

Detection of malicious VBA macros using Machine Learning methods

Ed Aboud, Darragh O'Brien

Dublin City University

Abstract. Since their appearance in 1994 in the *Concept* virus, VBA macros remain a preferred choice for malware authors. There are two main attack techniques when it comes to document-based malware: exploits and VBA macros, with the latter applied in the vast majority of threats. Although Microsoft have added multiple security features in an attempt to protect users against malicious macros, such protections are often easily circumvented by simple social engineering techniques. Anti-virus companies can no longer rely on static signatures due to the rate at which new macro malware is distributed, and thus are tasked with employing a more proactive approach to threat detection. This paper details the literature on machine learning methods for the detection of VBA macro malware. Further, a machine learning system for the detection of VBA macro malware is proposed and evaluated. A Random Forest classifier achieves a true positive detection rate of 98.9875% with a false positive detection rate of 1.07% over a set of 611 mixed (benign and malicious) malware samples.

1 Introduction

The Microsoft Office suite allows users to leverage a powerful scripting engine known as Visual Basic for Applications (VBA) in the files it creates. VBA grants users a great deal of power by exposing the native Windows API [1]. This essentially means that for the most part a macro-enabled document can perform all the same tasks as a Portable Executable (PE), the native Windows executable binary format. Originally, as with other malware types, malicious macros were rarely deployed for financial gain [2]. More commonly they would take the form of a file infector or virus, with the sole intention of spreading to as many machines as possible but without executing any particular payload [3]. With the global shift in primary motivation for malware authors, however, this category of threats has been almost completely replaced by one whose aim is financial gain [4].

Up until the mid 2000s, users received little protection from Anti-virus software and their Operating System. This was the case until the release of Office 2007, which brought with it new security features to protect users from macro malware. In particular, Microsoft disabled the automatic execution of macros by default [5], which thwarted threats that relied solely on the user opening the document. In response to these security features, attackers must now focus on

the human element in the chain of compromise. To achieve execution of their malware, the attacker's victim must either take action to explicitly execute the macro, or have an unsafe Office security configuration. Furthermore, the Office suite has seen many of its discovered vulnerabilities exploited in the wild by malware [6] [7] [8]. However, a recent study by Proofpoint showed that 99.7% of documents used in attachment-based campaigns relied on social engineering and macros [9], rather than exploits. Thus in this paper exploit-based threats are excluded from further analysis.

Nowadays, macros typically serve as the initial step in the chain of compromise whereby they download a secondary payload upon macro execution [10]. By default, when a user opens a macro-enabled document, they are presented with a warning, and notified of the dangers associated with macros. Also presented is an *Enable Content* button along side the warning that when clicked, executes the embedded macros. Attackers require their victims to click this button to execute their malware, and to ensure this happens, many social engineering methods are deployed at this stage. Common tricks include fake instructions present in the document that falsely inform the user that the document was created in a previous version of Office, and to view the document they must click *Enable Content*. Another approach is to display seemingly encrypted text with a note instructing the user to click *Enable Content* in order to decrypt it.

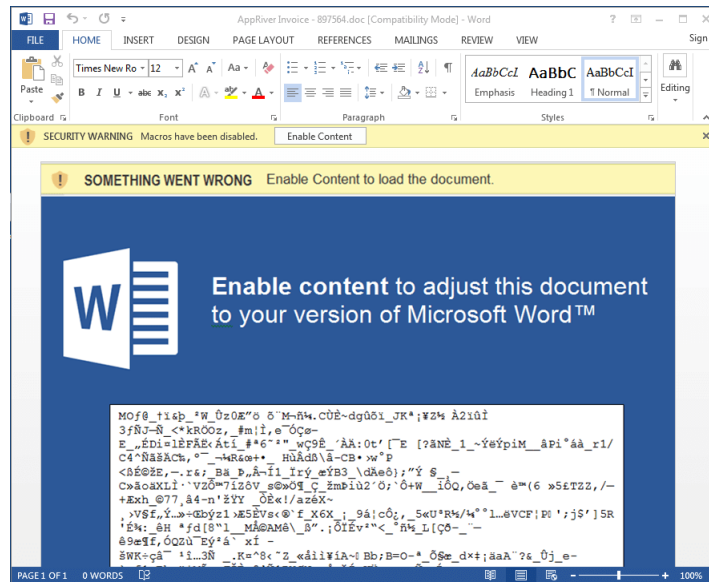


Fig. 1. Social Engineering

If the user clicks the *Enable Content* button, code execution begins at one of many possible entry points. Such entry points are event-triggered subroutines

that have the following definitions [11]:

AutoExec

When Word is started or a global template loaded

AutoNew

When a new document is created

AutoOpen/Document_Open

When an existing document is opened

AutoClose/Document_Close

When a document is closed

AutoExit

Upon exiting Word or unloading a global template

```
Private Sub Document_Open()
    Dim zLTELk As String
    dTvoRg = Left("astEosBfT[FeX", 4)
    While VNKNMP < 290
        NBWZB = Space(11)
        hNoap = "SaoAAF]I[UKY" + "zX#RsSda %AowDykdN" + ")y-&qJUA%F^k& wUJOE"
        uBUZsFx = RTrim("wzz OURZZ")
        ZrcknBcz = 922 + 1042 + 1090
        nQOid = Right("mZTnbse]kevcUev", 5)
        TrKMRRPD = LTrim("n^gp*klvaNTY")
        VNKNMP = VNKNMP + 3
    Wend
```

Fig. 2. Document_Open

The above definitions define the user-triggered events that cause the execution of a corresponding subroutine in the macro source code. The vast majority of macro malware contain at least one of such subroutine simply because they may be triggered with minimal user interaction. Recently, malware authors have been favouring the *AutoClose* subroutine due to the fact that some sandboxes are not sufficiently sophisticated to determine that to detonate the malware’s payload, the document must be opened and subsequently closed.

Another prevalent feature of macro malware is obfuscation. Unfortunately, however since some commercial software developers employ obfuscation to protect their source code, the presence of obfuscation alone does prove malicious intent. Rather, Office offers little protection to a developer who seeks to keep their VBA application’s source code closed and developers thus resort to obfuscation. This paper builds on previous research by including social engineering techniques as a feature in a machine learning classifier.

2 Motivation

In Symantec’s 2017 ISTR annual report on the current state of the threat landscape, it was found that macro-enabled documents and JavaScript downloaders

were the most commonly used approach to spreading malware via email [18]. According to the same report, macro-enabled documents were used in mass spam campaigns to spread high profile malware such as *Dridex* and *textitKotver*. In 2015 it was estimated that the financial damage due to *Dridex* was over \$40 million [22].

In 2015 Symantec published figures on the number of malicious email attachments blocked by their endpoint products. The most common file extension was *.doc* being blocked in 55.8% of emails with *.xls* in second place being blocked in 15.0% of emails [19].

Over the first half of 2017, Sophos reported in [20] that 68% of file types used to deliver malware were Word documents and 16% Excel spreadsheets.

According to a 2015 Cisco blog evaluating the detection rate of VBA macros by Anti-virus products, detection of malicious attachments at the time of email delivery ranges from 0 to 35% with an average detection rate of 8% [21]. In the same blog, it was also stated that malware download sites hosting malicious macros remained alive in some cases for over 29 days.

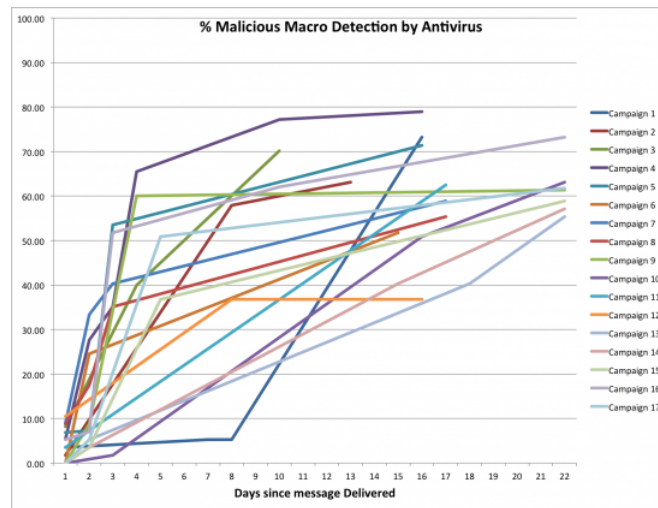


Fig. 3. <https://blogs.cisco.com/security/attackers-slipping-past-corporate-defenses-with-macros-and-cloud-hosting>

