2019 LONDON 2 – 4 October 2019

STATIC ANALYSIS METHODS FOR DETECTION OF MICROSOFT OFFICE EXPLOITS

Chintan Shah
McAfee, India

chintan_shah@mcafee.com

ABSTRACT

Despite recent advances in exploitation strategies and exploit mitigation techniques, fundamental infection vectors remain the same. It is critical to advance security solutions to inspect both new and known infection vectors in order to successfully mitigate targeted attacks. Apparently, the use of lure documents has become one of the most favoured attack strategies for infiltrating target organizations. Recently, some of the most high impact attacks using this conventional technique have been uncovered by the security community.

In this paper, we present an exploit detection tool that we built for the purpose of detecting malicious lure documents. This detection engine employs multiple binary stream analysis techniques for flagging malicious *Office* documents, supporting static analysis of RTF, Office Open XML and Compound Binary File format (MS-CFB). The use, by attackers, of weaponized lure documents necessitates deeper inspection of these file formats at the perimeter.

Object Linking and Embedding exposes a rich attack surface which has been abused by attackers over the past few years to hide malicious resources. For instance, OOXML files can be used to load OLE controls which can eventually facilitate remote code execution. Our proposed detection tool is built to extract embedded storage streams, OLE objects, etc. and apply binary stream analysis techniques over it, in addition to inspecting specific file sections and analysing embedded scripts, to identify malicious code. This detection tool had been tested over a wide set of in-the-wild exploits and variants.

INTRODUCTION

It is hardly surprising that some of the most infamous targeted attacks that we have spotted in the past used conventional attack vectors and infection techniques to penetrate their target organizations. Multiple attacks using lure documents have been uncovered by the security community over the last year. Since attackers using this technique to execute phishing attacks would most likely deliver the weaponized exploit documents to the target, it becomes a pressing need for any perimeter security solution to investigate these file formats a little deeper for signs of maliciousness. Network and endpoint security solutions have the capability to look deeper into several file formats, but seemingly have limited detection capability of weaponized documents exploiting zero-day vulnerabilities. Modern sandboxing solutions also support analysis of multiple file formats, but often do not provide complete behaviour visibility. It is critical to augment the exploit detection capability of these solutions with an engine that can perform static inspection of files and classify documents based on the characteristics of the embedded binary content.

Object Linking and Embedding (OLE), a technology based on Component Object Model (COM), is one of the features in *Microsoft Office* documents which allows the objects created in other *Windows* applications to be linked or embedded into documents, thereby creating a compound document structure and providing a richer user experience. OLE had been massively abused by attackers over the past few years in a variety of ways. OLE exploits in the recent past have been observed either loading COM objects to orchestrate and control the process memory, taking advantage of the parsing vulnerabilities of the COM objects, hiding malicious code or connecting to external resources to download additional malware. We have had multiple instances of OLE exploits using the multi-COM loading method to execute an attack. With this, it becomes vital for any security solution to inspect documents at the perimeter before they reach the endpoint. Additionally, it is fundamental to inspect other attack surfaces like embedded scripts, Flash files, etc. to be able to detect unknown attacks.

In the following sections, we describe the Static Analysis Engine (SAE) that we implemented for a similar purpose. The Static Analysis Engine supports the inspection of OLE Compound Binary File format (MS-CFB), Rich Text Format (RTF) and OOXML file format, and applies binary stream analysis techniques to identify unusual streams of data. SAE utilizes the underlying document parsing capabilities to extract all the embedded or linked COM objects from *Microsoft Office* documents and further analyses them for any suspicious or malicious indicators. It also extracts the embedded object and storage streams from the Compound Binary File format and explores the possibility of injected malicious code by emulating and statically analysing these streams. It is also important to analyse known attack surfaces such as embedded VB macro scripts, since targeted attacks using lure documents with obfuscated macro scripts have been on the rise. It is crucial to look for attack vectors that deliver other file format exploits, such as Flash files, from within the *Microsoft Office* documents, extract and probe them for any possible malicious indications.

In the following sections, we walk through some of the malicious indicators, inspection methods and heuristics implemented by the SAE over the various file formats and share some of the initial observed results towards the end. However, the methods outlined here are by no means an exhaustive list.

STATIC ANALYSIS OF RTF (RICH TEXT FORMAT) FILES

RTF documents have been one of the primary exploitation targets. Attackers have predominantly used RTF parsing and logic vulnerabilities to deliver malware and execute attacks. In the following sections, we highlight some of the inspection methods for identifying malicious RTF documents.

RTF control words

Rich Text Format files are heavily formatted using control words. Control words in RTF files primarily define the way a document is presented to the user. Since these RTF control words have associated parameters and data, parsing errors for them can become the primary target for exploitation. Exploits in the past have been found using control words to embed malicious resources as well. Consequently, it becomes important to examine destination control words that consume data, and to extract and analyse the embedded binary stream for malicious indicators. Figures 1 to 3 show a few instances of past exploits using control word parameters to introduce malicious code or executable payloads.

```
{\rtf1{\shp{\sp{\sn pFragments}{\sv 7
11111111acc8b890439c04dac0d97424f45b31c9b15683c30431430f03439fa169f877a7920187c81be4b6c8786ce8f80b2
0047259d19ff676d628bca0d9a9ed917829ecc55a103f189a5522d1ce0e2844ff3b6455747768dd69cf8bcc3f44d2cebe89
6e47d9ce4b11522427a0b275c80ffbba3b513b7ca424357f593f820285ca11a44e6cfe5582eb75596f7fd17d6eac6979fb5
3be08bf771a511b193b3fca265be0b382170ca7be755804f385980284f6aa8d3e91864699669f411ab82701e439570b236d
0723820eccb32bdb78bebbee7cbc3d877ec040ecf726124257f7d33217a7bb589898dc6273b1778d2de9ef34746191b9a30
f913241ef5cb320e389a4cafb4941ca914dc39d0d4c32e991af116ad550e45aad6772e2d98792e219def8e2718658b164c9
74a6345c779ee9f71f1cd73080df3243c71fc06c60773a2d908750adc0efaf82efdf5009b877dadc0ae6dbf4cbb6dcfbd74
9a674e7aa579d8cab57a1b2908198c0d7119fdb6237b6718c6bc853c39a11a51b2b6cb4dfbdb2052a4f18bfeb9f1696a1b2
5fd67ff33c0ac68ddad288a5a5ea505b53a25276389b2f68e4d82dbfb51c1124be3e67975da442eef46c1772bd32fb4a5be
```

