Be Careful of What You Embed: Demystifying OLE Vulnerabilities

Yunpeng Tian[†‡], Feng Dong[†‡], Haoyi Liu[†], Meng Xu[§], Zhiniang Peng[†¶*], Zesen Ye[¶], Shenghui Li[†], Xiapu Luo[‖], and Haoyu Wang[†]
[†]Huazhong University of Science and Technology[**], [§]University of Waterloo
[¶]Sangfor Technologies Inc., [‖]The Hong Kong Polytechnic University
[†]{siontian, dongfeng, liuhaoyi, lishenghui, haoyuwang}@hust.edu.cn
[§]meng.xu.cs@uwaterloo.ca, [¶]{jiushigujiu, whhhitc}@gmail.com, [‖]csxluo@comp.polyu.edu.hk

***Abstract*—Microsoft Office is a comprehensive suite of productivity tools and Object Linking & Embedding (OLE) is a specification that standardizes the linking and embedding of a diverse set of objects across different applications. OLE facilitates data interchange and streamlines user experience when dealing with composite documents (e.g., an embedded Excel sheet in a Word document). However, inherent security weaknesses within the design of OLE present risks, as the design of OLE inherently blurs the trust boundary between first-party and third-party code, which may lead to unintended library loading and parsing vulnerabilities which could be exploited by malicious actors. Addressing this issue, this paper introduces OLExplore, a novel tool designed for security assessment of Office OLE objects. With an in-depth examination of historical OLE vulnerabilities, we have identified three key categories of vulnerabilities and subjected them to dynamic analysis and verification. Our evaluation of various Windows operating system versions has led to the discovery of 26 confirmed vulnerabilities, with 17 assigned CVE numbers that all have remote code execution potential.**

I. Introduction

Object Linking and Embedding (OLE), developed by Microsoft, is a proprietary technology that facilitates the inclusion of a diverse set of objects within the same document. OLE enables the creation of compound documents that host an array of data types, allowing an amalgamation of text, audio, graphics, spreadsheets, and various other application elements into a single cohesive unit. For instance, an Excel worksheet embedded in a Word document allows users to seamlessly edit the embedded data through an in-place activation within the process group of the Word application itself instead of launching the Excel application.

Despite its widespread usage, the design of OLE inherently harbors vulnerabilities as it blurs the trust boundary between first-party code and third-party code in processing the same document. In fact, a multitude of exploits based on OLE vulnerabilities have been disclosed over the past few years with a vast majority leading to arbitrary code execution [1], [2]. Unsurprisingly, such exploits have been harvested by Advanced Persistent Threat (APT) groups for phishing and single-click attacks [3].

This paper is dedicated to the detection of vulnerabilities in OLE objects—more specifically—embedded objects. To the best of our knowledge, this is the first academic work on analyzing OLE systematically for security weaknesses. More importantly, we note that OLE could represent a typical architecture for large and complex software bundles with extensible functionalities (e.g., Office or super-apps [4] such as WeChat and Alipay)—a type of software we call *super software* (or *superware* for short). Therefore, we hope the paper can provide a train of thought on finding bugs in closed-source superware through a practical case study on OLE.

Superware is characterized by its architecture which comprises of a core host application for foundational operations and a *potentially unbounded* list of dynamically loadable modules for additional functionalities. While extensible and agile, this architecture poses some alarming features that make superware vulnerable:

1) Dynamic modules might be sourced from third-party providers, and yet, they are loaded into the same memory space shared with the host application (first-party) and other loadable modules.
2) User input controls which dynamic module(s) to load. For example, a spreadsheet object in a Word document dictates the Excel-related modules to be loaded.
3) User input may include foreign content that can be serialized and deserialized by an external module only. The host application, on the other hand, conducts minimal checking on the foreign content due to lack of semantics.

All these characteristics blur the trust boundary between first-party and third-party code in superware, which enlarges the attack surface and exacerbates consequences of an exploit. At the same time, they make user input for superware more powerful (and more complex) than it appears. Essentially, a Word document is more like an interpreted script than just static data from a language-theoretic security perspective [5].

[‡]Both authors contributed equally to this research.
[*] is the corresponding author.
[**] Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230547
www.ndss-symposium.org

Recent years have witnessed an explosion of vulnerability finding tools themed by fuzzing [6], [7], [8], [9], [10], symbolic execution [11], [12], [13], and concolic execution [14], [15], [16], [12]; including tools tailored to the Windows platform on which many programs are proprietary [17], [18], [19], [20], [21], [22], [23], [24], [25]. However, the closed-source nature of superware like OLE rules out most tools that require source code (e.g., whitebox fuzzing).

Greybox fuzzing, which does not strictly require source code, is not very effective in the OLE case either. For example, WinAFL [24], paired with DynamoRio [26], has been successful in fuzzing Windows binaries. However, it encounters significant difficulties in handling complex software like Office, for at least two reasons: 1) For formats that are highly structured, blob mutation is often less effective, and most random inputs fail at the sanity checking phase; and 2) DynamoRio's support on applications with graphical user interfaces is limited.

Extracting entry points via reverse engineering is possible but impractical facing hundreds or potentially thousands of OLEs available on the Windows system. Furthermore, each OLE is unique in its data processing logic. Effective fuzzing of an OLE module requires an extensive reverse engineering on its input format—another scalability blocker. As a result, prior fuzzing effort on Office has largely focused on file mutation and generation [27], [28], [29] for the host application and less on the loadable OLE modules.

Zooming out with an abstraction, the reasons why OLE-related vulnerabilities are hard to find with conventional tools might also apply to other superware. In particular:

1) The potentially unbounded list of loadable module candidates implies that static analysis techniques are inherently prone to incompleteness in code discovery.
2) Naive fuzzing via random mutation of the entire user input (e.g., an entire *.docx* file) is futile while structured or type-aware fuzzing [25] requires detailed specifications of input format which are complex and hard to reverse engineer.
3) Loadable modules can be difficult to isolate for execution (e.g., fuzzing with harness [30]), as running a loadable module typically requires an integrated environment, including the host application and potentially other modules.
4) Other complications of superware, such as the closed-source nature and scarcity of internal documentation further amplify the difficulty of conducting thorough analyses.

The significance of these challenges is exemplified by Office and OLE. Hence, investigating security weaknesses via the lens of OLE serves as a case study for how to find vulnerabilities practically and systematically in superware.

Overview of OLEXPLORE. OLEXPLORE is our attempt to find OLE-related vulnerabilities systematically. OLEXPLORE consists of five components:

- A *high-level input layout* that loosely maps regions in user input to their handlers (e.g., the host application or dynamically loaded modules). Obtaining this layout requires a modest degree of reverse engineering on Office.
- A *misaligned feature* that enables the loading and initialization of OLE components without GUI interactions. This avoids the need for reverse engineering and simulating GUI interactions for each OLE component.
- A *micro-snapshot fuzzer* that can efficiently fuzz each OLE component without reverse engineering the underlying input format required by each OLE component.
- A *bug oracle* that scrutinizes the module loading process and memory operations, vigilantly searching for indications of vulnerabilities. The spectrum of monitored vulnerabilities is meticulously curated, informed by the OLE attack surface and a compendium of historical CVE records.
- A *vulnerability analyzer* that assesses the severity of identified vulnerabilities reported by the bug oracle. We use a disguised RTF as the carrier for malicious attacks and bypass the Protected View Mode [31] by a separate crash-inducing bug. This is a weaponization technique that has not been previously disclosed to the public.

While the design principle of OLEXPLORE is universal, as detailed discussions in Section VI will show, porting it to other superware (e.g., Alipay or WeChat) still requires manual and even reverse engineering efforts, including:

- Identifying a feature that triggers the loading and execution of loadable modules without GUI.
- Identifying the input-consuming function used for chunk-by-chunk input processing (for micro-snapshot fuzzing).

The bug oracle developed in OLEXPLORE can be reused across different superwares, as the principles of scrutinizing the module loading process and memory operations, and vigilantly searching for indications of vulnerabilities, are commonly applicable in these large software systems. This facilitates some level of automation, reducing the need for extensive manual customization.

Summary. OLEXPLORE makes the following contributions:

- We conducted an exhaustive survey on all publicly disclosed OLE-related vulnerabilities and delved into the fundamental causes of these flaws. Building on the outcomes of our investigation, we identified current attack surfaces for OLE and employed three vulnerability patterns to evaluate the exploitability of OLE (Section III).
- We developed OLEXPLORE[1], a pioneering tool for systematic vulnerability detection in OLE components. Novel techniques proposed in OLEXPLORE include 1) GUI-interaction bypassing, 2) micro-snapshot fuzzing, and 3) a vulnerability weaponization technique to bypass Office Protected View Mode. We have privately disclosed the weaponization technique to Microsoft and this paper marks the first public disclosure of these exploits (Section IV).
- We systematically analyzed all registered COM components (a superset of OLE components) in popular Windows platforms, identified 257 OLE components, and reported 26 bugs. Out of those, 17 vulnerabilities have been assigned

[1]We open source OLEXPLORE at https://github.com/WinSysSec/OLExplore to facilitate the research in this area.

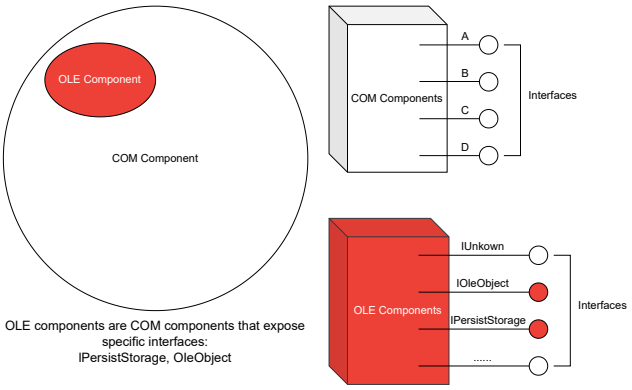


Fig. 1: OLE components are special COM components

CVE numbers, each with the potential of Remote Code Execution (Section V).

We hope OLEXPLORE can be inspirational for finding vulnerabilities in other superware and we endeavor to enhance the understanding and discovery of potential security gaps in complex software built upon dynamic library integration.

II. BACKGROUND

In this section, we present background information about OLE and illustrate OLE vulnerabilities with a real-world case.