Considering that VBA macro malware has been around for over 20 years, is responsible for the theft of millions of dollars annually and is not adequately protected against by common security tools like Anti-virus, research into how to better defend potential victims against VBA attacks is clearly warranted.

3 Related Work

Peer reviewed research on the topic of macro malware detection is limited. ALDOCX is a machine learning classifier and a structural feature extraction framework implemented by Nisam et al in 2017 [12]. ALDOCX uses features extracted solely from file paths in the ZIP archive structure used by the Office Open XML variant of Office files (*docx*, *xlsx*, *pptx*). These file path based features are fed into an online learning SVM classifier. Nissam et al reported a 93.6% true positive detection rate with a 0.19% false positive rate. There were significant shortcomings to this framework however, the first being that only OOXML-based office files are supported and not legacy OLE files which remain the most common format today [13]. Secondly, the training set ultimately favours simply classifying any samples containing macros as malicious, as only 0.16% of their benign training set contained macros and only ZIP file paths were used as features and while these signify the existence of macros they reveal nothing of their contents.

Detection of various security vulnerabilities relating to format specification abuse, VBA and other exploitation techniques are discussed by Lagadec in [14]. Lagadecs methods for detection rely on standalone detection tools and not machine learning, and hence fail to generalise to detect previously unseen malware efficiently.

Rudd et al researched whether machine learning techniques that had been previously successful in the classification of malicious PE files could be extended to file types typically found in email attachments [13]. They collected 5 million Office documents from Virustotal Intelligence and using both XGBoost and DNN classifiers were able to achieve a detection rate of 99% using features such as string length, byte entropy and N-gram histograms. They also showed that string length features made the biggest contribution to their classifiers efficacy.

4 Background

Files created by the Microsoft Office suite typically use one of two formats [15]. Office versions prior to Office 2007 exclusively used a binary format known as OLE to store all data needed to represent a document, spreadsheet, etc. OLE files can be identified by their file extension, *.doc*, *.xls*, *.ppt* or by their first eight bytes, *D0CF11E0A1B11AE1*. Internally, OLE uses a hierarchy of data streams to store various components of the document.

In Fig.2 depicts the streams that make up a typical *.doc* document file. For example, the *WordDocument* stream contains all of the document's textual data. Streams under the *Macros* directory are of most significance for malicious VBA detection. Most streams found here contain metadata on the VBA project itself, bar *_VBA_PROJECT* which contains the actual VBA source code. The source code in this stream is compressed using the MS-OVBA algorithm [16] and there exist many tools and libraries to aid in its decompression. Here, the *oletools* library by declage [17] was used to extract the macro source code. Starting in Office 2007, Microsoft introduced a new format known as Office Open XML [15].

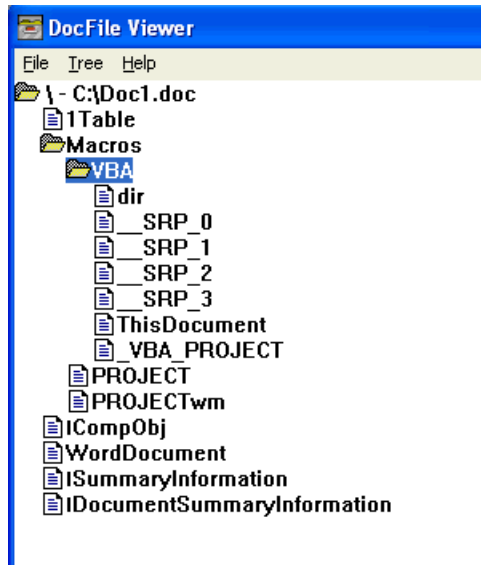


Fig. 4. OLE Structure

The OOXML format is essentially a ZIP archive containing an internal hierarchy of directories that in turn contain XML files used to both store both document metadata and actual document contents. Fig. 3 presents the top level structure of a *.xlsx* spreadsheet.

