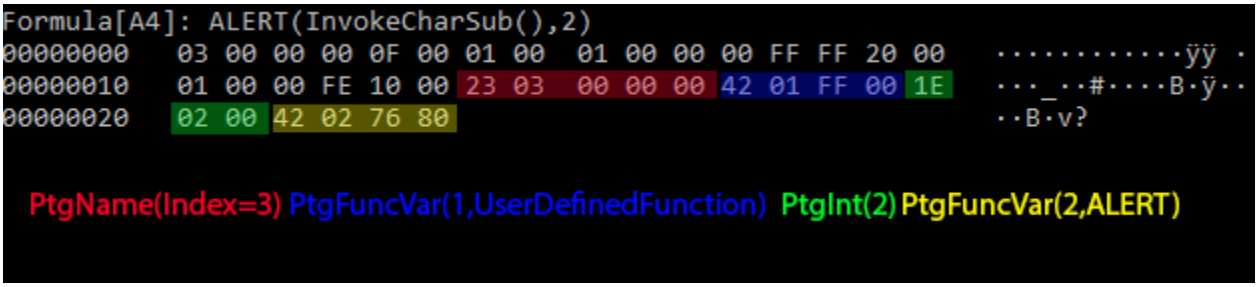
Example behavior from a maldoc submitted to VirusTotal in March 2019[20] (Image from @DissectMalware)

While using subroutines in this way might be slightly helpful for slowing analysis of a document, it's really only dipping its toes into the potential of "proper" subroutine usage in a maldoc. For example, what if instead of having the byte sequence `41 6F 00` every time we invoked **CHAR**, we moved the **CHAR** expression into a subroutine and just invoked the subroutine repeatedly? The predictable function invocation would only appear once, and it would be much harder to claim that EVERY usage of **CHAR** is malicious. Even Windows Defender's aggressive blocking of **=CHAR(#)** invocations requires other conditions beyond matching three bytes. Here's an example of what replacing the **CHAR** expression with a subroutine looks like:

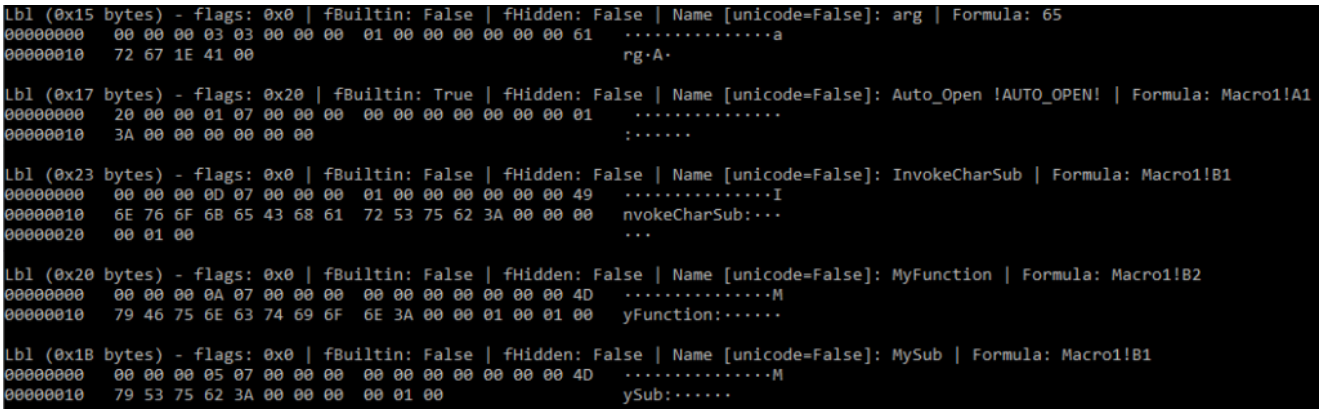


We can actually “create” our subroutine at runtime using **SET.NAME** to specify the subroutine cell and its argument
So this is slightly different from our previous examples, but the main difference is that we are invoking **SET.NAME** in order to specify two values:

1. We are defining the value of **InvokeCharSub** to be equivalent to a reference to cell B1. Later we invoke it using **InvokeCharSub()**, though we could also use **RUN(InvokeCharSub)**.
2. We are setting the value of the name “arg” to 65. This is essentially how we pass arguments to our subroutine. While there does appear to be an **ARGUMENT** function that allows explicitly defining names to store arguments, I haven’t been able to make this work any differently than just manually setting names or cell values. While porting EXCELntDonut macros into Macrome[21] I also realized that you can simply write **arg=65** in an Excel cell, and it will automatically be interpreted as **SET.NAME(“arg”,65)**



What a User Defined Function invocation looks like in byte form
Under the covers when we call **InvokeCharSub()**, we are having Excel call a user defined function through the **PtgFuncVar** Parse Thing object. User defined functions are a **PtgFuncVar** edge case – one of the arguments provided to the **PtgFuncVar** must be a **PtgName**[22]. **PtgName** objects reference a **Lbl**[23] entry stored within the Excel Workbook’s Globals Substream[24]. In this case, we are looking for the 3rd **Lbl** entry in the substream – it’s also worth noting that the index here starts at 1, rather than 0. We’ll come back to some “fun” that malware authors can have with these labels later.



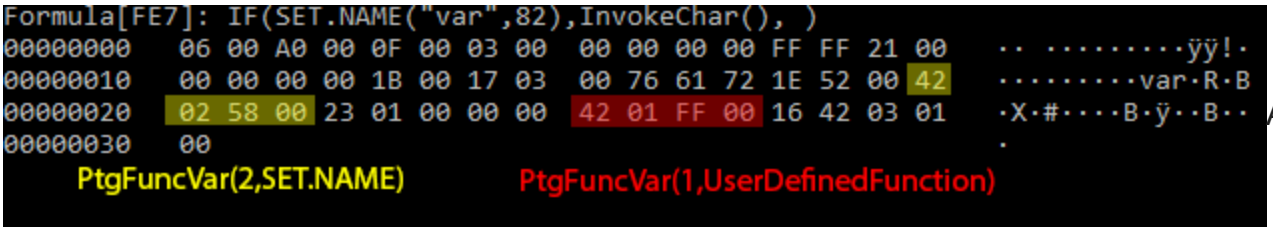
The **Lbl** list from our test document's Globals Substream – the 3rd item is InvokeCharSub, our subroutine name

So we have a mechanism to replace our **CHAR** function invocations with **SET.NAME** invocation followed by a call to a user defined function. This turns one very simple cell into two cells, but there's a workaround for that as well. A final possible optimization to reduce the size of our document is to combine our variable assignment with the invocation of our subroutine by abusing the **IF** function to execute two expressions in a single cell – for example:

```
=IF(SET.NAME("var",65),invokeChar(),)
```

The invocation of **SET.NAME** here saves us from having to use two cells to invoke our subroutine and lets us use a single cell which cuts down on our **FORMULA** record count by about half. This is the approach used by the **CharSubroutine** method in Macrome[10].

Going back to @Mattifestation's detection engineering approach – let's think about how we could detect this sort of approach and then analyze it. From a detection standpoint, a massive number of invocations of **SET.NAME** and **PtgFuncVar** objects with a user defined function would likely stand out. For example, if we look at the above **IF** statement at the byte level we get something like:



A single **FORMULA** record containing the **SET.NAME** and user defined function invocation

We can create a signature for this by keying on the presence of a **PtgFuncVar** invocation of **SET.NAME** (`42 02 58 00`) with some arbitrary locality to a **PtgFuncVar** invoking a user defined function (`42 ?? FF 00` – the **Ftab** value is `FF 00` , but we need a wildcard since we can't necessarily guess the argument count). Our signature doesn't need to care if **SET.NAME** comes before or after the user defined function, we just want to check for a large number of these instances. A Yara[25] signature for this could look like:

```
rule msxls_set_name_and_invoke_udf
{
  meta:
    description = "Finding XLS2003 documents with a suspicious number of SET.NAME and
User Defined Function invocations"
    Author = "Michael Weber (@BouncyHat)"
  strings:
    $ole_marker = {D0 CF 11 E0 A1 B1 1A E1}
    $setname_invokeudf = {42 02 58 00 [0-100] 42 ?? FF 00}
    $invokeudf_setname = {42 ?? FF 00 [0-100] 42 02 58 00}
  condition:
    $ole_marker at 0 and (#setname_invokeudf > 100 or #invokeudf_setname > 100)
}
```

Note that the wildcard range `[0-100]` probably makes this computationally expensive to run on a large dataset, but the upper bound of 100 wildcard bytes could be lowered as needed.