Figure 1: A previous exploit with embedded binary data inside the 'PFragments' RTF control word.

```
6F 72 69 74 79 33 39 20 5C 6C 73 64 6C 6F 63 6B   ority39 \lsdlock
65 64 30 20 54 4F 43 20 48 65 61 64 69 6E 67 3B   ed0 TOC Heading;
7D 7D 7B 5C 2A 5C 64 61 74 61 73 74 6F 72 65 20   }}{\*\datastore 
66 66 66 66 66 66 66 66 66 32 65 35 32 66 62 66   fffffffff2e52fbf
62 63 62 66 62 66 62 66 62 62 62 66 62 66 62 66   bcbfbfbfbbbfbfbf
34 30 34 30 62 66 62 66 30 37 62 66 62 66 62 66   4040bfbf07bfbfbf
62 66 62 66 62 66 62 66 66 66 62 66 62 66 62 66   bfbfbfbfffbfbfbf
62 66 62 66 62 66 62 66 62 66 62 66 62 66 62 66   bfbfbfbfbfbfbfbf
62 66 62 66 62 66 62 66 62 66 62 66 62 66 62 66   bfbfbfbfbfbfbfbf
62 66 62 66 62 66 62 66 62 66 62 66 62 66 62 66   bfbfbfbfbfbfbfbf
62 66 62 66 62 66 62 66 62 66 62 66 62 66 62 66   bfbfbfbfbfbfbfbf
36 66 62 66 62 66 62 66 62 31 61 30 30 35 62 31   6fbfbfbfb1a005b1
62 66 30 62 62 36 37 32 39 65 30 37 62 65 66 33   bf0bb6729e07bef3
37 32 39 65 65 62 64 37 64 36 63 63 39 66 63 66   729eebd7d6cc9fcf
63 64 64 30 64 38 63 64 64 65 64 32 39 66 64 63   cdd0d8cdded29fdc
64 65 64 31 64 31 64 30 63 62 39 66 64 64 64 61   ded1d1d0cb9fddda
39 66 63 64 63 61 64 31 39 66 64 36 64 31 39 66   9fcdcad19fd6d19f
66 62 66 30 65 63 39 66 64 32 64 30 64 62 64 61   fbf0ec9fd2d0dbda
39 31 62 32 62 32 62 35 39 62 62 66 62 66 62 66   91b2b2b59bbfbfbf
62 66 62 66 62 66 62 66 63 62 61 36 36 31 33 62   bfbfbfbfcba6613b
38 66 63 37 30 66 36 38 38 66 63 37 30 66 36 38   8fc70f688fc70f68
38 66 63 37 30 66 36 38 30 63 64 62 30 31 36 38   8fc70f680cdb0168
38 34 63 37 30 66 36 38 36 37 64 38 30 35 36 38   84c70f6867d80568
39 36 63 37 30 66 36 38 30 63 63 66 35 32 36 38   96c70f680ccf5268
38 61 63 37 30 66 36 38 38 66 63 37 30 65 36 38   8ac70f688fc70e68
63 63 63 37 30 66 36 38 36 37 64 38 30 34 36 38   ccc70f6867d80468
38 39 63 37 30 66 36 38 33 37 63 31 30 39 36 38   89c70f6837c10968
38 65 63 37 30 66 36 38 65 64 64 36 64 63 64 37   8ec70f68edd6dcd7
38 66 63 37 30 66 36 38 62 66 62 66 62 66 62 66   8fc70f68bfbfbfbf
62 66 62 66 62 66 62 66 65 66 66 61 62 66 62 66   bfbfbfbfeffabfbf
66 33 62 65 62 62 62 66 64 32 66 39 35 35 65 63   f3bebbbfd2f955ec
62 66 62 66 62 66 62 66 62 66 62 66 62 66 62 66   bfbfbfbfbfbfbfbf
```

Figure 2: CVE-2012-0158: Embedded executable payload inside the 'datastore' RTF control word.

```
38 37 33 5C 6C 65 76 65 6C 69 6E 64 65 6E 74 32   873\levelindent2
33 31 33 30 7B 5C 6C 65 76 65 6C 74 65 78 74 5C   3130{\leveltext\
27 66 66 5C 75 2D 34 38 38 33 31 20 3F 5C 75 34   'ff\u-48831 ?\u4
38 38 33 31 20 3F 5C 75 2D 37 31 39 35 20 3F 5C   8831 ?\u-7195 ?\
75 2D 36 34 38 32 20 3F 5C 75 2D 35 35 34 35 39   u-6482 ?\u-55459
20 3F 5C 75 2D 37 31 39 35 20 3F 5C 75 2D 36 33    ?\u-7195 ?\u-63
37 33 34 20 3F 5C 75 2D 37 31 39 35 20 3F 5C 75   734 ?\u-7195 ?\u
2D 36 33 37 33 34 20 3F 5C 75 2D 34 36 35 34 38   -63734 ?\u-46548
20 3F 5C 75 2D 35 35 34 36 33 20 3F 5C 75 2D 32    ?\u-55463 ?\u-2
30 34 31 34 20 3F 5C 75 2D 35 35 34 36 34 20 3F   0414 ?\u-55464 ?
5C 75 2D 31 36 39 31 38 20 3F 5C 75 2D 35 35 34   \u-16918 ?\u-554
35 35 20 3F 5C 75 2D 36 30 39 38 34 20 3F 5C 75   55 ?\u-60984 ?\u
2D 35 35 34 36 34 20 3F 5C 75 2D 35 35 33 30 20   -55464 ?\u-5530 
3F 5C 75 2D 35 35 34 35 36 20 3F 5C 75 2D 36 35   ?\u-55456 ?\u-65
34 30 37 20 3F 5C 75 2D 35 35 34 35 38 20 3F 5C   407 ?\u-55458 ?\
75 2D 36 35 35 33 36 20 3F 5C 75 2D 34 39 31 35   u-65536 ?\u-4915
32 20 3F 5C 75 2D 36 35 35 33 36 20 3F 5C 75 2D   2 ?\u-65536 ?\u-
36 35 35 32 30 20 3F 5C 75 2D 35 33 32 34 38 20   65520 ?\u-53248 
3F 5C 75 2D 36 35 35 33 36 20 3F 5C 75 2D 36 35   ?\u-65536 ?\u-65
34 37 32 20 3F 5C 75 2D 36 35 35 33 36 20 3F 5C   472 ?\u-65536 ?\
75 2D 36 31 34 34 30 20 3F 5C 75 2D 36 35 35 33   u-61440 ?\u-6553
36 20 3F 5C 75 2D 31 37 31 35 36 20 3F 5C 75 2D   6 ?\u-17156 ?\u-
35 35 34 35 37 20 3F 5C 75 2D 37 31 39 35 20 3F   55457 ?\u-7195 ?
5C 75 2D 36 33 37 33 34 20 3F 5C 75 2D 36 33 33   \u-63734 ?\u-633
39 31 20 3F 5C 75 2D 35 35 34 35 38 20 3F 5C 75   91 ?\u-55458 ?\u
2D 37 31 39 35 20 3F 5C 75 2D 36 33 37 33 34 20   -7195 ?\u-63734 
3F 5C 75 2D 37 31 39 35 20 3F 5C 75 2D 36 33 37   ?\u-7195 ?\u-637
33 34 20 3F 5C 75 2D 37 31 39 35 20 3F 5C 75 2D   34 ?\u-7195 ?\u-
36 33 37 33 34 20 3F 5C 75 2D 37 31 39 35 20 3F   63734 ?\u-7195 ?
5C 75 2D 36 33 37 33 34 20 3F 5C 75 2D 31 37 37   \u-63734 ?\u-177
32 37 20 3F 5C 75 2D 35 35 34 35 38 20 3F 5C 75   27 ?\u-55458 ?\u
2D 36 34 37 32 39 20 3F 5C 75 2D 35 35 34 35 38   -64729 ?\u-55458
20 3F 5C 75 2D 37 31 39 35 20 3F 5C 75 2D 36 33    ?\u-7195 ?\u-63
37 33 34 20 3F 5C 75 2D 36 35 35 33 36 20 3F 5C   734 ?\u-65536 ?\
```

Figure 3: CVE-2014-1761: Embedded shellcode inside the 'listlevel' RTF control word.