A. Basics of OLE

This paper focuses on OLE 2.0, the currently in-use version. At its core, OLE 2.0 is built on the Component Object Model (COM) and structured storage, and OLE objects are essentially COM objects that can be embedded or linked to existing documents. Figure 1 shows the relationship between OLE and COM. The vision for OLE, and more fundamentally COM, is to enable software developers to create more streamlined and focused applications. Should developers opt to extend services beyond core functionalities, such add-ons could be implemented as separate modules which can be loaded into memory only when required.

1) COM interfaces: As COM components, all OLE components expose the `IUnknown` interface, which facilitates its clients in discovering other interface pointers, including those offering specific OLE functionalities such as `IOleObject`, `IOleLink`, and `IViewObject2`.

2) Structured storage: All OLE components implement the `IPersistStorage` interface for state persistence and optionally `IPersistStream`. `IPersistStorage` defines standard methods for persisting object states to external storage in a way that is agnostic to any specific storage backend. The counterpart, i.e., the host application that processes documents with OLE objects embedded, must hence provide the `IStorage` interface to enable object data storage and retrieval. The format and semantics of persisted data, however, is completely oblivious to the host application. As different OLE components offer different implementations for `IPersistStorage`, the format of data persisted in the storage also varies significantly.

Furthermore, based on how an OLE object is hosted at runtime, OLE can be further categorized into two types:

In-process OLE is an OLE component that executes in the same process as the host application. In-process OLE can be initialized through `ole32!OleLoad()`.

Separated-process OLE is an OLE component that executes in a separate and standalone process. Separated-process OLE are instantiated by `ole32!OleRun()`.

Within OLE, objects are primarily categorized into two types: *embedded objects* and *linked objects*. Embedded objects are self-contained objects with all related information stored in the host document. For instance, users can use the "Insert Object" feature to embed a Word document (with all of its content) directly into an Excel worksheet [32]. With this, users can view and edit the Word document without opening the Word application. Conversely, a linked object references an object located in a different file or application. The OLE object in the host document simply maintains a link rather than enclosing the complete data. As a result, updates to the original object are reflected in the linked object accordingly.

This paper focuses on *in-process embedded objects* which draw most of the security concerns (Appendix A).

B. How OLE Works

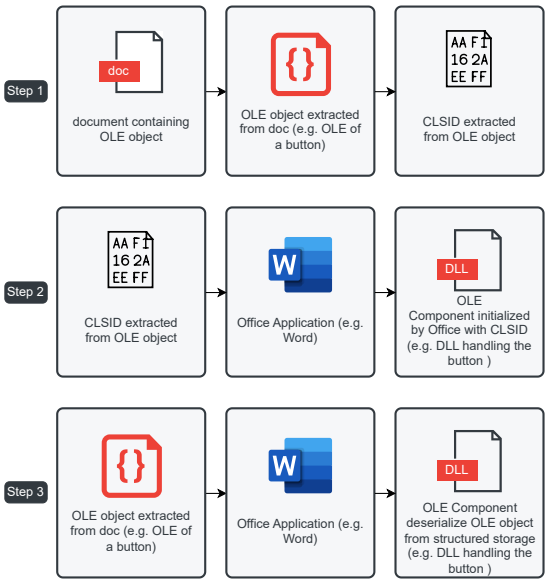


Fig. 2: Loading process of OLE objects with Office

As shown in Figure 2, loading an OLE object within Microsoft Office involves a series of interactions between various entities, out of which three are especially relevant here.

Step 1: Retrieve the CLSID from the document. In this phase, the host application extracts a class identifier (CLSID) from the document to determine which OLE module needs to be dynamically loaded and initialized. CLSIDs can be embedded in a wide range of document formats, including:

- Office binary formats (e.g., .doc, .xls, .ppt, .pps, etc.)
- Office OpenXML formats (e.g., .docx, .xlsx, .pptx, etc.)
- RTF format (.rtf)

The mechanisms for locating the embedded CLSIDs differ by file format. Taking the OpenXML format as an example,

the embedded CLSIDs reside in a form named `OLESSFormat` stored within the root directory entry, which can be inspected and extracted using the OffVis tool [33].

Step 2: Invoke `CoCreateInstance` to load the module. Creating an OLE object in the current execution context requires initializing the OLE module first via the `CoCreateInstance` function. This process involves loading the DLL associated with the CLSID into the current process. The Registry holds a mapping between CLSID to its corresponding DLL under the key `HKEY_CLASSES_ROOT\CLSID`, which ensures that the system can accurately identify and load the requested DLL to unblock subsequent operations. After the DLL is loaded, `CoCreateInstance` not only ensures the module is in memory but also triggers the execution of initialization code. Then, a designated entry function within the loaded DLL is invoked to complete the setup process.

Step 3: Invoke `IPersistStorage::Load` to deserialize the OLE object. The `IPersistStorage` interface includes several methods, among which `Save` and `Load` are the most critical, as they are responsible for serializing and deserializing the persistent states of an object, respectively. Once the DLL for an OLE object is ready, the DLL invokes the `IPersistStorage::Load` method to deserialize an OLE object from the structured storage. The `Load` function takes an `IStorage *pStg` argument—a pointer to the actual location of the OLE storage within the document—supplied by the host application. Since any data embedded in the document can be subject to attackers' control, corrupted OLE storage serves as the primary way to carry out exploits (such as memory corruption and logical errors).

C. OLE Structured Storage

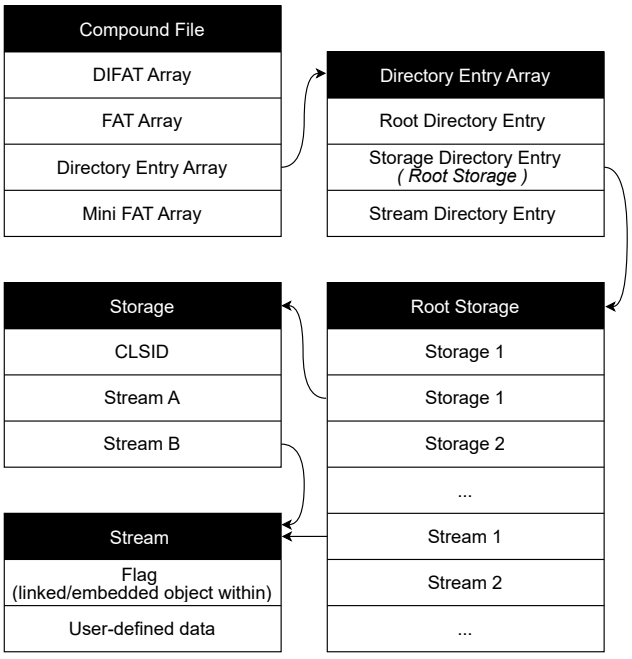


Fig. 3: A sketch of the format of a compound document with OLE objects embedded

Structured storage addresses the problem of storing heterogeneous data objects within a single file in a way that is transparent to the code (e.g., both the host application and OLE DLLs) that operates on these data objects.

More specifically, structured storage introduces an abstraction named "stream", realized through the `IStream` interface, and resembles traditional files which provide read and write methods. Enclosing streams are an abstraction named "storage", described by the `IStorage` interface, and is analogous to directories in a filesystem. Naturally, a storage can contain both streams and more storages.

A compound document amalgamates a diverse set of data types such as text, graphics, spreadsheets, audio, and video, all of which are embedded within the document as independent and self-contained objects. To persist multiple objects into a single file, one can open an `IStorage` representing the file's content and save each object in a separate `IStream`. Figure 3 shows a sketch of the scheme of storing OLE objects in a compound document file. While the host application is responsible for managing the root-level `IStorage`, the specifics of how to read and write an `IStream` are delegated to each OLE DLL which is typically encoded in the implementation of the `IPersistStream` interface. Given the varied implementations of `IPersistStream` by different OLE DLLs, the data formats vary significantly, and fuzz testing these formats generally requires manual reverse engineering of the inner structure of each `IStream` to build a harness.

III. OLE Attack Vectors

In this section, we categorize OLE vulnerabilities based on a comprehensive review of all known OLE-related CVEs. In particular, we surveyed 21 OLE-related CVEs and discovered that they can be classified into three types. Combined, these OLE-related CVEs account for 43.23% of CVEs targeting Office (see Appendix A for more details).

A. Type-1: Loading a COM Component Not Intended for OLE

Type-1 CVEs account for 3 out of the 21 surveyed CVEs. In Office, a CLSID serves as an index for the OLE component that a user wishes to initialize. Upon recognizing a CLSID in the input document, the host application calls `CoCreateInstance` to load the corresponding DLL and trigger certain initialization functions. During this process, memory is allocated, and the corresponding objects are instantiated.

However, not all components associated with a CLSID are OLE components. With thousands of CLSIDs existing in the system, only a subset corresponds to actual OLE components. This means that some COM components not intended for use in Office might still be loaded as OLE components. While Office already performs a series of complicated checks before loading a COM component to ensure it is intended for OLE, our evaluation shows that existing checks are inadequate. Furthermore, if an unintended COM component fails to initialize its memory appropriately and fails to report exceptions, it can expose vulnerabilities such as uninitialized read.

```
CoCreateInstance (
  rclsid =
  ↪ {CDDBCC7C-BE18-4A58-9CBF-D62A012272CE},
  pUnkOuter = NULL,
  dwClsContext = CLSCTX_INPROC_SERVER |
  ↪ CLSCTX_INPROC_HANDLER,
  riid =
  ↪ {00000000-0000-0000-C000-000000000046},
  ppv = lpTargetPpv
)
```

Fig. 4: In CVE-2015-1770, a COM instance is created by calling CoCreateInstance.

In summary, a Type-1 vulnerability can be triggered by 1) placing a non-OLE CLSID in the input document, 2) utilizing the content succeeding `objdata` within the RTF trigger, as delineated in Section IV-B, to populate the storage for objects of this particular CLSID type, and 3) loading the document which might automatically trigger `CoCreateInstance` to run to load and initialize the fictitious "OLE" component.

CVE-2015-1770 is a typical example of Type-1 vulnerability. The root cause of this CVE stems from the improper initialization of the COM component `OSF.DLL`. Despite having a CLSID, `OSF.DLL` is actually not architecturally intended to function as a loadable OLE component. More specifically, `CoCreateInstance` on this component (identified by the `rclsid` argument as shown in Figure 4) crashes unexpectedly with page-heap enabled. The crash can be traced back to the interface pointer `lpTargetPpv`, which is obtained during the incorrect process of loading `OSF.DLL` as an OLE component.