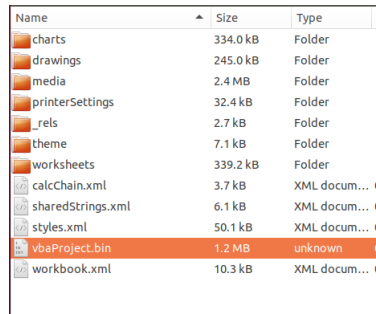
| Name | Size | Type |
|---------------------|-----------|--------------|
| docProps | 4.7 kB | Folder |
| _rels | 735 bytes | Folder |
| xl | 4.7 MB | Folder |
| [Content_Types].xml | 21.3 kB | XML docum... |

Fig. 5. Top level hierarchy

Of primary interest here is locating the VBA source code. This can be found in file *vbaProject.bin* as depicted in Fig. 4

5 Method

The aim of this research is to develop a classifier to make a binary classification on the behaviour of a macro-enabled document. The two possible classifications



| Name | Size | Type |
|-------------------|----------|----------------|
| charts | 334.0 kB | Folder |
| drawings | 245.0 kB | Folder |
| media | 2.4 MB | Folder |
| printerSettings | 32.4 kB | Folder |
| _rels | 2.7 kB | Folder |
| theme | 7.1 kB | Folder |
| worksheets | 339.2 kB | Folder |
| calcChain.xml | 3.7 kB | XML docum... 0 |
| sharedStrings.xml | 6.1 kB | XML docum... 0 |
| styles.xml | 50.1 kB | XML docum... 0 |
| vbaProject.bin | 1.2 MB | unknown 0 |
| workbook.xml | 10.3 kB | XML docum... 0 |

Fig. 6. xl directory containing VBA project

are benign and malicious. Thus it was required to collect known benign and malicious samples to construct a training set. Malicious samples can be collected from various sources online. Here, malicious samples were exclusively collected from the website, *malshare.com*. This website exposes an API to end-users and maintains good classification of samples by YARA rules created by community members. Once samples were downloaded, they were verified as malicious by scanning with VirusTotal and any samples not deemed malicious by at least three vendors were discarded. Subsequent manual inspection of each sample was carried out to ensure its malicious nature. Collection of benign samples was more challenging. To create a benign training set, various VBA tutorial websites offering example macros were consulted. In addition, Google searches for *filetype:doc* and *filetype:xls* with keywords such as *VBA* and *macros*, revealed more samples. Again, this set was further validated by scanning each sample on VirusTotal and excluding any sample that deemed malicious by any detector. Subsequent manual inspection of each sample ensured it was indeed benign. From each sample in the training set, all features are extracted and used to train a classifier. Once trained, the model is serialized to a file on disk. This enables the persistence of already trained models, and allows one to swap with ease between classifier models.

6 Feature Selection

In previously published literature, features typically selected are ones that arise purely from obfuscation. Thus, high detection rates can be achieved because the vast majority of malicious macros are obfuscated. However such an approach can lead to an increase in false positives. False positives are likely to occur in benign macros where the author has taken measures to obfuscate the source code. To handle such cases, a new approach is proposed here that includes the presence of social engineering tricks as an additional feature as these will almost never occur in a benign obfuscated macro. To automate the detection of such a feature, it was necessary to first extract images embedded within the document. As mentioned previously, such images contain text that aims to social engineer

the user into executing the macro. Generally speaking, commercial and open source tools cannot extract embedded images the reason being that Microsoft does not document how the Office suite, specifically Word and Excel, process embedded images. For Powerpoint files, images are stored within a *Pictures* stream, however this is not the case for Word and Excel.