It is crucial for the Static Analysis Engine to scan the destination control words, especially those that consume data, since these could be the target for hiding the malicious content. *Microsoft* RTF specifications mention several such destination control words, a snapshot of which is shown in Figure 4.

To be able to generically detect such auxiliary strategies, an RTF document parser must be able to scan the control words that consume data and extract the stream, so that it can be passed on for

481	mbarPr	Consumes data
482	mbaseJc	Consumes data
483	mbegChr	Consumes data
484	mborderBox	Consumes data
485	mborderBoxPr	Consumes data
486	mbox	Consumes data
487	mboxPr	Consumes data
488	mchr	Consumes data
489	mcount	Consumes data
490	mctrlPr	Consumes data
491	md	Consumes data
492	mdPr	Consumes data
493	mdeg	Consumes data
494	mdegHide	Consumes data
495	mden	Consumes data
496	mdiff	Consumes data
497	me	Consumes data
498	mendChr	Consumes data
499	meqArr	Consumes data
500	meqArrPr	Consumes data
501	mf	Consumes data
502	mfName	Consumes data

Figure 4: Microsoft RTF specifications mention several destination control words that consume data.

Figure 5: The SAE extracts the data consumed by the 'datastore' control word and then passes it to the stream analyser.

additional scanning. RTF parsers must also be able to handle the control word obfuscation mechanisms commonly used by attackers, otherwise significant detections could be missed. The Static Analysis Engine integrates an RTF parser which performs sanity checks for such obfuscation attempts and extracts the data for the specific control words, apparently for performance reasons, which are then passed onto the supplementary scanning module. Figure 5 shows one of the instances as described previously: SAE extracts the data consumed by the 'datastore' control word and then passes it on to the stream analyser, which identifies the embedded single-byte XORed executable payload.

Malicious code inside embedded Microsoft OLE objects

Microsoft OLE links or *Microsoft* OLE embedded objects are represented in RTF documents as RTF objects, more precisely as a parameter to the RTF control word 'objdata'. The data for the object is hex encoded, stored in the OLESaveToStream format, which is supplied to the respective OLE application for processing when the OLE client is loaded into the application via a specified Class ID. It is imperative that this embedded OLE object is extracted from the RTF document and scanned for possible malicious code. On several occasions, crafted RTF exploits used as lure documents to execute a targeted attack have been observed to embed shellcodes in the object data and to exploit the vulnerability in the embedded OLE controls.

The CVE-2015-2424 RTF exploit, as shown in Figure 6, uses a multiple COM loading technique where malicious code is planted within the Forms.Image.1 OLE object while exploiting a memory corruption vulnerability within the Control.TaskSymbol.1 OLE object.

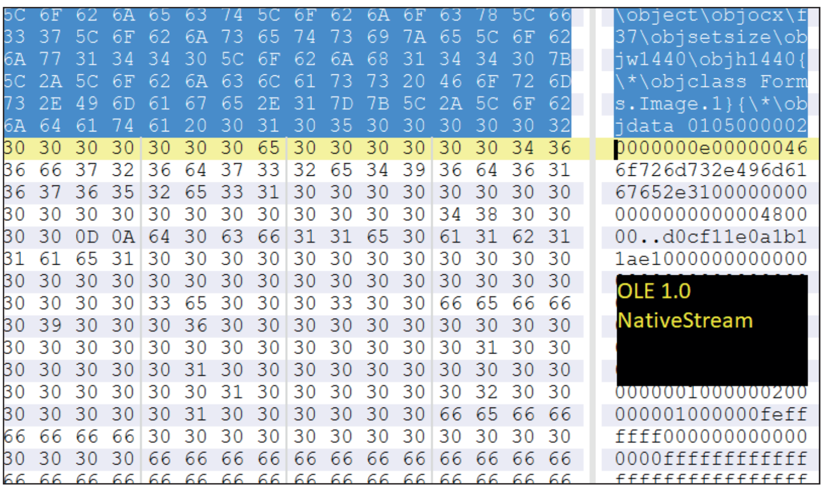

Figure 6: CVE-2015-2424 uses a multiple COM loading technique.

Figure 7 shows the injected code inside the OLE object.

The Static Analysis Engine can extract all the *Microsoft* OLE objects embedded inside RTF documents, parsing the RTF 'objdata' control word, and can inspect them for possible hidden malicious code.

```
ff ff 74 30 30 74 74 30  30 74 f1 9a 80 7c ea 63  |..t00tt00t...|.c
14 77 00 00 d1 00 00 00  20 00 00 30 00 00 40 00  |.w...... ..0..@.
00 00 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................
90 90 e9 19 00 00 00 5e  31 db b3 67 31 c9 b9 51  |.......^1..g1..Q
00 00 00 89 f7 ac 30 d8  aa e2 fa e9 05 00 00 00  |......0.........
e8 e2 ff ff ff 8f 6b 67  67 67 fe fe fe 1e 5d 17  |......kggg....].
a3 5c 67 fb e3 f3 3d 8e  52 67 67 67 39 de 5a 65  |.\g...=.Rggg9.Ze
67 67 ec 25 63 68 c3 65  6f ec 3d 6f 68 c3 bf 6f  |gg.%ch.eo.=oh..o
a6 84 6f ee 25 63 ee 3d  6f ed 65 57 23 69 98 ed  |..o.%c.=o.eW#i..
25 61 ed 3d 6d 57 bf ef  25 6e 85 b1 8e 62 67 67  |%a.=mW..%n...bgg
67 8f a1 98 98 98 6c cd  3a 70 18 40 ca eb 58 08  |g.....l.:p.@..X.
be e4 72 38 3d 1a ca c9  a6 f0 31 0f fc 32 9b 19  |..r8=.....1..2..
40 25 0f 54 83 ea e1 5a  61 5c d1 8c 5a 5d b6 bc  |@%.T...Za\..Z]..
85 7f 73 fb 2c 9b af a4  59 f7 42 d5 e2 cc 8e ba  |..s.,...Y.B.....
```