Reverse engineering reveals that the pointer `lpTargetPpv` is a 184-byte object created and partially initialized by two subroutines: `PpvPartOneInitialization` and `PpvPartTwoInitialization`. At offset `0x54`, it contains a pointer (`lpTargetPpv->pUnkObj`) to another 116-byte object, allocated and partially initialized by `PpvOffset54UnkObjAllocAndInit` and `PpvOffset54UnkObjInit`. A field at offset `0x58` within this new object remains uninitialized, causing a crash under PageHeap when accessed.

B. Type-2: DLL Preloading Attacks

Type-2 CVEs account for 11 out of the 21 surveyed CVEs. When the `CoCreateInstance` function is called, if there is a DLL associated with the specified CLSID, the `LoadLibrary` function will be invoked. If the DLL is requested while the Registry does not hold a complete path for the DLL, Windows searches for the required library according to a predefined sequence of directories (i.e., DLL search order). An attacker who gains control over any one of these directories can force the program to load a malicious DLL, thus supplanting the legitimate DLL that should have been loaded. This is typically known as *DLL preloading attack*, and represents a pervasive security issue for all platforms that support dynamic library loading. Exploiting this vulnerability, an attacker can execute arbitrary code in the host application's privileges [34].

Attacks based on `LoadLibrary`. The following outlines the DLL search order for `LoadLibrary` and `LoadLibraryEx`, two commonly used functions for dynamic DLL loading:

1) The directory from which the application loaded
2) The system directory
3) The 16-bit system directory
4) The Windows directory
5) The current working directory
6) The directories listed in the `PATH` environment variable

Attacks based on `SearchPath`. There are comparable risks when programs use the `SearchPath` API to locate and dynamically load a DLL from a path returned by `SearchPath`. The following exemplifies the default search order of the `SearchPath` API:

- An OLE component attempts to load `xxx.dll` from the current directory.
- An Office application, such as Word, not only searches for `xxx.dll` in the current directory but also loads the DLL into the Word process. An attacker could exploit this step by inserting a malicious DLL containing a command to run arbitrary code, thereby accomplishing the attack.

The Windows operating system exhibits a design vulnerability during the initialization of OLE components, wherein the loaded DLLs are not authenticated, stemming from a specific process analyzed from an attacker's perspective. An OLE object is a subset of COM objects, which are collections of objects associated with CLSIDs and registered within the operating system.

In certain instances, even if a DLL possesses an associated CLSID in the registry, it may not be a legitimate COM component. Despite this, `CoCreateInstance` continues to load such DLLs with CLSIDs into the process, without considering whether the DLL is genuinely related to OLE. Our primary concern herein lies with DLL preloading attacks. Given the significant variation in component installation across different Windows versions, some, such as Windows Server, do not come with certain system functional components pre-installed. As a result, on these variants, even if a CLSID exists, the corresponding DLL may not, leading to path searching for the DLL. At this juncture, should an attacker provide a malicious DLL in the anticipated path, it would be loaded, and the malicious behavior executed. This reality constitutes a design flaw during the initialization of OLE objects in Windows.

C. Type-3: OLE Data Parsing Error in `IPersistStorage`

Type-3 CVEs account for 7 out of the 21 surveyed CVEs. Data parsing is always a hotspot of vulnerabilities. As the input data is inherently untrusted and maliciously crafted data can cause both memory errors and logic bugs. An analysis of disclosed CVEs reveals that a majority of vulnerabilities related to OLE are closely related to the implementation of `IPersistStorage::Load`, which is responsible for loading objects stored within the storage section in the input document.

Typically, data in the storage section is in binary format. Moreover, the storage format for each OLE component varies

and the format is within total control of the associated OLE itself. Due to its proprietary and intricate nature, binary storage might accidentally conceal security exploits (i.e., security through obscurity). Attackers can exploit specially crafted binary storage to trigger specific vulnerabilities and execute attacks. Consequently, systems must enforce stringent data validation and security checks when handling storage data from untrusted sources to mitigate threats.

It is noteworthy that storage data originates from the `IStorage` parameter of function `IPersistStorage`. Since code executing the `IPersistStorage` interface resides within OLE components, how `IStorage` is handled depends on the OLE object itself. Storage data is embedded within document files, meaning that this parameter can be influenced by external factors. However, an attacker may tamper with the storage data contained in the OLE objects within a document. When Office suites act as OLE containers processing documents containing OLE objects, they load the corresponding OLE components and invoke the `Load()` method of the OLE component's `IPersistStorage` interface to parse the storage data and initialize the object's initial state. During this initialization, the data source remains unverified, thus in a potentially untrustworthy state, harboring latent security risks.

CVE-2017-11882 [35] is a notorious vulnerability related to OLE. The underlying mechanics of this vulnerability are not only straightforward but also robust, enabling exploitation without user interaction or detection. Several APT groups have added this vulnerability into their arsenal, extensively utilizing it in spear-phishing and watering hole attacks [36]. This vulnerability resides within the Equation Editor module `EQNEDT32.exe`, which was originally developed by Design Science Inc., compiled in 2000, and later maintained by Microsoft. Regrettably, Microsoft did not prioritize security for this module, resulting in the flaw going undetected for 17 years. When inserting or editing equations in Office documents, the host application invokes the `EQNEDT32.exe` process via RPC to handle equation parsing and editing. Although Office versions post-2007 replaced `EQNEDT32.exe` with a built-in equation editor, all versions of Microsoft Office (including Office 365) still support formulas initially created with `EQNEDT32.exe` for backward compatibility.

The vulnerability manifests during the processing of *Equation Native* data streams, often containing MathType formulas provided by the document. When parsing the font name, the length of the name within the stream lacks proper validation, leading to a stack overflow. An attacker could exploit this by crafting data to overwrite the function return address, hijacking the control flow of the application. Within the described overflow, after the return address has been overwritten, the contents of the string are transformed in the `_strupr` function whereby, following subtraction of `0x20` from the code, the return address is changed to the address of `WinExec`, thereby achieving the invocation of `WinExec`. Figure 5 outlines a typical case of exploitation.

Although both the CLSID and storage data can be supplied by an attacker, there is a distinction in how they are used. OLE

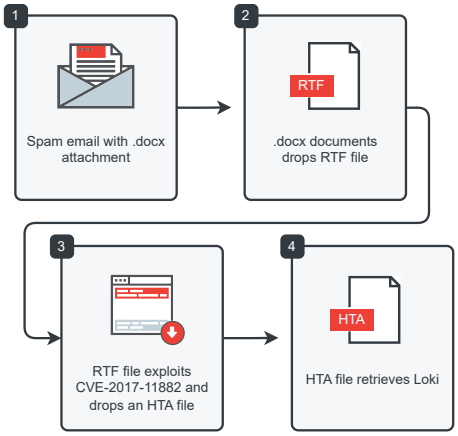


Fig. 5: A real-world exploitation.

containers like Office or WordPad primarily read the CLSID to initialize an OLE component. Subsequently, storage data is extracted from the OLE object embedded within the document and transmitted to the OLE component for subsequent processing. This delineates the fundamental difference between Type-1 and Type-3 vulnerabilities.

IV. OLExplore: Systematic Bug Finding for OLE

Figure 6 outlines the workflow of OLExplore. The primary objective of this tool is to facilitate the efficient and accurate detection of OLE-related vulnerabilities. Our system functions in the following main steps:

Phase 1: Analyzing OLE components. Initially, OLExplore identifies all CLSIDs present on the system It then enumerates the interfaces and properties exposed by each COM component to filter and identify OLE components, acquiring their pertinent information.

Phase 2: Constructing an OLE runtime. OLExplore uses a GUI-free RTF-based input template to trigger various OLE components. Upon encountering this trigger, Office will proceed to load the corresponding DLLs, thereby bypassing the need for GUI interaction and fostering the construction of an environment conducive to the instantiation of OLE objects within the system.

Phase 3: Fuzz OLE-specific storage segment. During this phase, for recognized formats, OLExplore utilizes ActiveX to generate the corresponding storage. In the case of unrecognized formats, OLExplore employs micro-snapshot fuzzing, a technique that enables efficient fuzz testing of each OLE component independently, thus obviating the need for reverse engineering the specific input format each OLE component necessitates.

Phase 4: Behavior Detection. Utilizing the established testing framework, batch tests are conducted and abnormal events such as DLL loading behavior as well as memory corruption crash are monitored and logged using Process Monitor [37].

Phase 5: Vulnerability Analysis. The analysis stage involves dissecting crash-captured dump files and validating vulnerability exploitability through constructed proof-of-concept (PoC)

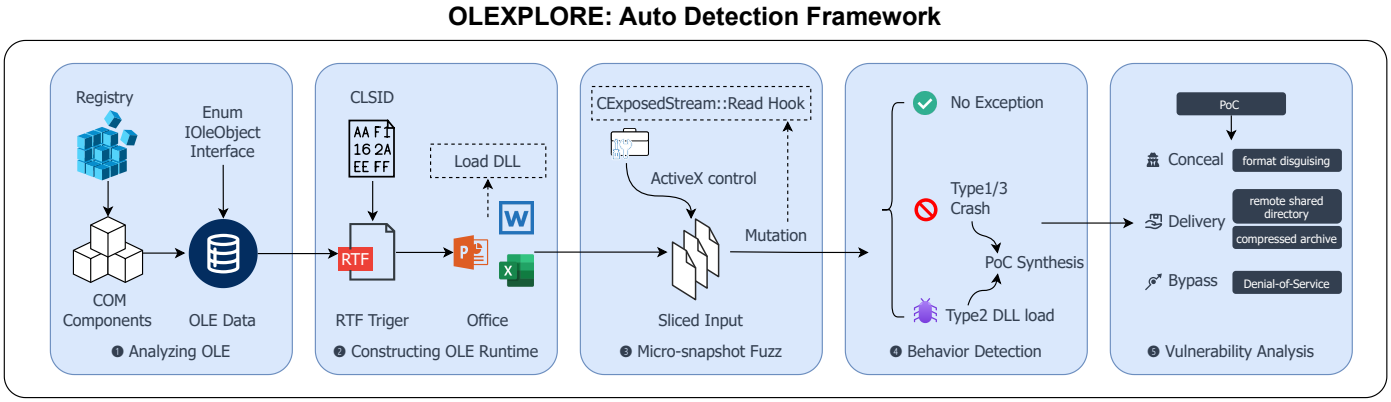


Fig. 6: Overview of OLEXPLORE

exploits. We employed a novel and not yet fully disclosed method of vulnerability weaponization to assess the severity of the vulnerabilities discovered.