This signature could still be avoided (as is true for most signatures) with a little additional effort on the part of the attacker. As demoed in Outflank's research[26], we can use Excel's **WHILE** functionality to iterate over a column of seemingly harmless numbers and use them to build strings of binary data or additional macro statements to populate with the **FORMULA** function.

	A	B
1	=RETURN(CHAR(var))	stringToBuild=""
2	65	invokeChar=A1
3	66	curCell=A2
4	67	=WHILE(curCell<>"END")
5	68	var=curCell
6	69	stringToBuild=stringToBuild&invokeChar()
7	END	curCell=ABSREF("R[1]C",curCell)
8		=NEXT()
9		=ALERT(stringToBuild,2)
10		=HALT()

Here we have a Macro, starting at B1, that replaces our numerous **CHAR()** *invocations with a* subroutine at A1

But let's assume that there is a foolproof signature to identify our document and that our document has made its way into the hands of an analyst armed with a tool like XLMMacroDeobfuscator[6] or olevba[27]. Are there any weird behaviors that can be abused to trick analysts attempting to examine our document? Thanks to Excel's "flexibility" with **Lbl** records, the answer is yes.

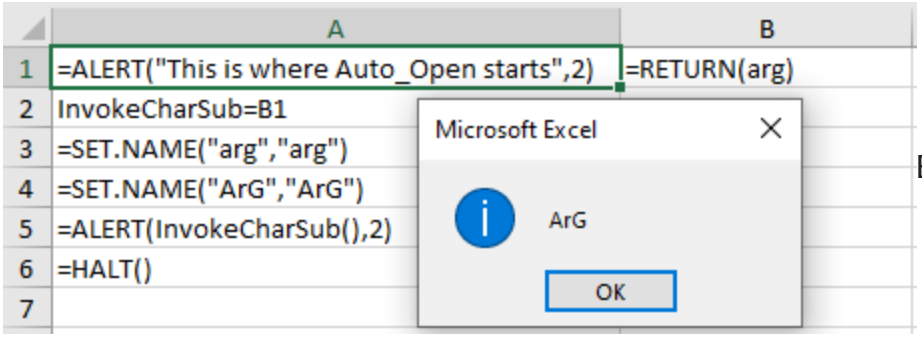
(Ab)Using Names in Excel 4.0 Macros

The usage of **Lbl** record lookups when resolving names is another opportunity for malware authors to frustrate analysis. In my previous blog post[1] I discussed how Excel's flexible handling of the **Auto_Open Lbl** record made signature creation extremely challenging. It

seems like similar issues would apply to “variable” and subroutine name invocation as well. For example – what would you expect the output of the following macro sheet to be?

	A	B
1	=ALERT("This is where Auto_Open starts",2)	=RETURN(arg)
2	InvokeCharSub=B1	
3	=SET.NAME("arg","arg")	
4	=SET.NAME("ArG","ArG")	
5	=ALERT(InvokeCharSub(),2)	
6	=HALT()	

Assuming case sensitivity were used, the string “arg” should be displayed

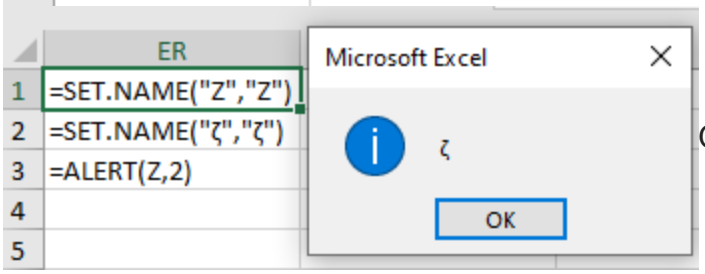


But Excel **Lbl** records are much more flexible than that

This looks like a nice trick, but it doesn’t appear to do much to frustrate analysis – at a glance. Just HOW flexible is Excel’s interchangeability with upper case and lower case letters?