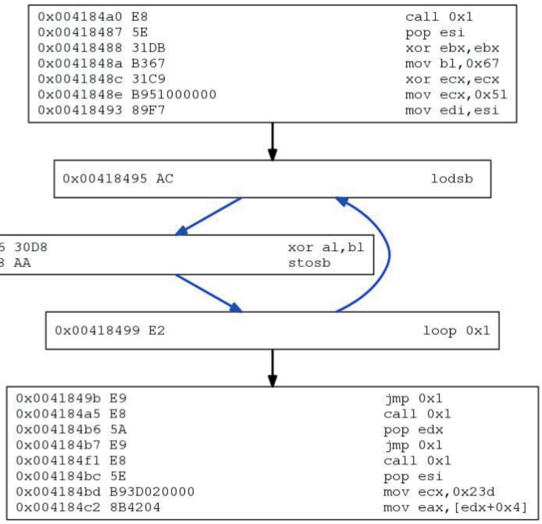

Figure 7: Injected shellcode inside OLE object.

```
01 05 00 00 02 00 00 00 09 00 00 00 4F 4C 45 32   ............OLE2
4C 69 6E 6B 00 00 00 00 00 00 00 00 00 00 0E 00   Link............
00 D0 CF 11 E0 A1 B1 1A E1 00 00 00 00 00 00 00   .ÐÏ.à¡±.á.......
00 00 00 00 00 00 00 00 00 3E 00 03 00 FE FF 09   .........>...þÿ.
00 06 00 00 00 00 00 00 00 00 00 00 00 01 00 00   ................
00 01 00 00 00 00 00 00 00 00 10 00 00 02 00 00   ................
00 01 00 00 00 FE FF FF FF 00 00 00 00 00 00 00   .....þÿÿÿ.......
00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   .ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

Figure 8: CVE-2017-0199 had an embedded OLE2Link object.

Links to external resources inside embedded Microsoft OLE objects

As a part of static analysis, it is critical to scan embedded OLE objects for any links pointing to external resources. Apparently, by embedding specific OLE controls inside *Microsoft Office* documents, exploits can be crafted to invoke respective handlers to parse or handle the downloaded resources. Evidently, attackers can take advantage of this functionality to exploit either logic bugs or resource-parsing vulnerabilities, which can eventually lead to full remote code execution.

Figure 8 is an example of a similar infamous RTF vulnerability, CVE-2017-0199, which was found to be exploited in the wild to deliver additional malware, and which had an embedded OLE2Link object.

The OLE2Link object enables Winword.exe to initiate the HTTP request to fetch an .hta file from the remote server. If we look at the OLE file, the OLE Stream object contained a link to the external resource, as highlighted in Figure 9, which, based on the server response, invoked the resource handler – in this case mshta.exe to execute the inserted malicious script inside the .hta file.

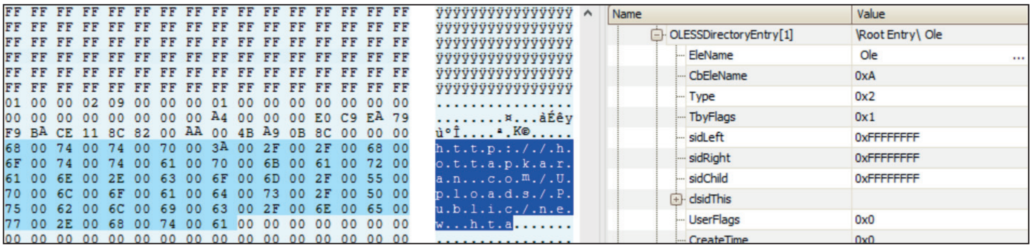

Figure 9: The OLE Stream object contained a link to the external resource.

Several other analogous cases have also been observed in the recent past. CVE-2017-8756 exploited the Web Service Description Language (WSDL) parsing code injection vulnerability by inserting an external link into the WSDL definition, which gets downloaded and parsed by the WSDL SOAP parser exploiting validation bug, leading to remote code execution.

Figure 10: CVE-2017-8756 inserted an external link into the WSDL definition.

CVE-2017-11882 was yet another vulnerability exploited in the wild to infect victims. This was a stack overflow in the Equation Editor OLE object with a link to download external resources.

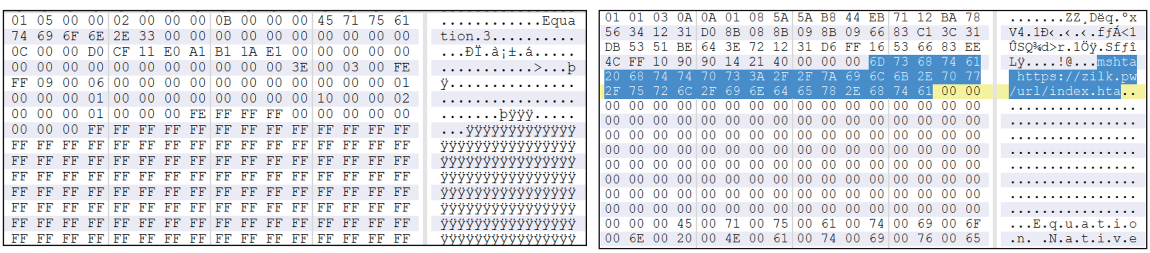

Figure 11: CVE-2017-11882 was a stack overflow in the Equation Editor OLE object with a link to download external resources.

Overlay data in RTF files

Overlay data is the additional data which is appended to the end of an RTF document and is predominantly used by exploit authors to embed decoy files or additional resources either in clear or encrypted form, and usually decrypted when the attacker-controlled code is executed. Overlay data having a volume beyond a certain size should be deemed suspicious and must be extracted and analysed further. However, the *Microsoft Word* RTF parser will ignore the overlay data while processing RTF documents. Figure 12 shows one RTF exploit, CVE-2015-1641, with 380KB of data appended at the end of the file, storing both the decoy document and multi-staged shellcodes with appropriate markers to aid decryption when the attacker-controlled code is executed.