As a result, an image extractor that does not rely on parsing the OLE header was implemented. Rather, it searches the OLE binary for potential image magic numbers. Once an image magic number is found within the OLE binary, the image header is parsed and the entire image extracted. Next, images are passed through the *tesseract* Optical Character Recognition library. Text is thus extracted from from these embedded images. A binary feature codes the presence of common phrases found in social engineering images, such as *Enable Content* and *Previous Version*. For features that code characteristics of the source code, a small custom VBA parser was implemented. This facilitated the simple incorporation of new source code-related features. Features found to achieve a maximum detection rate after hyper-parameter tuning are documented below:

Average Variable Assignment Length: It was found that many obfuscated macros declared abnormally long string variables. This was determined by comparing the average length for both benign and malicious sets. To capture this feature, the average length of string variables in the VBA source was computed.

Count of Integer Variables: Another characteristic common to the malicious macro set was that they defined more integer variables than the benign set. To capture this feature, the count of integer variables in the macro source code divided by the length of the source code was computed.

Count of String Variables: The same finding held for string as for integer variables. To capture this feature, the count of string variables in the macro source code divided by the length of the source code was computed.

Macro Keywords: This is a binary feature that encodes the presence of certain keywords that were found to be significantly more prevalent amongst the malicious set. These keywords related to event-based subroutines described earlier, namely *AutoOpen*, *AutoClose*, *DocumentOpen* and *DocumentClose*.

Highest Number of Consecutive Mathematical Operations: A subset of the malicious sample set employed anti-analysis techniques by declaring many variables in the following way:

Declaring variables in this fashion is characteristic of obfuscated scripts. This feature was captured by finding the highest number of consecutive mathematical operations carried out across all variables declared in the script. This integer value was then included in the feature vector.


```

CZpDLwLzJCC = "bYDG3XARdiw/ywW7EomCgGbg6wUWhjIdxYLkafFH0AB3L7GVHkeIkyKx"
KZiNmt = Left(Right(CZpDLwLzJCC, 54), 9) + CStr(Left(Right(CZpDLwLzJCC, 27), 10))
qLBHW = uobizHYvWuJ + CSng(179040) + 155196 / Sin(692027 - CByte(241113) / 553143
- Round(179040)) + DrofAzRh * GnBzqw - (155196 + 692027 + 241113 - 1790400)
Set CvPknLpXB = aUABbZz
BzHFJBBmPk = Chr(43)
FVjQqa = "Eeb"
tsFIXY0jkw = Left(Right(FVjQqa, 3), 1)

```

Fig. 7. Consecutive mathematical operations

Casing Ratio in Variable Declarations: Typically a programmer adopts a naming convention when it comes to variables. They may use lower-case exclusively, upper-case or camel case. In any case, one would expect the ratio of upper-case to lower-case characters in variable names to be either close to zero or significantly greater than one. To capture this feature, the latter ratio is computed for each variable in the script and averaged over the whole script.

```

NBWZB = 1347 - 906 - 386
eCjAai = UCase("tg.(GHBU-ABEa__")
BugTKjgp = 836 + 1041 + 1957
BugTKjgp = UCase("HRYhIbFteGcdR0#tW ")
NBWZB = UCase("HJMEuiSQmn*om")
hNoap = RTrim("F]HiZ%W vRt)?iv.OsNc")
eCjAai = RTrim("OdozFUXyXimw")
nQOid = 1751 - 902 - 1471
ZnGknBCz = RTrim("WrvN$KaHmMvLUvs.w(")
uBUZeFx = Space(8)
hNoap = StrReverse("W@PYdIXV&U")
hNoap = StrReverse("ni0%!0n*fxRAJ@ Q^?T")
uBUZeFx = 580 + 1528 + 1677
TmKMRPD = 765 + 1956 + 1343
JLQtw = 1798 + 1033 + 1689
DdFFiBCF = LTrim("oBUieEa-YH#A")
uBUZeFx = 1202 - 1516 - 1610
eCjAai = LTrim("!&_Q@sqZrUZmV FNey")
uBUZeFx = RTrim("pH*Ekp?^Z?")
NBWZB = Left("yUEK)NMUjw(EL#)", 5)

```

Fig. 8. Mixed Casing

Count of Variables: Obfuscated scripts tend to declare many more distinct variables in an effort to make analysis more difficult. To capture this feature, the number of distinct variable declarations in the script divided by the length of the script is computed.