A. Collecting OLE Components

OLEXPLORE collects registered COM components from Windows registry key `HKEY_CLASSES_ROOT\CLSID` by crawling unique CLSID identifiers and subsequently retrieve the filesystem location(s) of the associated DLL file by reading the `InprocServer32` and `LocalServer32` sub-keys.

As introduced in Section II-A, OLE components are a subset of COM components. Specifically, a COM component may be classified as an OLE if it implements the `IOleObject` interface. To ascertain whether a given COM component is an OLE component, one can systematically instantiate objects using the list of collected CLSIDs and then enumerate each COM component's exposed interfaces and properties to determine the implementation of the `IOleObject` interface.

The COM interfaces are declared using Microsoft Interface Definition Language (MIDL). PowerShell provides a `Get-Member` cmdlet [38] to list the interfaces and properties of a component. Upon confirming that the COM component offers the `IOleObject` interface, OLEXPLORE utilizes OleViewDotNet [39] to decompile the interface from the binary file and export the declaration.

B. Constructing an OLE Runtime Environment

After collecting all OLE components available in the system, OLEXPLORE needs to construct a runtime environment for each OLE component. To identify Type-1 and Type-2 vulnerabilities (i.e., those can manifest upon invoking the `CoCreateInstance` function), it is imperative to ask the host application to instantiate each OLE object, during which the host application will invoke the `CoCreateInstance` function.

Unfortunately, the conventional way of loading an OLE component requires non-trivial GUI interactions (1) such as clicking on the `Insert` tab, (2) clicking on `Object`, (3) browsing to the OLE file, and (4) clicking `OK` to confirm[2].

[2]See this Microsoft Office help manual (https://support.microsoft.com/en-us/office/embed-or-link-to-a-file-in-word-8d1a0ffd-956d-4368-887c-b374237b8d3a) for details on loading an OLE object.

```
{\rtf1
{\object\objemb{\*\objclass None}
{\*\oleclsid\
'7b00000000-0000-0000-0000-000000000000\'7d}
{\*\objdata
↪  01050000010000000100000000000000000000000000
00000000000000000000000000000000
00000000000000000000}}}
```

Fig. 7: An example of an RTF document that triggers the loading of an OLE object without requiring user interaction

It is not feasible to manually run the GUI interactions for all OLE components found by OLEXPLORE, which is at the scale of hundreds of not thousands.

To avoid GUI interactions in automated OLE component initialization, OLEXPLORE exploits a "feature-not-bug" in the current Office implementation. As reported by Steven Vittitoe in 2015 [40], the opening of the RTF document shown in Figure 7 results in the loading of OLE objects without user interaction. This issue was reported to Microsoft in 2016 [41], but Microsoft did not recognize it as a vulnerability. In contrast, when an OLE-embedded *.docx* file is opened, Office issues a warning prior to the activation of the object. For instance, Office 2016 emits a warning when opening a *.docx* file containing an OLE object. However, when the identical *.docx* file is saved in RTF format and later opened with Office 2016, no prompt appears for the embedded OLE object. This RTF template serves as a highly customizable base case because it provides a way to load and initialize an OLE component without manual interaction. We use the RTF loading trick to avoid graphical interactions and reduce performance overhead. Using an AutoIT script requires waiting for Office to fully initialize, including various UI windows. With PageHeap, this UI initialization becomes very slow. In contrast, RTF files bypass the UI constraints and can trigger OLE parsing without needing full initialization.

We produced RTFs for every OLE component found in the system. The structure of the framework is straightforward; it merely requires the creation of an RTF document with an embedded CLSID that resides in the system registry. This

approach circumvents the necessity to reverse-engineer the GUI interfaces and the initialization functions within the OLE component. It simulates user interactions such as opening and closing the document, as well as actions akin to clicking or double-clicking on the OLE object. This simulation facilitates the loading of the OLE object by Office, alongside the associated DLLs, thus triggering the OLE-specific initialization functions required for assembling and establishing the OLE runtime environment.

C. Testing An OLE Component by Mutating Storage Data

Testing the `IPersistStorage` interface requires storage data. However, gathering a relevant corpus presents considerable challenges due to the lack of previous research or public effort on this matter. In OLEXPLORE, for OLE components with matching ActiveX plugins, we utilize ActiveX to directly generate storage data. For other OLE components with unknown and customized storage formats, we employ micro-snapshot fuzzing to automatically probe an input format.

1) ActiveX-based input generation: To build an initial corpus for known storage formats, we developed 74 kinds of ActiveX controls (e.g., the `CheckBox Control` with `CLSID`: `8BD21D40-EC42-11CE-9E0D-00AA006002F3` is related to the module `FM20.DLL`). ActiveX controls, which are built on a foundation of many lower-level objects and interfaces in OLE [42], are a type of OLE object that can be embedded within documents such as Word or Excel files to provide interactive functionality.

These ActiveX controls exhibit a variety of attributes, such as `Font` and `Caption`, which are persistently stored within a designated storage structure. Office provides a suite of developer tools designed for the rapid creation of various ActiveX controls. By specifying the creation of controls such as "Text Box" or "Button" classes, we ensured that these controls contain unique storage formats related to their OLE components. Once generated, the ActiveX controls are encapsulated within `.docx` documents. By decompressing these documents, we can retrieve the corresponding binary files.

We generated batches of controls for different classes and randomly assigned properties to them in order to extract the storage data. This method allowed us to create a diverse and comprehensive dataset of storage formats, which is essential for thorough testing and analysis of OLE object vulnerabilities.

2) Micro-snapshot fuzzing: For OLE components that take unknown formats of storage and do not have ActiveX control support, a naive solution would be to treat the entire input stream as a blob and apply random mutations on the blob (e.g., basic techniques in AFL++ [8]). However, for formats that are highly structured, blob mutation is often less effective, and most random inputs fail at the sanity-checking phase. The ideal solution would be to reverse engineer the input formats and use grammar-based input generation or mutation to generate near-correct inputs [43] [44] [45]. However, this is not feasible facing hundreds if not thousands of OLE components that are close-sourced with little to no public documentation.

In OLEXPLORE, we develop a new technique, micro-snapshot fuzzing (see details in Section IV-D), to achieve better mutation efficiency than the plain blob mutation without reverse engineering effort on the OLE DLLs. Micro-snapshot fuzzing, as the name suggests, is inspired by snapshot fuzzing which has been proven effective in stateful systems such as network protocols [46] and operating system kernels [47]. In fuzzing stateful systems, the fuzzer takes a snapshot whenever an input event (e.g., one network message or one syscall) discovers new coverage. The snapshot inherently encompasses all previous events that lead the system to the current state, and allows the state to be further advanced into different possibilities when resumed and fed with different new events.

D. Micro-snapshot Fuzzing

Micro-snapshot fuzzing in OLEXPLORE divides the input blob (which is of an unknown format) into chunks where each chunk corresponds to an event in snapshot fuzzing. This is achieved with the following observation on how OLE components process input from storage.

- Input reading only occurs via one standard interface: `CExposedStream::Read(`
  - `stream_to_read`,
  - `receiving_buffer`,
  - `number_of_bytes_to_read`,
  - `number_of_bytes_actually_read`)

  (abbreviated as `S::Read` subsequently).
- Input is processed chunk-by-chunk, and each chunk is relatively small in size (compared with the size of the input stream)
- Input chunks are read sequentially and the OLE never re-reads a chunk.

With these observations, we note that one chunk of input stream is akin to one message in a network protocol or one syscall to the kernel. If one chunk of input yields new coverage, we can take a snapshot of the current process and save it to the seed pool; in future fuzzing rounds, if this seed is selected as base of mutation, the snapshot will be restored first and we focus on mutating the next chunk to further advance the state. However, different from snapshot fuzzing which actually snapshot the system state (e.g., a VM snapshot in kernel fuzzing), the snapshot in OLEXPLORE is simply an accumulation of input chunks that lead the execution of an OLE component into its current state. This is feasible because OLE components do not involve multi-threading in processing the inputs, hence, a deterministic sequence of events will lead to a deterministic state.

And yet, one question remains: how to break the input stream into chunks? Reverse engineering the format of the input stream naturally reveals the chunks but is not scalable. In OLEXPLORE, the chunk boundaries are decided by the OLE code logic and are passively collected by OLEXPLORE.

To be more specific, every time `S::Read` is invoked, OLEXPLORE identifies an input chunk by the amount of bytes read. The second parameter of `S::Read` is a buffer pointer

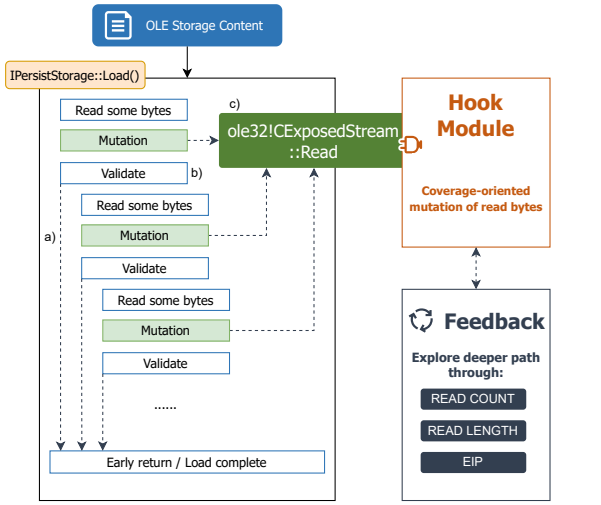
designated for storing the data that is read; the third parameter dictates the volume of data to be read; and the final parameter is an integer pointer that yields the actual number of bytes that have been read. For example, `S::Read(_, _, 20, _)` marks a 20-byte chunk; while `S::Read(_, _, 36, _)` marks a 36-byte chunk.