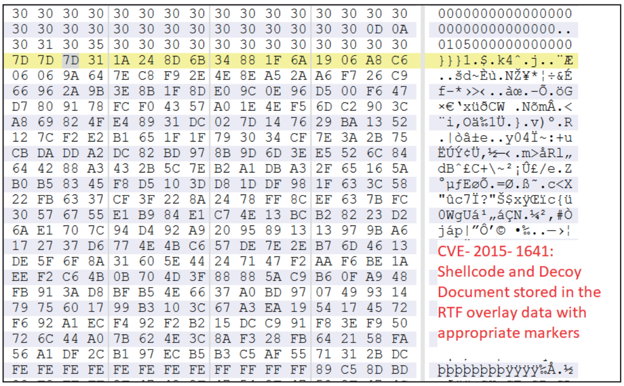

Figure 12: CVE-2015-1641 with decoy document and shellcode in the overlay section of the document.

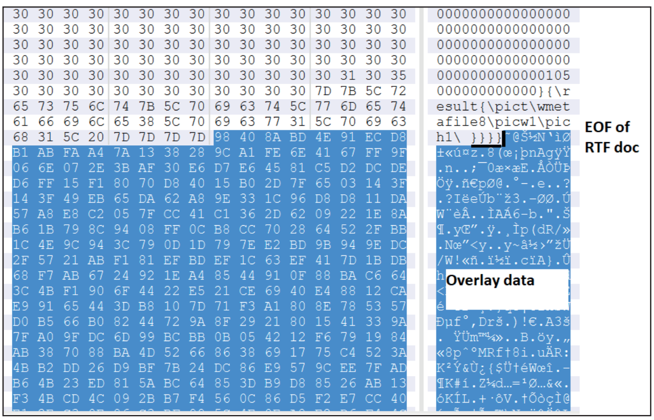

Figure 13: CVE-2017-11826 with 189KB of overlay.

To test the detection of overlay data inside RTF files, we ran the Static Analysis Engine over 2,483 RTF files with large-sized overlay data. The results are shown in Table 1. We found that more than 90% of the RTF documents with overlay data of more than 500 bytes had been found malicious as per *VirusTotal* detection.

Size of RTF overlay data section	Total RTF documents having overlay tested: 2,483	
	Overlay data section > 100B found: 2,310 [93 %]	
Overlay > 500B	Total found: 2,093	Malicious: 1,928 [92.11 %]
300B > Overlay <= 500B	Total found: 137	Malicious: 136 [99.27 %]
100B > Overlay < = 300B	Total found: 80	Malicious: 73 [91.25 %]
10B > Overlay <= 100B	Total found: 173	Malicious: 156 [90.17 %]

Table 1: Breakdown of results.

Embedded files inside RTF documents

Besides OLE files, RTF documents can have other files embedded at multiple locations, e.g. Flash files, Office Open XML format files, image files, etc. Extracting and re-analysing the embedded files becomes extremely important as a part of the static analysis process and on several occasions can become a decisive factor in identifying zero-day exploits. Extracted files can then be forwarded to the respective analysis modules for re-analysis. For instance, RTF exploits in the recent past had been found delivering Flash zero-day exploits, subsequently infecting the target with the additional malware. To support the exploitation process, weaponized RTF documents had been observed embedding OOXML files, on most occasions to perform the heap spray. Figure 14 is a snapshot of the CVE-2017-11826 RTF exploit used in the wild embedding malicious Office Open XML files to assist the further exploitation.

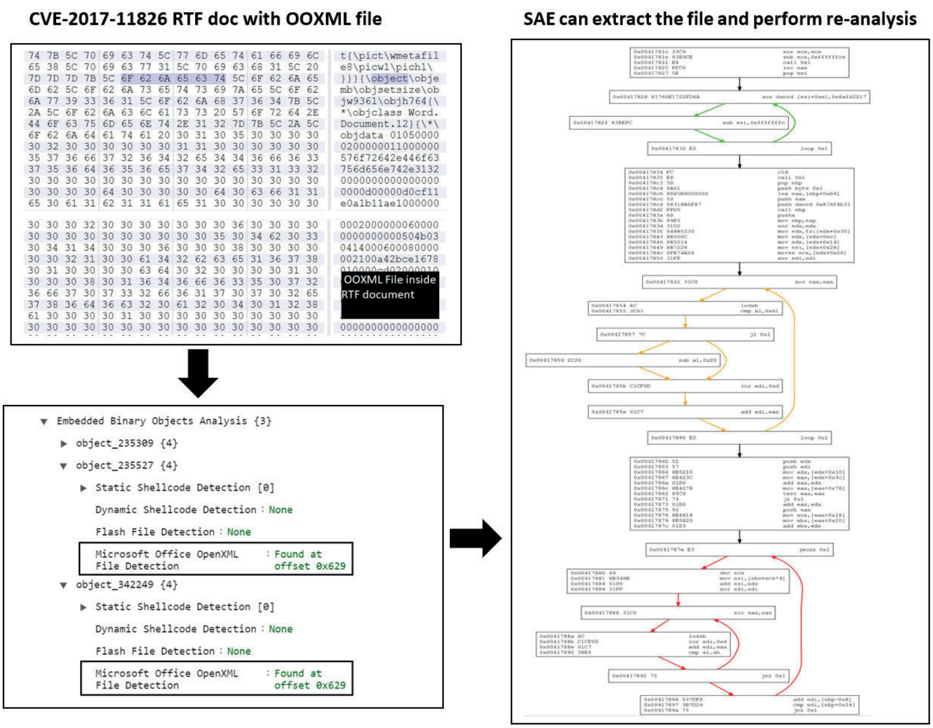

Figure 14: CVE-2017-11826 embeds malicious OOXML files.

STATIC ANALYSIS OF MS-OOXML (MICROSOFT OFFICE OPEN XML) FORMAT

Microsoft Office version 2007 and above introduced a new way of representing the documents in the form of XML schema, which replaced the previous binary file format representation. Office Open XML file format was specifically designed to consume less storage space, to increase performance and to increase the interoperability across multiple other applications. An Office Open XML (OOXML) file is preserved on the disk in the form of a compressed archive, comprising multiple compartmentalized markup documents with described relationships among them. The security and integrity of OOXML documents was also enhanced.

With the new document format, attack methods in OOXML files still revolve predominantly around exploiting OLE-based vulnerabilities and embedding malicious VBA macros. A sizeable proportion of exploits used in targeted attacks have been found exploiting OLE vulnerabilities, from memory corruption or logic bugs to undermining the *Windows* exploit mitigations by loading vulnerable or insecure OLE objects. The Static Analysis Engine essentially emphasizes the analysis of embedded ActiveX objects for any suspicious binary streams commonly used to assist further exploitation processes. The SAE also examines the objects for other inserted file formats and extracts them in order to forward them to other independent static analysis modules.