Shannon Entropy: Obfuscated scripts tend to be less readable, using random, non-readable words for variable declarations and subroutine names. As a result, they tend to have higher entropy than non-obfuscated scripts. This value was added to the feature vector.

7 Classification

For each training sample, an N dimensional feature vector is created where N is the number of features used. Feature vectors are represented in a numpy array and passed to a classifier implemented by the scikit-learn Python library. Five different classifiers from scikit-learn were investigated: *KNeighborsClassifier*, *DecisionTree*, *RandomForest* and *GaussianNB*. Each classifier was trained over the training set consisting of 200 known benign samples and 200 known malicious samples. To improve classifier performance, hyper-parameters were first tuned using a randomised grid search using 10-fold cross validation. This narrowed down the ranges for which hyper-parameters were most performant. Next, another grid search was performed using 10-fold cross validation but defining more narrow parameter ranges that were close to those determined most effective by the randomised search. The `sklearn.model_selection.RandomizedSearchCV` and `sklearn.model_selection.GridSearchCV` classes were applied for this purpose.

Once the most effective hyper-parameters had been determined for each classifier, the classifiers' predictive capabilities were evaluated using samples from a separate test set. The test set consisted of 528 known malicious samples and 83 known benign samples. The samples in the test set were not present in the training set. Similar to Rudd et al [13], care was taken to ensure that all samples present in the malicious test set were newer than the newest sample in the malicious training set. The aim here is to evaluate the classifiers by using *deployment performance* metric. The latter captures the effectiveness of the classifier after deployment in terms of classifying new and previously unseen malware. This is important because of the evolving nature of obfuscated malware. For example, in any given week a malware campaign will favour one packer/obfuscator, but once anti-virus signatures catch-up, a switch to a new packer/obfuscator evades detection. To evaluate their performance, the True Positive Rate (TPR) was defined to be the percentage of the test set that was classified correctly and the False Positive Rate to be the percentage of the test set that was classified incorrectly.

Hyper-parameters, TPR/FPR are shown below. No hyper-parameter tuning was required for GaussianNB as the classifier doesn't accept parameters.

| KNeighbors | DecisionTree | RandomForest | GaussianNB |
|-------------------------|-----------------------|-----------------------|-------------------|
| n_neighbors = 20 | min_samples_split = 6 | min_samples_leaf = 2 | |
| weights = 'distance' | criterion = 'entropy' | n_estimators = 800 | |
| algorithm = 'ball_tree' | max_depth = 100 | max_features = 'sqrt' | |
| | | {max_depth = 50 | |

The table below documents my results for TPR and FPR against my test set with the RandomForest classifier being the best performer. The time row shows the average time taken to train the classifier on an i7-4790K CPU @ 4.00GHz.

| | RandomForest | KNeighbors | DecisionTree | GaussianNB |
|------|--------------|------------|--------------|------------|
| TPR | 98.9875% | 97.527% | 98.225% | 97.02% |
| FPR | 1.07% | 2.46% | 1.768% | 2.97% |
| time | 7m 34s | 6m51s | 6m51s | 6m59 |

8 Conclusion

Despite the relatively small training set, the observed performance is promising and on a par with other studies. This research suggests that machine learning may provide viable approaches to detecting VBA macro malware. Furthermore, in theory an approach such as that described in this paper can be easily ported to classify other prevalent script-based threats such as JavaScript, VBScript, Powershell and Windows batch files.

In terms of deployment, this approach could serve as either a host-based agent where document files are scanned prior to opening in Office, or a network-based agent where all document files on the network are inspected prior to reaching their destination. Detection logic could be updated by drawing the persistent model files from a remote location, analogous to how anti-virus products update their signatures. One challenge faced throughout this research was acquiring benign samples. Unlike malicious samples, there were no online resources identified that hosted verifiably benign macro-enabled documents. This significantly restricted training set size, and as a result impacted overall classifier performance. In an ideal scenario, access to Google's VirusTotal service would enable the ready download of benign samples. However, this service comes with a major annual price tag.