	ER
1	=SET.NAME("Z","Z")
2	=SET.NAME("ζ","ζ")
3	=ALERT(Z,2)

What happens if we go into the Unicode character sets?



Obviously the lower case Zeta symbol (**ζ**) was going to overwrite that capital Zeta (**Z**)

It’s pretty flexible. There are a surprising number of multi-case characters to confuse Excel, just take a glance at the library of valid lower case Unicode characters[28]. Unfortunately, for defenders, the **PtgStr** record[29] used by Excel to invoke **SET.NAME** will happily allow attackers to set arbitrary Unicode content for arguments, so this is a challenging situation to avoid. The issues don’t stop at casing confusion either – Excel also respects Unicode Equivalence[30]. This behavior, which is part of the Unicode specification[31], is a consistent[32] source of pain[33] in the security world[34].

One example of how Unicode Equivalence can frustrate analysis is Decomposed Unicode. Decomposed Unicode values are alternate representations of Unicode characters that use a series of characters instead of a single Unicode character. For example – consider the Unicode character ạ[35]. This can be represented as 2 bytes in UTF-16 (Excel's Unicode interpretation) as `1E 01`. Alternatively, we can represent it as the letter **a** and the ◌̣ combining diacritical mark[36] – or `00 61 03 25`. (Note: These diacritical marks are the same bit of fun that can be used to create Zalgo monstrosities[37])

There also exist Unicode characters, like the Combining Graphene Joiner[38] (`03 4F`) which are essentially no-op characters for most Unicode strings. The Wikipedia article for the character explicitly describes it as "default ignorable" in the first sentence:

> "The **combining grapheme joiner** (CGJ), U+034F COMBINING GRAPHEME JOINER (HTML) is a Unicode character that has no visible glyph and is "default ignorable" by applications."
>
> *https://en.wikipedia.org/wiki/Combining_Grapheme_Joiner*

Finally, there are a sizable number of Unicode whitespace characters[39] which can change the byte contents of a string without changing its appearance. The "most interesting" of these whitespace characters are the zero-width Unicode characters. A zero-width character makes no visible change to the label. Some of these characters are ignored by Excel when comparing strings (U+200C, U+200D, U+2060, and U+FFEF), but others (U+180E and U+200B) are not. These characters can be used to pad variable names, or create decoy names that look the same but are not actually assigned when invoking **SET.NAME**.

There's nothing fundamentally bad about following the Unicode specification, but combining support for Unicode equivalence with some of Excel's other flexibility can lead to very counter-intuitive equivalencies. For example, `1E 01` (ạ) is considered the same as `20 60 00 41 03 25 03 4F 00` (a decomposed Ạ with some ignored Unicode characters added to the string). Replacing some of those bytes with a `18 0E` or `20 0B` would break the equivalency as well, which allows us to create strings that look identical, but are not treated as such by Excel. In practice this lets us create, using Macrome's[10] **AntiAnalysisCharSubroutine** method, the following content :

	FD	FE
1	=RETURN(CHAR('vạr'))	=IF(AND(SET.NAME("vạr",80),SET.NAME("vạr",137)),"(),)
2		=IF(AND(SET.NAME("vạr",1),SET.NAME("vạr",230)),"(),)
3		=IF(AND(SET.NAME("vạr",130),SET.NAME("vạr",221)),"(),)
4		=IF(AND(SET.NAME("vạr",178),SET.NAME("vạr",194)),"(),)
5		=IF(AND(SET.NAME("vạr",32),SET.NAME("vạr",217)),"(),)
6		=IF(AND(SET.NAME("vạr",43),SET.NAME("vạr",118)),"(),)
7		=IF(AND(SET.NAME("vạr",244),SET.NAME("vạr",245)),"(),)
8		=IF(AND(SET.NAME("vạr",218),SET.NAME("vạr",91)),"(),)
9		=IF(AND(SET.NAME("vạr",143),SET.NAME("vạr",83)),"(),)
10		=IF(AND(SET.NAME("vạr",121),SET.NAME("vạr",89)),"(),)