Suspicious loading of ActiveX objects

For ActiveX objects embedded inside an OOXML file, *Microsoft Office* creates a unique ActiveX.bin file, which is Compound Document Format, containing the CLSID corresponding to the library to be loaded in the application. *Office* reads the CLSID from the OLESS (OLE Structured Storage) stream and, post initialization of the OLE object, passes the storage data to the object for further processing via exposed interfaces. Attackers can abuse this OLE object-loading mechanism to load multiple OLE objects with the same but legitimate CLSID in order to perform heap spray. In some of the in-the-wild exploits, attackers have been found using fake CLSIDs, which do not point to any of the ActiveX libraries, to optimize and accelerate the heap spray process.

Figure 15: In some exploits attackers use fake CLSIDs which do not point to an ActiveX library.

It becomes important to examine if the OLE objects in the *Office* OOXML document are loaded suspiciously, along with performing a stream analysis of ActiveX.bin files for any malicious attributes. The same ActiveX object loading repeatedly should be deemed suspicious and corresponding .bin files should be analysed further.

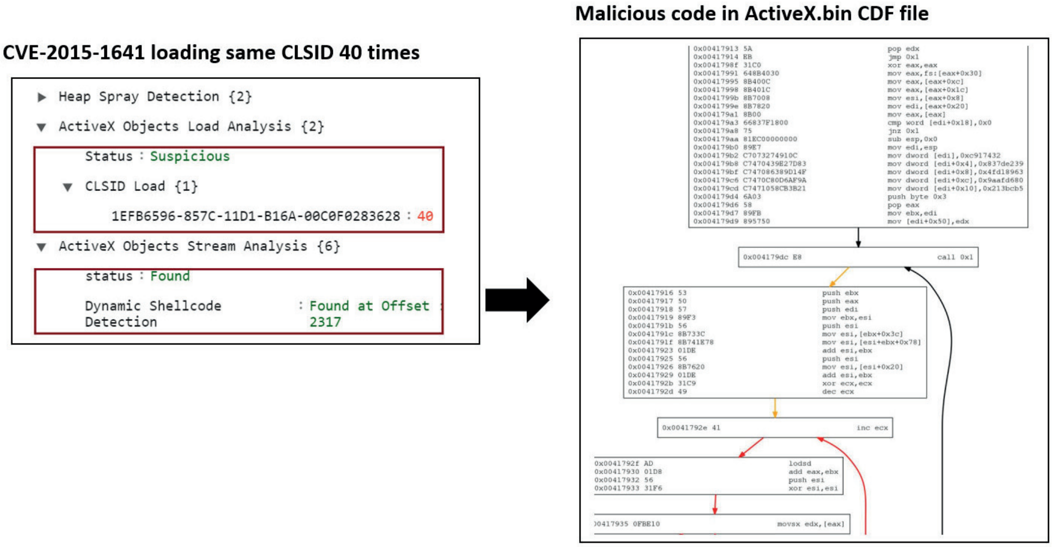

Figure 16: The same CLSID loading multiple times should be considered suspicious.

Another exploit, CVE-2017-11826, used in multiple targeted attacks, loaded a non-existent CLSID multiple times to be able to optimize and accelerate the heap spray process. Since there is no library associated with the class-id, heap spray time can be drastically reduced, increasing the overall performance.

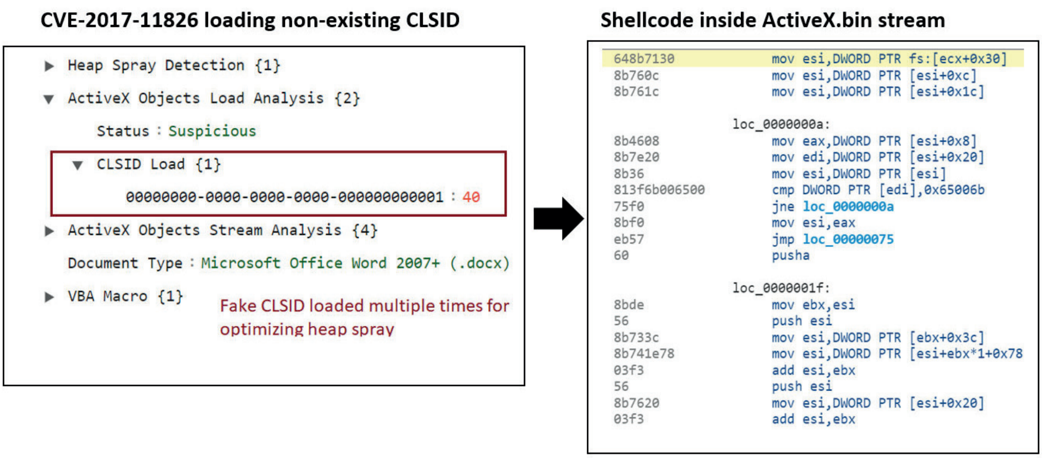

Figure 17: CVE-2017-11826 loaded a non-existent CLSID multiple times.

Identifying ROP chains and sledges in OLE object

The Static Analysis Engine also analyses the embedded OLE structured storage streams for any possible sledges, which is most likely the address within the loaded module that points to the instructions, usually a junk code to increase the possibility of successful exploitation. Sledges are then usually followed by ROP gadgets, which are subsequently executed to bypass the *Windows* exploit mitigations. Figure 18 shows an OLE stream from one of the previous exploits used in the targeted attacks, highlighting the sledges, ROP chain and the shellcode.

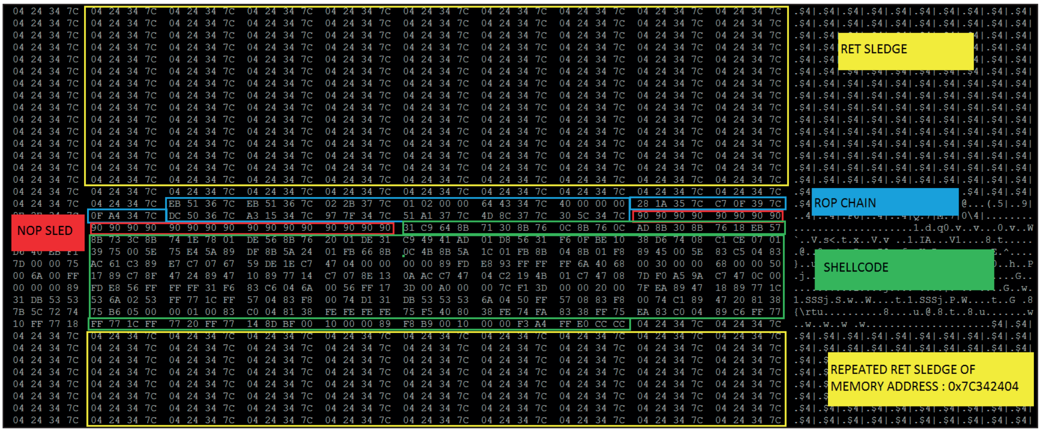

Figure 18: OLE stream with sledges, ROP chain and shellcode highlighted.

The SAE applies an analysis algorithm to guess the valid address sequence within the binary stream and then attempts to further establish the sequence by performing deeper checks to eliminate false positives. Figure 19 shows the result of the SAE correctly extracting the ROP chain and the sledge from the binary stream shown in Figure 18.

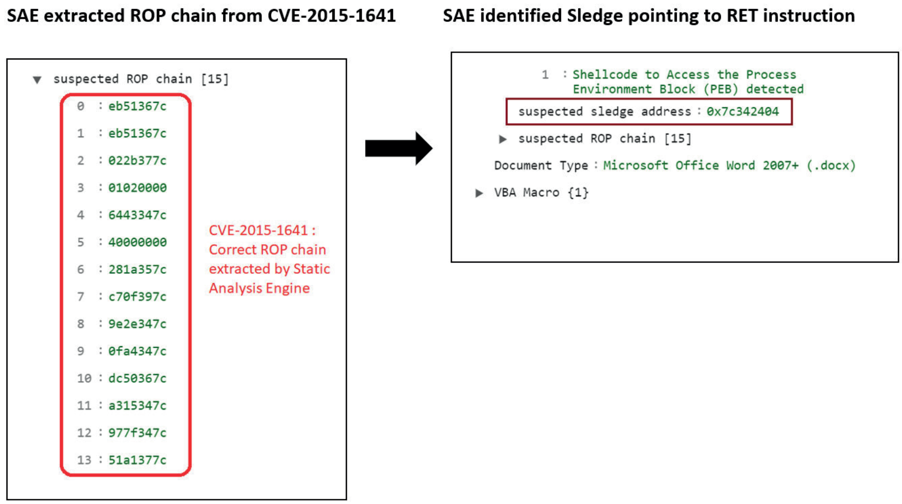

Figure 19: The SAE correctly extracts the ROP chain and sledge.

STATIC ANALYSIS OF MS-CFB (MICROSOFT COMPOUND FILE BINARY FILE) FORMAT

Compound Binary File format is a complex and legacy file format that existed before *Office 2007*, after which the newer and much simpler OOXML format was introduced. A compound file format provides a user with an efficient way to store multiple different kinds of objects (images, charts, documents, etc.) within a single hierarchical file structure in the form of stream and storage objects. All these stream and storage objects are stored in a separate directory entry, collectively known as structured storage, which increases the overall performance of the file system. Compound file format is organized in the form of sectors, containing user-defined data for stream objects; directory sectors which contain several directory entries; and free space to store additional objects when required. Sectors can be of multiple types such as FAT sectors, DIFAT sectors and mini FAT sectors.