Future work on will be focus on implementing an online learning based solution. Such an approach, would not require re-training against the entire training set each time a new sample is added. This would make the software more suitable for deployment in a corporate network environment, with the ultimate intention of it consuming all Office samples traversing the network, and updating its classification model as it sees them.

References

1. Maria Wenzel, Matt Hoffman et al *Calling Windows APIs (Visual Basic)*. <https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/com-interop/walkthrough-calling-windows-apis>

2. Zhengchuan Xu, Qing Hu, Chenghong Zhang *Why computer talents become computer hackers* Communications of the ACM. Volume 56 Issue 4, April 2013. Pages 64-74
3. Vesselin Bontchev *Possible macro virus attacks and how to prevent them* Computers & Security. Volume 15, Issue 7, 1996, Pages 595-626
4. Sara L.N. RaId, Jens M. Pedersen *An Updated Taxonomy for Characterizing Hackers According to Their Threat Properties* 2012 14th International Conference on Advanced Communication Technology (ICACT)
5. *Macro Security for Microsoft Office* <https://www.ncsc.gov.uk/guidance/macro-security-microsoft-office>
6. Haifei Li *RTF Attack Takes Advantage of Multiple Exploits* <https://securingtomorrow.mcafee.com/mcafee-labs/rtf-attack-takes-advantage-of-multiple-exploits/>
7. Graham Chantry *CVE-2012-0158: Anatomy of a prolific exploit* <https://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/CVE-2012-0158-An-Anatomy-of-a-Prolific-Exploit.PDF>
8. Genwei Jiang, Rahul Mohandas, Jonathan Leathery, Alex Berry, Lennard Galang *CVE-2017-0199: In the Wild Attacks Leveraging HTA Handler* <https://www.fireeye.com/blog/threat-research/2017/04/cve-2017-0199-hta-handler.html>
9. Evan Gaustad *Applied Machine Learning: Defeating Modern Malicious Documents* https://www.rsaconference.com/writable/presentations/file_upload/ht-w02-applied-machine-learning-defeating-modern-malicious-documents.pdf
10. Gabor Szappanos *VBA is not dead!* <https://www.virusbulletin.com/virusbulletin/2014/07/vba-not-dead>
11. John Austin *Auto Macros* <https://msdn.microsoft.com/en-us/vba/word-vba/articles/auto-macros>
12. Nir Nissim, Aviad Cohen, Yuval Elovici *ALDOCX: Detection of Unknown Malicious Microsoft Office Documents Using Designated Active Learning Methods Based on New Structural Feature Extraction Methodology* *IEEE Transactions on Information Forensics and Security* vol. 12, no. 3, pp. 631-646, March 2017.
13. Ethan M. Rudd, Richard Harang, and Joshua Saxe *MEADE: Towards a Malicious Email Attachment Detection Engine* *arXiv preprint arXiv:1804.08162, 2018*
14. Philippe Lagadec *OpenDocument and Open XML security (OpenOffice.org and MS Office 2007)* *Journal in Computer Virology* May 2008, Volume 4, Issue 2, pp 115-125
15. Zhangjie Fu, Xingming Sun, and Jie Xi *Digital Forensics of Microsoft Office 2007/2013 Documents to Prevent Covert Communication* *JOURNAL OF COMMUNICATIONS AND NETWORKS, VOL. 17, NO. 5, OCTOBER 2015* pp 525-533
16. [https://msdn.microsoft.com/en-us/library/dd923471\(v=office.12\).aspx](https://msdn.microsoft.com/en-us/library/dd923471(v=office.12).aspx)
17. <https://github.com/decalage2/oletools/wiki>
18. <https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf>
19. <https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-government-en.pdf>
20. <https://nakedsecurity.sophos.com/2017/05/31/wolf-in-sheeps-clothing-a-sophoslabs-investigation-into-delivering-malware-via-vba/>
21. <https://blogs.cisco.com/security/attackers-slipping-past-corporate-defenses-with-macros-and-cloud-hosting>
22. <https://securelist.com/dridex-a-history-of-evolution/78531/>