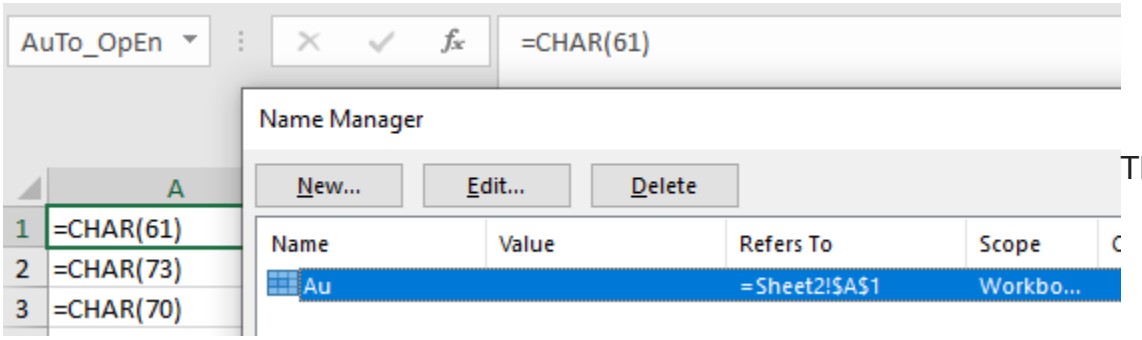
It is random whether the first SET.NAME or second SET.NAME in each cell set the value passed to the subroutine

Although the **vạr** strings appear to be identical, they are in fact quite different on disk. This means that any analysis of the cell to figure out what will actually happen will require running Excel or manually reproducing Excel's EXACT handling of Unicode characters. Reproducing the behavior is going to require handling a lot of edge cases. If you want a sense of what analysts could be up against, here's what the above example looks like in binary:

```
Formula[FE783]: IF(AND(SET.NAME("v?????r",227),SET.NAME("v?????r",68)), (), )
00000000   0E 03 A0 00 0F 00 03 00  00 00 00 00 FF FF 21 00   .. .........ÿÿ!.
00000010   00 00 00 00 42 00 17 07  01 76 00 0B 20 0E 18 0B   ....B....v..  ...
00000020   20 01 1E 0E 18 72 00 1E  E3 00 42 02 58 00 17 07    ....r..a.B.X...
00000030   01 76 00 60 20 0C 20 0D  20 01 1E 4F 03 72 00 1E   .v.` . . ..O.r..
00000040   44 00 42 02 58 00 42 02  24 00 23 01 00 00 00 42   D.B.X.B.$.#....B
00000050   01 FF 00 16 42 03 01 00                           .ÿ..B...

Decoy Arg Bytes: 00 76 20 0B 18  0E 20 0B 1E 01 18 0E 00 72
Real Arg Bytes: 00 76 20 60 20 0C 20 0D 1E 01 03 4F 00 72
Lbl Name Bytes: 00 00 00 76 FF FD 00 41 03 25 FF EF 00 72

NOTE: Adjusted for Byte Endianness

Lbl (0x1D bytes) - flags: 0x1 | fBuiltin: False | fHidden: True | Name [unicode=True]:  v?A??r
00000000   01 00 00 07 00 00 00 00  00 00 00 00 00 00 01 00   ................
00000010   00 76 00 FD FF 41 00 25  03 EF FF 72 00           .v.yÿA.%.ïÿr.
```

Note that both **SET.NAME** arguments are very different from the **Lbl** name used in **=RETURN(CHAR('vạr'))**

In the above example the "Real" argument bytes are considered a match for the **Lbl** name bytes, but the "Decoy" argument bytes are not. The fact that **Lbl** record strings can be so wildly different from the **PtgStr** arguments passed to **SET.NAME** makes it challenging to follow Excel's data flow without actually running Excel. Even then, Excel isn't consistent with handling Unicode values – see what happens when null bytes are injected into the **Auto_Open** label after the **u** character:



The Name Manager sees Au, but the cell label is AuTo_OpEn

Given the already low detection rate for Excel 4.0 macros in the wild, we may never see attackers need to rely on this level of trickery. If AV does start getting better signal with their signatures though, I will not be surprised to see various forms of Unicode abuse begin to crop up.