Snapshots are taken right before the invocation of `S::Read` if new coverage is discovered between the last `S::Read` and the upcoming `S::Read`. To illustrate the micro-snapshot fuzzing process in OLEXPLORE,

1) select seed s as mutation base from seed pool, s is essentially a snapshot of sequence [`S::Read(_, _, 8, _)`, `S::Read(_, _, 16, _)`, `S::Read(_, _, 12, _)`]
2) after resuming s, the next chunk is marked by `S::Read(_, _, 20, _)`, so OLEXPLORE will generate and mutate 20-byte chunks (e_1, e_2, ..., e_k) and feed to the resumed snapshot.
3) for each new event e_i, OLEXPLORE will monitor its execution until the execution hits the next `Read`. If new coverage is discovered, OLEXPLORE will save $s{+}{+}e_i$ as a new seed in the seed pool.

Furthermore, OLEXPLORE follows a systematic exploration of the input stream which is inductive in nature. More specifically:

- **Base case**: probe for one-chunk snapshots (by mutating the chunk marked by the first `S::Read`).
- **Induction case**: for each k-chunk snapshot:
  1) resume the snapshot
  2) produce (with randomness) the $(k+1)$-th chunk
  3) check whether the $(k+1)$-th chunk yields new coverage, if so, save it as $(k+1)$-chunk snapshot



a) Repeated reading and checking during OLE storage retrieval.
b) Terminate reading if byte check fails.
c) Retrieving storage for the OLE component ultimately entails calling ole32!CExposedStream::Read.

Fig. 8: Illustration of micro-snapshot fuzzing

Implementation details — Figure 8 demonstrates OLEXPLORE's micro-snapshot fuzzing workflow for the module under testing. In the testing process, the fuzzer tracks `CExposedStream::Read` (i.e., raw bytes read) using Detours [48] (which monitors and instruments API calls on Windows platforms). The fuzzer monitors the DLL execution until the next read is encountered. If new coverage is discovered between the last read and the upcoming read, a snapshot is taken. The next mutation will start from a snapshot. In other words, a snapshot is a seed (in fuzzing terms) in the seed pool.

Effectively, a snapshot encompasses all chunks (in raw bytes) that are previously read via `CExposedStream::Read` by the OLE component. Therefore, replaying these chunks essentially replays a prefix of the execution path in the DLL (modulo concurrency) and fuzzing continues based on this prefix. It is worth noting that while concurrency is heavily used in Office, parsing the storage for a single OLE object is, in fact, entirely sequential in our tested modules. Therefore, micro-snapshot fuzzing in OLEXPLORE is designed with the implicit assumption that input chunks are read and processed sequentially. Each chunk of raw bytes is either generated or mutated randomly due to the lack of type or structural information.

To measure and track coverage of code executing during fuzzing, the dynamically loaded DLL is instrumented with TinyInst [49]–a binary rewriter–when loaded. TinyInst recognizes basic blocks in the instruction sequence, and we hook the first instruction of every basic block for coverage tracking.

Taking CVE-2017-11882—a vulnerability discovered within the `IPersistStorage::Load` function—as a case study, our tool OLEXPLORE monitors the affected `EQNEDT32.exe` process. Within this vulnerable function, read operations are intercepted through hook functions specific to `S::Read`. The crux of the vulnerability lies in handling Equation Native Stream Data within the document. The program reads two structures in two phases: `EQNOLEFILEHDR` and `MTEFData` (Equation Native Stream Data consists of `EQNOLEFILEHDR` and `MTEFData`). In processing `MTEFData`, it invokes a subroutine to iteratively read the constituent record sub-structures, with field values dictating the function's execution path.

Initially, 8 bytes of data are read—which corresponds to the first 8 bytes of the Contents data block onto the stack via `S::Read`. Subsequently, the program validates the read data to ensure a match against specific data patterns, such as 0x12344321 and 0x08. Upon successful validation, the remainder of the Contents data block is read; otherwise, the process is terminated. Importantly, the data retrieval is not performed in a single operation but in batches, coupled with validation steps. A batch only proceeds with additional reads if its data passes validation until all data is processed.

E. Behavior Detection

When the `CoCreateInstance` function is invoked in the instantiation process of an OLE object, the system may load additional DLLs. To capture unintended DLL loading events, we utilized the Process Monitor [37]—a widely applied API behavior monitoring software on the Windows platform.

Through monitoring the `CreateFile` API, it is possible to detect whether any DLL file is being created or opened.

Should there be activity involving a DLL file at such time, OLEXPLORE performs a scan of its function call stack in search of the `LoadLibraryEx` API. The purpose behind searching for this API lies in the fact that the presence of `LoadLibraryEx` within the call stack of the DLL signifies that the DLL has been dynamically loaded by the process.

By employing the following filtering strategies, all DLL loading activities are meticulously captured and displayed. For DLLs that do not exist within the system, our approach involves monitoring instances where search results include the phrase *not found*. In exploitation scenarios analyzed, it has been observed that counterfeit DLL files can be loaded successfully without requiring any export functions. Notably, these incidents of successful DLL loading precede any internal errors that may occur within the process thereafter. For DLLs that are present within the system, OLEXPLORE examines them to identify externally accessible export functions. The export table enumerates every function the DLL offers to other executables—these serve as the DLL's entry points; only functions in this export table are available to other executable files. All other functions not listed remain encapsulated within the DLL, inaccessible from outside. To validate the security measures in place, OLEXPLORE creates a facsimile DLL, replicating both the file name and exported function names from the original. It then places this mock-DLL into a directory of higher priority to test whether it supersedes the legitimate DLL in the loading sequence.

By enabling PageHeap [50], a debugging tool designed to detect heap-related memory errors, the system can identify more detailed memory issues, including Type-1 and Type-3 vulnerabilities. In the event of a crash, memory context information and function call stack details are automatically dumped and logged within the system for subsequent analysis. Leveraging the WinDbg [51] debugger on the Windows platform, we delved into the analysis of dump files produced by crashes. We employed WinDbg to scrutinize the crash dump files, to further study the root cause of vulnerabilities and identify points of exploitation. To validate the effectiveness of these analyses, we conducted manual tests using previously generated RTF documents, verifying potential vulnerabilities in practice. Such a verification process aids in constructing PoC for the vulnerabilities, thereby enhancing corresponding security measures.

F. Vulnerability Analysis

To assess the severity of these identified vulnerabilities, OLEXPLORE includes the process of weaponization. This enables a comprehensive understanding of the potential implications of each vulnerability and informs the development of mitigation strategies. In OLEXPLORE, the weaponization techniques for these vulnerabilities are innovative, featuring concealment methods that have not been disclosed, despite the fact that we have responsibly disclosed these vulnerabilities and the methods of weaponizing them to Microsoft.

Post-construction, it is confirmed that 18 vulnerabilities detected by OLEXPLORE are capable of facilitating remote code execution for malevolent purposes while the remaining ones can lead to at least DoS attacks. After being meticulously crafted, these vulnerabilities could be exploited with just one click, allowing malicious code or DLLs to execute imperceptibly, posing a significant threat. The complete exploitation process comprises three sequential stages: 1) disguising the RTF document as an alternate file format, 2) delivering a malicious DLL in a stealthy manner, and 3) bypassing the Protected View Mode [31] to allow OLE content to be loaded by Office applications.

We use the Word application (i.e., `Winword.exe`) with Protected View Mode to conduct our tests. Protected View Mode feature is the well-known yellow bar with an "Enable Editing" button in MS Office. In Protected View mode, features that may pose security or privacy risks are disabled. This includes ActiveX controls, Object Linking and Embedding, macros, and loading of remote resources. Protected View Mode only is enabled when Word, Excel or PowerPoint opens documents with the Mark-of-the-Web (MOTW) [52]. We provide a detailed description of the process of weaponizing vulnerabilities in Appendix B.

V. EVALUATION

In this section, we evaluate OLEXPLORE by analyzing all registered COM and OLE components within Windows 10, Windows Server 2019, Windows Server 2022, Windows Server 2023, and Windows 11 environments. The operating systems were configured to their default settings, and we installed several commonly used applications (e.g., Visual Studio, Microsoft Exchange). The evaluations were performed on a desktop system equipped with an Intel i9-13900H processor and 32GB of memory. Our assessment addressed the following research questions:

- RQ1: How effective are the most important components of OLEXPLORE (i.e., OLE identification and storage fuzzing)?
- RQ2: How effective is OLEXPLORE on detecting vulnerabilities within OLE components specific to Office?
- RQ3: How precise is OLEXPLORE in detecting unsafe OLE components?

A. RQ1: Evaluation of OLExplore's Components

1) Identifying OLE components from COM components: OLEXPLORE, operating under the Windows 10 environment, enumerated and analyzed a total of 7,361 COM components. We focus on COM components developed by Microsoft first as non-Microsoft ones are less likely to be exploited by attackers due to the lack of universality and stability. Microsoft COM components derived from the Windows Software Development Kit (SDK) were also incorporated into the screening scope by OLEXPLORE, making them suitable for analyzing the machines with development environments.