Scanning storage and stream objects

While variants of FAT sectors are predominantly for the allocation of space within the compound file, there is one that is of primary interest to us: File Directory sectors, which contain information about the stream objects and storage objects. Stream sectors are typically a collection of bytes and contain the user-defined data streams. There are no restrictions on the contents of the stream. It is critical for the Static Analysis Engine to parse the directory entries and locate these stream objects to be able to scan the byte streams for malicious code. Figure 20 shows an instance of the previous exploit parsed by the available Compound Binary File format parser, showing all the directory entries and storage types.

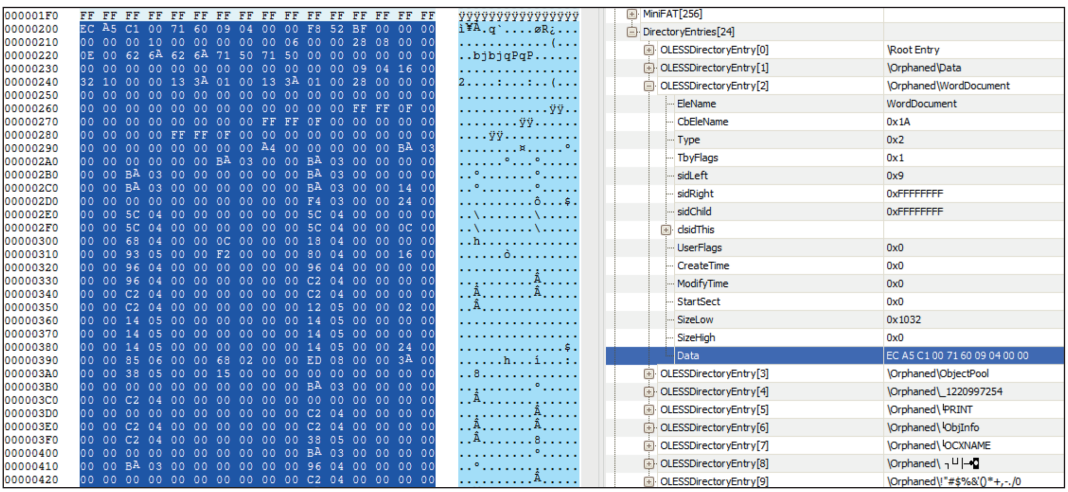

Figure 20: Exploit parsed by the Compound Binary File format parser.

Another section of the Compound Binary File which is of specific interest is the ObjectPool storage. ObjectPool storage contains storage for the embedded OLE objects and can be abused by attackers to insert malicious code into the weaponized exploits, as discussed in the earlier sections. Figure 21 shows the CVE-2018-4878 MS-CFB file-embedding Flash exploit, where the Static Analysis Engine extracts all the stream objects.

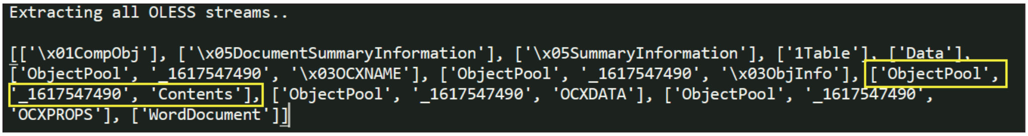

Figure 21: The SAE extracts all the stream objects.

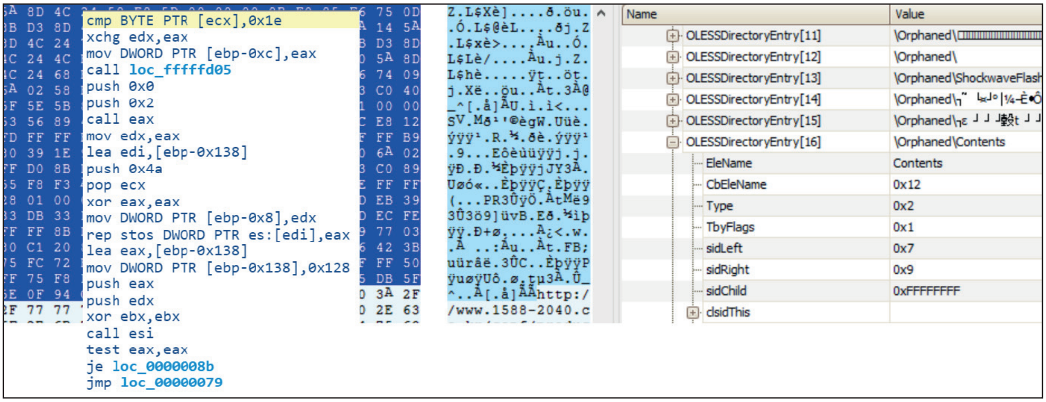

Figure 22: CVE-2018-4878: Compound Document format with embedded Flash exploit.

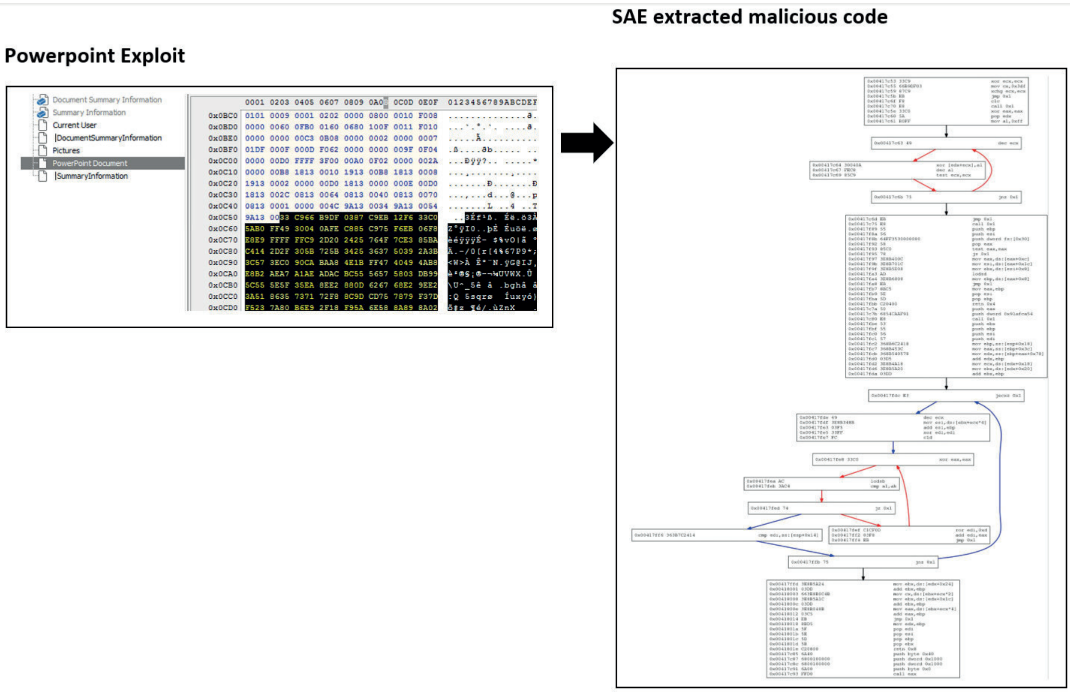

Figure 23: A weaponized PowerPoint exploit.

On analysis of the stream data, malicious code was identified in the 'Contents' stream of the ObjectPool storage.

While ObjectPool storage is one of the critical areas in the Compound Binary File format to examine, it is also essential, as indicated before, to locate stream objects in the other directory entries and scan them as well, predominantly looking for signs of embedded malicious code. Figure 23 shows an instance of a weaponized *Microsoft PowerPoint* exploit, where malicious code was found hidden in the 'PowerPoint Document' binary stream.

Extraction and analysis of VB macro code

In Compound Binary files, Visual Basic macro source code is located across multiple streams under the storage object called macros at the root storage of the OLE file. The macros storage object contains a VBA structured storage object which contains the /VBA/_VBA_PROJECT, /VBA/dir/ and several other streams containing macro source code. This code is stored as a compressed stream in the binary structure using rgw RLE (Run Length Encoding) compression algorithm, hence it is necessary to parse these binary streams in order to locate the code stream accurately. In an OOXML file, macro source is stored in the OLE file 'vbaProject.bin' within the zip archive. As indicated, this is again the OLE file with the same structure as the CFB file storage, and the macro source code is stored in the same format.