Updates to Macrome

In the process of digging deeper into Excel documents, I've often come across a need to examine the byte content of specific records as a hex dump. While I don't mind crawling through a wall of hex text, I've managed to save some time by modifying my tool Macrome to dump the hex content of **Lbl** and **Formula** records. All of the hex examples from this post were generated using this dump functionality. I've also implemented code for generating proof-of-concept documents using some of the subroutine and Unicode shenanigans that I discussed in this post. If you want to try generating some malicious documents to see how your tooling will handle these kinds of documents I'd suggest heading over to https://github.com/michaelweber/Macrome and grabbing the latest release.

As always, if folks have any suggestions for features or improvements, please let me know here in the comments or open an issue on the Github project page.

References

1. https://malware.pizza/2020/05/12/evading-av-with-excel-macros-and-biff8-xls/
2. https://www.lastline.com/labsblog/evolution-of-excel-4-0-macro-weaponization/
3. https://inquest.net/flash-alerts/IQ-FA004%3AMultiple_Actors_Abusing_New_Macro_Methods
4. https://twitter.com/InQuest/status/1268568312499376130
5. https://twitter.com/DissectMalware/status/1268491222299086854
6. https://github.com/DissectMalware/XLMMacroDeobfuscator
7. https://twitter.com/Anti_Expl0it/status/1269895583633829888
8. https://github.com/FortyNorthSecurity/EXCELntDonut/
9. https://github.com/TheWover/donut
10. https://github.com/michaelweber/Macrome
11. https://www.virustotal.com/gui/file/b159b25b80b1830acf40813c06a48f3e72666720b7efcd406ea5031c7f214c31/detection

12. https://twitter.com/mattifestation/status/1263416936517468167
13. https://pastebin.com/V8SGgdZL
14. https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/87ce512d-273a-4da0-a9f8-26cf1d93508d
15. https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/00b5dd7d-51ca-4938-b7b7-483fe0e5933b
16. https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/5d105171-6b73-4f40-a7cd-6bf2aae15e83
17. https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/8e3c6978-6c9f-4915-a826-07613204b244
18. https://excelofftthegrid.com/using-excel-4-macro-functions/
19. https://twitter.com/DissectMalware/status/1269535826813366273
20. https://www.virustotal.com/gui/file/a53be0bd2a838ffe172181f3953a2bc8a1b7c447fb56d885391921a7c3eac1f9/details
21. https://github.com/michaelweber/Macrome/releases/tag/0.2.0
22. https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/5f05c166-dfe3-4bbf-85aa-31c09c0258c0
23. https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/d148e898-4504-4841-a793-ee85f3ea9eef
24. https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/ca4c1748-8729-4a93-abb9-4602b3a01fb1
25. https://virustotal.github.io/yara/
26. https://outflank.nl/blog/2018/10/06/old-school-evil-excel-4-0-macros-xlm/
27. https://github.com/decalage2/oletools/wiki/olevba
28. https://www.compart.com/en/unicode/category/Ll
29. https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/87c2a057-705c-4473-a168-6d5fac4a9eba
30. https://en.wikipedia.org/wiki/Unicode_equivalence
31. https://www.unicode.org/versions/Unicode13.0.0/UnicodeStandard-13.0.pdf
32. https://www.dionach.com/en-us/blog/fun-with-sql-injection-using-unicode-smuggling/
33. https://hackernoon.com/%CA%BC-%C5%9B%E2%84%87%E2%84%92%E2%84%87%E2%84%82%CA%88-how-unicode-homoglyphs-will-break-your-custom-sql-injection-sanitizing-functions-1224377f7b51
34. https://book.hacktricks.xyz/pentesting-web/unicode-normalization-vulnerability
35. https://www.compart.com/en/unicode/U+1E01
36. https://www.compart.com/en/unicode/U+0325
37. https://zalgo.it/en/
38. https://en.wikipedia.org/wiki/Combining_Grapheme_Joiner
39. https://en.wikipedia.org/wiki/Whitespace_character