On Windows Server 2022 version 10.0.20348.1487, OLEXPLORE extracted 7,274 CLSIDs, whereas 7,369 were identified on Windows 11, indicating small variations in the total count of COM components across different versions of the Windows operating systems. Among the numerous COM components,

TABLE I: List of vulnerabilities discovered by OLExplore

Module	Vuln. Type	Impact	Confirmed Version	Status
Windows Runtime	Type-1	Remote Code Execution	Windows 10 in 2021	CVE-2022-21878
	Type-1	Remote Code Execution	Windows 10 in 2021	CVE-2022-21888
	Type-1	Remote Code Execution	Windows 10 in 2021	CVE-2022-21971
	Type-1	Denial of Service	Windows 10 & Windows Server 2022	Confirmed
	Type-1	Denial of Service	Windows 10 & Windows Server 2022	Confirmed
	Type-1	Remote Code Execution	Windows 10 in 2021	CVE-2022-21992
	Type-1	Remote Code Execution	Windows 10 in 2021	CVE-2022-21974
	Type-1	Remote Code Execution	Windows 11 & Windows Server in 2023	CVE-2023-29366
	Type-1	Remote Code Execution	Windows 11 & Windows Server in 2023	CVE-2023-29367
	Type-1	Remote Code Execution	Windows 11 & Windows Server in 2023	CVE-2023-35313
	Type-1	Remote Code Execution	Windows 11 & Windows Server in 2023	CVE-2023-35323
	Type-1	Remote Code Execution	Windows 11 & Windows Server in 2023	CVE-2023-36704
Visual Studio	Type-1	Remote Code Execution	Visual Studio in 2023	CVE-2023-28296
Windows Geolocation Service	Type-2	Remote Code Execution	Windows Server 2019 & 2022	CVE-2023-35343
Tablet Windows UI App. Core	Type-2	Remote Code Execution	Windows 11 21H2 & 22H2	CVE-2023-36898
Windows UI App. Core	Type-2	Remote Code Execution	Windows Server 2019 & 2022	CVE-2023-36393
Windows Runtime	Type-2	Remote Code Execution	Windows 11 23H2 & 22H2	CVE-2024-21435
Microsoft Exchange Server	Type-2	Remote Code Execution	Microsoft Exchange Server 2019	CVE-2024-26198
Windows Runtime	Type-2	Remote Code Execution	Windows Server 2019 & 2022	Confirmed
Windows Inking COM	Type-3	Remote Code Execution	Almost all versions of Windows	CVE-2022-23290
Windows Runtime	Type-3	Denial of Service	Windows 10 & Windows Server 2022	Confirmed
	Type-3	Denial of Service	Windows 10 & Windows Server 2022	Confirmed
	Type-3	Denial of Service	Windows 10 & Windows Server 2022	Confirmed
	Type-3	Denial of Service	Windows 10 & Windows Server 2022	Confirmed
	Type-3	Denial of Service	Windows 10 & Windows Server 2022	Confirmed
	Type-3	Denial of Service	Windows Server 2022	Confirmed

OLEXPLORE has identified 257 existing OLE components in the system by enumerating and filtering interfaces. In comparison to Windows 10, the quantity of OLE components existing in Windows 11 did not exhibit any significant change

For comparison purposes, we conducted tests using Dranzer [53], an outdated tool but still the best match we can find, on systems with similar configurations. Dranzer found only 5,839 COM components and failed to recognize any OLE components within its purview. The scope of Dranzer's testing was largely confined to parameter testing multiple COM components loaded within Internet Explorer, aimed at detecting potential crash scenarios. However, given that Internet Explorer is no longer supported in versions of Windows beyond 10 and 11, the relevance of these tests has significantly diminished. And yet, if we were to adapt Dranzer for Office, Dranzer is still unable to detect these vulnerabilities. This limitation arises because Dranzer's tests focus exclusively on the loading of ActiveX controls. Although Office does include ActiveX capabilities, due to Microsoft's security policies, ActiveX parsing is disabled by default. Consequently, when it reaches the `CoCreateInstance` stage, it will not proceed with further parsing of ActiveX components.

2) Effectiveness of coverage and feedback in micro-snapshot fuzzing: The feedback component plays a crucial role in conjunction with snapshot fuzzing. Without feedback and instrumentation, bypassing progressive checks via random mutation (as in WinAFL) is almost impossible. For example, `MTEFData` objects in `EQNEDT32.exe` are read 21 times during the loading process and have 7 checksum fields. Due to the presence of the feedback component, whenever a check is passed, new code coverage is revealed and hence, the current chunk and all previous chunks are saved as a snapshot. This snapshot bootstraps the input mutation with a focus on either passing the next check or testing other code logic after these checks. This example shows that compared with blackbox fuzzing (i.e., no feedback on the usefulness of input chunk), coverage-guided micro-snapshot fuzzing can significantly increase the chances of discovering vulnerabilities that require passing multiple validation checks.

3) Effectiveness of micro-snapshot fuzzing in general: We compare Type-3 bug detection with and without micro-snapshot fuzzing. Among the 7 identified Type-3 bugs, 4 were found with micro-snapshot fuzzing in non-ActiveX items, whereas none of these bugs could be found without micro-snapshot fuzzing (i.e., by randomly mutating the entire "structured storage"). As showcased in Section V-A2, the combined use of snapshotting and coverage-guided feedback allows for a more effective fuzzing process. Snapshotting saves the execution context, making it easier to resume operations, while feedback provides real-time information that guides the mutation process, enhancing the likelihood of passing sanity checks and uncovering complex bugs.

4) Performance of micro-snapshot fuzzing: Over a 2500-minute testing period on `inkobj.dll`, the total number of basic blocks increased from 1239 with whole-storage random mutation (baseline) to 1639 with whole-storage mutation with

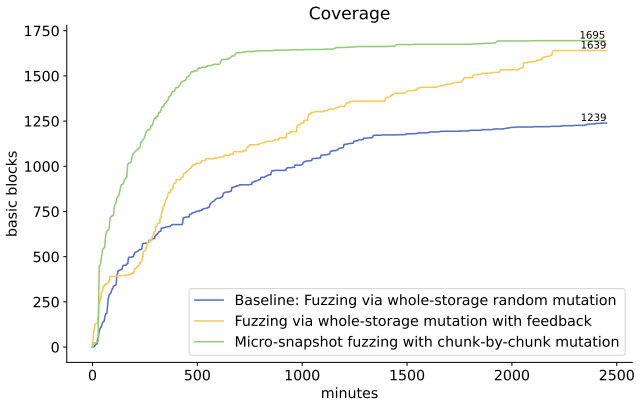


Fig. 9: Performance of micro-snapshot fuzzing

feedback, and finally to 1695 with micro-snapshot fuzzing with chunk-by-chunk mutation, as shown in Figure 9.

Compared to using only feedback-based methods, the inclusion of micro-snapshot fuzzing does not show a significant improvement in the total number of basic blocks covered. However, since snapshotting saves the execution context, there is a noticeable increase in coverage during the early testing phases. Higher coverage is crucial because it implies a more thorough examination of the code, potentially leading to the discovery of edge cases and hidden bugs that might not be detected with lower coverage. This demonstrates the superior effectiveness of micro-snapshot fuzzing in exploring execution paths and uncovering potential vulnerabilities.

We measured an average of 107 and 112 executions per second on `inkobj.dll` and `inked.dll`, respectively, where one execution spans from loading a snapshot to the end of `IPersistStorage::Load`. Without snapshotting, the rates were 375 and 472 `IPersistStorage::Load` executions per second, respectively.

Although the execution rate is lower when using micro-snapshot fuzzing, the trade-off is justified by the increased coverage. The snapshotting approach allows for deeper exploration of the code, focusing on previously unexplored paths and increasing the likelihood of finding critical vulnerabilities that might be missed by traditional fuzzing methods. Thus, while micro-snapshot fuzzing incurs a performance overhead, it substantially enhances code coverage and the effectiveness of vulnerability discovery.

B. RQ2: Vulnerability Finding Effectiveness

Table I presents the detailed information of confirmed vulnerabilities, with 17 of the vulnerabilities being assigned CVE IDs. For each verified CVE, we provide the module where the vulnerability resides, the type of vulnerability, its impact, the affected build versions, and the assigned CVE ID.

18 vulnerabilities listed are capable of being exploited for remote code execution. We will offer a root-cause analysis for each type of vulnerability in the following subsections.

Type-1 vulnerability example: CVE-2022-21971. Within the constructor `CreateInstance`, an instance of a `WapAuthProvider` object is allocated and initialized, occupying 0x78 bytes in memory. However, the object's state is not fully initialized. Upon invocation of the destructor `WapAuthProvider:: WapAuthProvider`, the pointer at offset 0x50 is released without being previously initialized. Figure 10 displays the vulnerable code snippet exploitable for remote arbitrary code execution.

```
1  void
   ↪  WapAuthProvider::~WapAuthProvider(__int64
   ↪  this) {
2    void *v2; // rcx
3    void *v3; // rcx
4
5    *(_QWORD *)this =
     ↪  &WapAuthProvider::`vftable';
6    LocalFree(*(HLOCAL *)(this + 56));
7    v2 = *(void **)(this + 64);
8    *(_QWORD *)(this + 56) = 0i64;
9    LocalFree(v2);
10   v3 = *(void **)(this + 80);        // <--
     ↪  [0] uninitialized
11   *(_QWORD *)(this + 64) = 0i64;
12   LocalFree(v3);                     // <--
     ↪  [1] free
13   *(_QWORD *)(this + 80) = 0i64;
14 }
```

Fig. 10: Simplified code snippet of CVE-2022-21971 Windows Runtime Remote Code Execution Vulnerability

Type-2 vulnerability example: CVE-2023-35343. Throughout the course of our testing, we have uncovered two significant security vulnerabilities. For instance, CVE-2023-35343 serves as a prototypical case; it constitutes a remote code execution vulnerability pertaining to the Windows Location Service, adversely affecting the standard configurations of Windows Server 2019 and 2022.

Our root cause analysis found that the vulnerability occurs when the `CoCreateInstance` function calls the `GetFindMyDeviceEnabled` method, leading to an attempt to load the library `mdmcommon.dll` with `LoadLibraryW`.

However, the `mdmcommon.dll` file does not exist within the Windows Server environment. An attacker can exploit this flaw by placing a malicious library file `mdmcommon.dll` in the current directory, potentially triggering a remote code execution attack. A logical error has been uncovered, and the discovered vulnerability could be readily exploited under the condition that the attacker manages to place both a malicious document and a corresponding malicious DLL in the victim's current working directory to carry out the attack(e.g., by placing a document and a DLL in the same zip file).

Type-3 vulnerability example: CVE-2022-23290. While investigating the `IPersistStorage` interface, we invoked the method in the interface named `CSketchInk::IPersistStreamInit_Load` and performed a memory allocation operation. However, this process only achieved partial initialization. Upon analyzing the function `InkObj!CSketchInk::FreeStrokeList`, we

```
1  HeapAlloc(*(HANDLE *)Default, *((_DWORD
   ↪  *)Default + 2), 0x70);
2  ...
3  v6 = *(void **)(this+0x10); //
   ↪  Uninitialized pointer
4  HeapFree(*(HANDLE *)Default, *((_DWORD
   ↪  *)Default + 2), v6);
5
6  mov rdi, qword ptr [rax+10h]
7  ds:00000158`42fcbfa0=c0c0c0c0c0c0c0c0
```

Fig. 11: Simplified code snippet of CVE-2022-23290 Windows Inking COM Remote Code Execution Vulnerability

identified an issue with an uninitialized pointer. To observe this behavior more closely, we enabled full PageHeap and conducted a trace in WinDbg. Figure 11 illustrates the vulnerability segment after simplifying the logic at `InkObj!CSketchInk::FreeStrokeList` +0x3d and the corresponding assembly code during single-step execution. It is evident that accessing the allocated memory results in abnormal values `(c0c0c0c0c0c0c0c0)`, which likely indicates a memory corruption issue.