Malicious VBA (Visual Basic Application) malware has been on the rise in the recent past. Multiple high-impact targeted attacks have been executed by embedding malicious VBA macros inside *Office* documents. Therefore, it is essential for any static analysis solution to extract and classify the severity of the macro code. Figures 24 and 25 illustrate the storage of macro code in the two file formats.

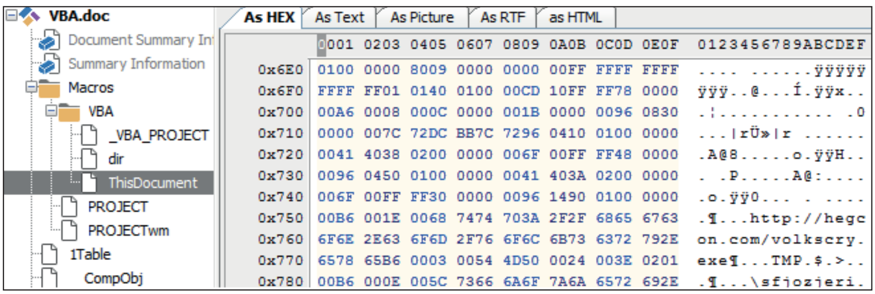

Figure 24: Macro storage in the 'ThisDocument' stream of a compound binary file.

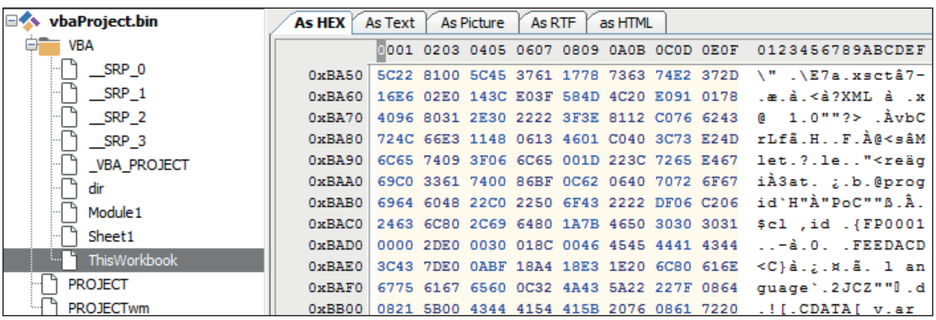

Figure 25: Macro storage in the 'ThisWorkBook' stream of an OOXML file.

The VB macro code classification module can extract the embedded VB macro code from the MS-CFB and MS-OOXML file formats, and applies code analysis for classification of malicious macros. Table 2 shows the results of initial testing done over 10,500 malicious macro embedded documents.

	Positive	Negative
True	96 %	3%
False	0.5 %	0.5%

Table 2: Results of initial testing over 10,500 malicious macro embedded documents.

HIGH-LEVEL IMPLEMENTATION OF THE STATIC ANALYSIS ENGINE

The Static Analysis Engine implements all of the previously described static analysis methods and concludes the classification of the input file based on the severity of the triggered heuristics. It includes multiple sub-analysis modules responsible for analysing various file formats depending upon the type of file passed as the input. Each sub-analysis engine has multiple heuristics implemented and respective checks are applied after the file is parsed by the integrated parser. It also implements an auxiliary generic stream analyser which is used by other analysis modules as and when required. Figure 26 is a high-level pictorial representation of the implementation. Analysis modules and their respective functionalities are indicated in the representation itself.

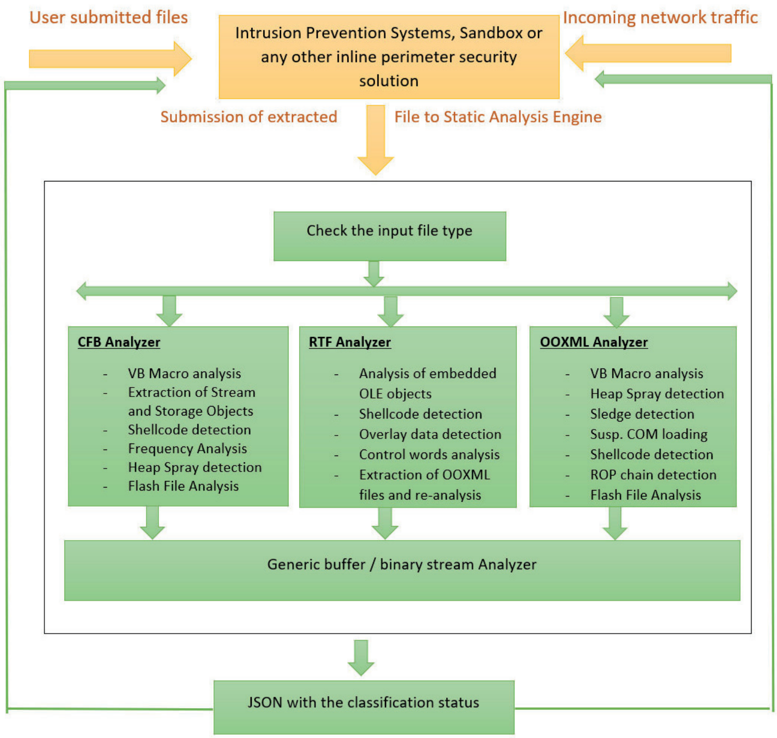

Figure 26: Implementation of the Static Analysis Engine.

RESULTS OVER IN-THE-WILD EXPLOITS

The Static Analysis Engine has been tested with all the implemented detection techniques over a number of in-the-wild exploits used in targeted attacks. Since the exploits used in the targeted attacks are weaponized, the implemented heuristics can be best tested over them. The results of this preliminary testing are shown in Table 3.

Exploit type	Type	Total exploits	Detected	Not detected	Rate
CVE-2012-0158	Mixed exploits (Targeted attacks and variants)	2000	1809	191	90.4%
CVE 2013-3906	Exploits used in targeted attacks	32	32	0	100%
CVE 2014-1761	Exploits used in targeted attacks	35	29	6	83%
CVE 2015-1641 CVE-2015-2424 CVE-2015-6172	Exploits used in targeted attacks and variants	150	138	12	92%
CVE 2016-4117	Exploits used in targeted attacks	87	77	10	88.5%
CVE-2017-11882	Exploits used in targeted attacks	12	11	1	91.6%
CVE 2018-4878 CVE-2018-15982	Exploits used in targeted attacks	30	27	3	90%

Table 3: Results of preliminary testing.

Mixed exploits detection results

Table 4 shows the results when the Static Analysis Engine was tested over the exploit variants. This includes multiple variants of malicious files with the CVEs shown above.

Exploit type	Total exploits	Detected	Not detected	Rate
Exploit variants from 2012 to 2018	4185	3754	431	89.70%

Table 4: Results when the Static Analysis Engine was tested over exploit variants.

It seems that the discussed detection mechanisms show a lot of promise in mitigating targeted attacks. Careful selection and implementation of additional heuristics will significantly improve the detection rate and, together, can certainly help to mitigate future attacks.