TABLE II: Bug statistics by OLEXPLORE and others

Tool	Crashes	Bugs	CVEs
WinAFL	1	1	0
WINNIE	0	0	0
OLExplore	40	26	17

During the 24-hour testing period, OLExplore triggered 40 crashes, of which 26 were confirmed as bugs. In contrast, WinAFL only triggered a single crash, while WINNIE did not trigger any crashes at all. These vulnerabilities are challenging to detect through conventional data mutation fuzzing because constructing the data without knowing to maintain equal lengths in certain parts is problematic. Furthermore, byte misalignment often prevents the data from passing verification checks. This difficulty in crafting suitable test cases is why WinAFL and WINNIE were unable to achieve similar results. Table II shows the comparison results of three tools.

Responsible Disclosure. Upon identifying vulnerabilities, we promptly reported them to the Microsoft Security Response Center (MSRC) with PoC exploits. In October 2021, we submitted the first batch of vulnerabilities affecting Windows 10. These issues were progressively addressed, with patches released by early 2022. In February 2023, we identified and reported a second batch of vulnerabilities affecting Windows 11 and Windows Server, and these were patched throughout the latter half of 2023. Later in 2023, we conducted a comprehensive screening across all scenarios, identifying additional vulnerabilities, which were reported and subsequently patched by Microsoft, with the final patch released in March 2024. Throughout this process, we maintained continuous communication with MSRC to ensure timely resolution and patch release for all reported issues.

C. RQ3: False Positives in OLEXPLORE

We conducted an in-depth investigation into the exposed bugs and the details of vulnerability crashes. Our analysis indicates that the majority of false positives are attributable to null pointer exceptions on Windows systems.

TABLE III: Null pointer and vulnerability counts

Version	# Crash	# NULL pointer	# CVE
Windows 10	12	7	5
Windows Server 2022	13	8	5

As demonstrated in Table III, on Windows 10 version 10.0.19041.1237, Type-1 and Type-3 vulnerabilities yielded a total of 12 crash dump files. Out of these, 5 were confirmed as CVEs, while the remaining 7 were identified as null pointer issues upon further examination. Similarly, on Windows Server 2022 version 10.0.20348.1487, the count of null pointer problems stands at 8. The rationale for classifying these bugs as false positives rather than CVEs is rooted in their exploitability, or lack thereof, under modern Windows and Office mitigation mechanisms; such issues are substantially less likely to be leveraged for security attacks.

VI. DISCUSSIONS

A. Generality of OLEXPLORE and Future Work

OLE is a typical superware architecture where third-party modules are dynamically loaded into the same memory space shared by the main application and other modules. External inputs are processed by external modules solely with the main application performing minimal checks. This paradigm applies beyond OLE to other popular software packages as well, such as WeChat and Alipay.

Porting OLExplore to other superware. Migrating the collection and analysis of OLE components to certain Office-like superware (e.g., WPS Office) does not require excessive manual work since most mechanisms can be directly ported, as these types of software directly support OLE. However, to migrate micro-snapshot fuzzing to other software, it is necessary to identify input-reading functions like `CExposedStream::Read` (e.g., the `DecodeWxam` function in WeChat that processes proprietary formats like WXAM [54]). This still requires some manual work in reverse engineering to confirm that inputs are processed chunk-by-chunk and to choose the appropriate instrumentation mechanism, all of which require manual effort. Once these aspects are verified, the fuzzing architecture can be migrated. To find Type-1/2 vulnerabilities, manual effort is needed to identify the dynamic module loading mechanism, especially key functions like `CoCreateInstance`. The bug oracle of OLExplore can be reused.

Reusing Attack Surface Among Large and Complex Software Systems. It is well known that vulnerabilities in dynamically loadable modules offer significant exploit opportunities. In fact, attack vectors discussed in Section III are also applicable to other software systems, some of which have been well studied. For example: Type-1 vulnerabilities (dependency

confusion) are common issues in supply chain security [55]; Type-2 vulnerabilities (DLL-preloading) have been highlighted in recent studies [56] particularly in applications such as Chrome and Adobe Reader; and Type-3 vulnerabilities, which often involve parsing and handling of embedded objects, are prevalent in Adobe Reader, as evidenced by CVE-2019-8014 [57] and CVE-2021-44711 [58].

Extending OLExplore to Vulnerabilities in Linked Objects. A significant security concern titled "Moniker Magic: Running Scripts Directly in Microsoft Office" [59] highlighted vulnerabilities such as CVE-2017-0199, which exploit the URL Moniker feature to load remote `.hta` files. These vulnerabilities arise from flawed design choices in the Office file loading mechanisms and require IE10 or IE11 to be present. The exploit can occur without user interaction, although a dialog box appears during the process. FireEye later disclosed details of a captured sample exploiting CVE-2017-0199 [60]. Despite patches that introduced blacklist mechanisms for unsafe Monikers, subsequent vulnerabilities indicate that many problematic Monikers remain unidentified.

To address these linked object vulnerabilities, OLExplore can be extended to analyze and detect such issues. OLExplore includes a component for collecting OLE components. By adding features such as scanning for `StdOleLink` identifiers, it can quickly identify and gather all linked objects on the system. In future work, it is necessary to investigate the vulnerabilities of the identified linked objects, categorize them based on their root causes, and incorporate the corresponding bug oracles into the OLExplore components. By enhancing OLExplore's capabilities to scrutinize both embedded and linked objects, it can provide a more comprehensive security analysis of Office documents.

VII. Related Work

COM Security Enhancements. Dewey et al. [61] found that COM has many vulnerabilities that attackers can exploit to compromise systems. Attackers can bypass security policies of popular applications, which allow the use of many flawed controls. To solve this, they created COMBLOCK, a reference monitor that ensures all COM operations follow a global policy.

Detection of Vulnerabilities in COM Objects. Gu et al. [62] introduced COMRACE, a tool that detects data race vulnerabilities in COM objects. It uses static binary analysis to identify insecure methods in COM binaries and validates these findings with synthesized PoCs, effectively identifying and mitigating data race vulnerabilities.

Type Confusion Vulnerabilities in COM. Zhang et al. [63] presented COMFUSION, the first tool to find union type confusion vulnerabilities in Windows COM. Before this, no tools could recover union types in binaries. COMFUSION bridges this gap, improving COM application security by detecting and fixing these vulnerabilities.

Fuzzing Techniques for Windows. WINNIE [17] improves Windows fuzzing by directly invoking target functions and using an efficient fork implementation, avoiding unnecessary GUI code. WinFuzz [19] introduces target-embedded snapshotting, allowing applications to snapshot themselves without source code or kernel modifications, enhancing precision in fuzzing.

VIII. Conclusion

In this paper, we introduce OLExplore, a new tool crafted for systematic detection of vulnerabilities within OLE components for Office applications. OLExplore is based on an exhaustive analysis of historical vulnerabilities associated with OLE components, from which we summarized three distinct vulnerability patterns. OLExplore conducts static analysis on all COM entities available on the platform to cull a subset of OLE components, which are then subjected to dynamic analysis leveraging the trio of vulnerability traits for comprehensive security evaluation. In our empirical studies, OLExplore scrutinized the attack surface of 257 OLE objects, uncovering 26 vulnerabilities, out of which 17 have been assigned CVE numbers with remote code execution potential.

Acknowledgment

This work was partly supported by the National Key R&D Program of China (2021YFB2701000), the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), the National NSF of China (grants No.62302181, 62072046), the Knowledge Innovation Program of Wuhan-Basic Research, Huawei Research Fund, and HUSTCSE-FiberHome Joint Research Center for Network Security.

References

[1] NCC Group, "Understanding Microsoft Word OLE Exploit Primitives: Exploiting CVE-2015-1642 Microsoft Office CTaskSymbol Use-After-Free Vulnerability," https://research.nccgroup.com/wp-content/uploads/2020/12/Understanding-Microsoft-Word-OLE-Exploit-Primitives-Exploiting-CVE-2015-1642.pdf, 2015.

[2] McAfee, "An Inside Look into Microsoft Rich Text Format and OLE Exploits," https://www.mcafee.com/blogs/other-blogs/mcafee-labs/an-inside-look-into-microsoft-rich-text-format-and-ole-exploits/, 2020.

[3] Q. Li and Q. Jin, "Needle in A Haystack: Catch Multiple Zero-days Using Sandbox," https://images.seebug.org/archive/Catch_Multiple_Zero-Days_Using_Sandbox-EN.pdf, 2019.

[4] S. Salehi, I. Miremadi, M. Ghasempour Nejati, and H. Ghafouri, "Fostering the Adoption and Use of Super App Technology," *IEEE Transactions on Engineering Management*, 2023.

[5] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them," in *2016 IEEE Cybersecurity Development (SecDev)*, 2016, pp. 45–52.

[6] M. Zalewski, "Afl," https://lcamtuf.coredump.cx/afl, 2017.

[7] Google, "ClusterFuzz," https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md, 2018.

[8] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[9] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, and R. Beyah, "V-Fuzz: Vulnerability Prediction-Assisted Evolutionary Fuzzing for Binary Programs," *IEEE Transactions on Cybernetics*, vol. 52, no. 5, pp. 3745–3756, 2022.

[10] Google, "Honggfuzz," https://github.com/google/honggfuzz, 2010.

[11] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft." *Queue*, vol. 10, no. 1, p. 20–27, jan 2012. [Online]. Available: https://doi.org/10.1145/2090147.2094081

[12] The KLEE Team, "Klee," https://klee.github.io, 2015.

[13] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[14] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[15] K. Weiss and J. Schütte, "Annotary: A Concolic Execution System for Developing Secure Smart Contracts," in *Computer Security – ESORICS 2019*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds. Cham: Springer International Publishing, 2019, pp. 747–766.

[16] X. Song, Z. Wu, and Y. Wang, "Directer: A Parallel and Directed Fuzzing based on Concolic Execution," in *Proceedings of the 7th International Conference on Software and Information Engineering*, ser. ICSIE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 87–92. [Online]. Available: https://doi.org/10.1145/3220267.3220272

[17] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "Winnie: Fuzzing windows applications with harness synthesis and fast cloning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[19] L. Stone, R. Ranjan, S. Nagy, and M. Hicks, "No linux, no problem: Fast and correct windows binary fuzzing via target-embedded snapshotting," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4913–4929. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/stone

[20] Google Project Zero, "Jackalope," https://github.com/googleprojectzero/Jackalope, 2020.

[21] J. Pan, G. Yan, and X. Fan, "Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 149–165. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/pan

[22] J. E. Forrester and B. P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, ser. WSS'00. USA: USENIX Association, 2000, p. 6.

[23] Y. Choi, H. Kim, and D. Lee, "An Empirical Study for Security of Windows DLL Files Using Automated API Fuzz Testing," in *2008 10th International Conference on Advanced Communication Technology*, vol. 2, 2008, pp. 1473–1475.

[24] Google Project Zero, "WinAFL," https://github.com/googleprojectzero/winafl, 2016.

[25] J. Choi, K. Kim, D. Lee, and S. K. Cha, "NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 677–693.

[26] M. Heuse, "AFL-DynamoRIO," https://github.com/vanhauser-thc/afl-dynamorio, 2018.

[27] Q. Jin, "How I Found 16 Microsoft Office Excel Vulnerabilities in 6 Months," https://conference.hitb.org/hitbsecconf2021ams/materials/D2T1%20-%20How%20I%20Found%2016%20Microsoft%20Office%20Excel%20Vulnerabilities%20in%206%20Months%20-%20Quan%20Jin.pdf, 2021.

[28] M. Debasish, "OpenXMolar," https://github.com/debasishm89/OpenXMolar, 2017.

[29] Check Point, "FUZZING THE OFFICE ECOSYSTEM," https://research.checkpoint.com/2021/fuzzing-the-office-ecosystem/, 2021.

[30] LLVM Project, "LibFuzzer," https://llvm.org/docs/LibFuzzer.html, 2003.

[31] Y. C. Koh, "Understanding the Microsoft Office 2013 Protected-View Sandbox," https://labs.withsecure.com/content/dam/labs/docs/UNDERSTANDING-THE-MICROSOFT-OFFICE-2013-PROTECTED-VIEW-SANDBOX-WP3.pdf, 2015.

[32] Microsoft, "Insert an object in your Excel spreadsheet - Microsoft Support," https://support.microsoft.com/en-us/office/insert-an-object-in-your-excel-spreadsheet-e73867b2-2988-4116-8d85-f5769ea435ba, 2023.

[33] ——, "OffVis," https://download.microsoft.com/download/1/2/7/127ba59a-4fe1-4acd-ba47-513ceef85a85/offvis.zip, 2011.

[34] ——, "Privileges," https://learn.microsoft.com/en-us/windows/win32/secauthz/privileges, 2021.

[35] "CVE-2017-11882," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11882, 2017.

[36] FireEye, "New Targeted Attack in the Middle East by APT34, a Suspected Iranian Threat Group, Using CVE-2017-11882 Exploit," https://www.mandiant.com/resources/blog/targeted-attack-in-middle-east-by-apt34, 2017.

[37] Microsoft, "Process Monitor," https://learn.microsoft.com/en-us/sysinternals/downloads/procmon, 2023.

[38] ——, "Get-Member," https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-member?view=powershell-7.4, 2023.

[39] J. Forshaw, "OleViewDotnet," https://github.com/tyranid/oleviewdotnet, 2014.

[40] S. Vittitoe, "Issue 514: Microsoft Office / COM Object DLL Planting with els.dll," https://bugs.chromium.org/p/project-zero/issues/detail?id=514, 2015.

[41] W. Dormann, "Attacking COM via Word RTF," https://insights.sei.cmu.edu/library/attacking-com-via-word-rtf/, 2021.

[42] Microsoft, "ActiveX Controls Architecture," https://learn.microsoft.com/en-us/windows/win32/com/activex-controls-architecture, 2019.

[43] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-Aware Greybox Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 724–735.

[44] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13. [Online]. Available: https://doi.org/10.1145/3395363.3397372

[45] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, "GRIMOIRE: Synthesizing Structure while Fuzzing," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1985–2002. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

[46] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2597–2614. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[47] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[48] Microsoft, "Detours," https://github.com/microsoft/Detours, 2018.

[49] Google Project Zero, "TinyInst," https://github.com/googleprojectzero/TinyInst.

[50] Microsoft, "GFlags and PageHeap - Windows drivers," https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap, 2022.

[51] ——, "WinDbg," http://www.windbg.org, 2023.

[52] Outflank, "Mark-of-the-Web from a Red Team's Perspective," https://www.outflank.nl/blog/2020/03/30/mark-of-the-web-from-a-red-teams-perspective/, 2017.

[53] D. Plakosh and W. Dormann, "Dranzer," https://github.com/CERTCC/dranzer, 2009.

[54] Signal Labs, "Fuzzing WeChat's Wxam Parser," https://signal-labs.com/fuzzing-wechats-wxam-parser/, 2022.

[55] S. Neupane, G. Holmes, E. Wyss, D. Davidson, and L. D. Carli, "Beyond Typosquatting: An In-depth Look at Package Confusion," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3439–3456. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/neupane

[56] C. Yu, Y. Xiao, J. Lu, Y. Li, Y. Li, L. Li, Y. Dong, J. Wang, J. Shi, D. Bo, and W. Huo, "File Hijacking Vulnerability: The Elephant in the Room," *Proceedings 2024 Network and Distributed System Security Symposium*,

2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:267621808
[57] "CVE-2019-8014," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8014, 2023.
[58] "CVE-2021-44711," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44711, 2023.
[59] H. Li and B. Sun, "Moniker Magic: Running Scripts Directly in Microsoft Office," https://leo00000.github.io/pdf/Moniker_Magic_final.pdf, 2017.
[60] FireEye, "CVE-2017-0199: In the Wild Attacks Leveraging HTA Handler," https://www.mandiant.com/resources/blog/cve-2017-0199-hta-handler, 2017.
[61] D. Dewey and P. Traynor, "No Loitering: Exploiting Lingering Vulnerabilities in Default COM Objects," in *NDSS*, 2011.
[62] F. Gu, Q. Guo, L. Li, Z. Peng, W. Lin, X. Yang, and X. Gong, "COMRace: Detecting Data Race Vulnerabilities in COM Objects," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3019–3036. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/gu-fangming
[63] Y. Zhang, X. Zhu, D. He, M. Xue, S. Ji, M. S. Haghighi, S. Wen, and Z. Peng, "Detecting Union Type Confusion in Component Object Model," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.
[64] houjingyi, "office-exploit-case-study," https://github.com/houjingyi233/office-exploit-case-study, 2018.
[65] Offsec, "Exploit Database," https://www.exploit-db.com/, 2024.
[66] Google Project Zero, "project-zero," https://bugs.chromium.org/p/project-zero/issues/list, 2024.
[67] "CVE-2023-21716," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-21716, 2023.
[68] Microsoft, "Description of the security update for Office 2016: April 11, 2017," https://support.microsoft.com/en-us/topic/description-of-the-security-update-for-office-2016-april-11-2017-c4cec448-05d9-1ae2-37b0-869ec9a0aa71, 2017.
[69] ——, "Data Execution Prevention," https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention, 2023.
[70] Pax, "ASLR," https://pax.grsecurity.net/docs/aslr.txt, 2023.
[71] Microsoft, "Blocking Flash, Shockwave, Silverlight controls from activating in Office Applications for Security," https://techcommunity.microsoft.com/t5/security-compliance-and-identity/blocking-flash-shockwave-silverlight-controls-from-activating-in/ba-p/191729, 2018.
[72] H. Li and B. Sun, "Attacking Interoperability: An OLE Edition," https://www.blackhat.com/docs/us-15/materials/us-15-Li-AttackingInteroperability-An-OLE-Edition.pdf, 2015.
[73] UINT 42, "In-Depth Analysis of July 2023 Exploit Chain Featuring CVE-2023-36884 and CVE-2023-36584," https://unit42.paloaltonetworks.com/new-cve-2023-36584-discovered-in-attack-chain-used-by-russian-apt/, 2023.
[74] A. Sotirov, "Heap Feng Shui in Javascript," https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Whitepaper/bh-eu-07-sotirov-WP.pdf, 2007.
[75] Parvez, "Spraying the heap in seconds using ActiveX controls in Microsoft Office," https://www.greyhathacker.net/?p=911, 2015.
[76] McAfee, "Microsoft Kills Potential Remote Code Execution Vulnerability in Office (CVE-2017-8630)," https://www.mcafee.com/blogs/other-blogs/mcafee-labs/microsoft-kills-potential-remote-code-execution-vulnerability-in-office-cve-2017-8630/, 2017.

APPENDIX

A. Summary of Known Office Vulnerabilities

Our work on OLE vulnerability detection is motivated by an exhaustive analysis of all historical vulnerabilities associated with Office products. The data comes from publicly disclosed research [64], [3] and databases [65], [66]. These vulnerabilities can roughly be categorized into 7 types based on their root cause and Figure 12 illustrates their respective proportions in past occurrences.

Notably, Embedded OLE Object Parsing Problems are predominant, accounting for 43.24% of the vulnerabilities, and they continue to pose a significant risk at present. Among the remaining vulnerabilities, 27.03% are no longer applicable because the vulnerable applications have reached the end of life. Additionally, 13.51% of the vulnerabilities are challenging to exploit and are unlikely to be used in practical scenarios, as such vulnerabilities have neither been disclosed nor captured being exploited in the wild over the past five years.

- *RTF Control Word and Open XML Tag Parsing Problems*. These two types are grouped together due to their association with specific markers or identifiers within their respective formats. Both require memory corruption that necessitates precise memory control like heap feng shui for exploitation. However, the absence of good primitives for memory manipulation typically makes exploitation more challenging. Despite contemporary mitigation techniques, such vulnerabilities persist. For example, CVE-2023-21716 [67] demonstrates a Microsoft Word remote code execution vulnerability. Yet, as evidenced by the lack of detected memory corruption exploits over the past five years, these vulnerabilities are now significantly more difficult to exploit than before.
- *EPS File Parsing Problems*. EPS, once supported by Office as an image format featuring PostScript program code which is amenable to memory manipulations, hence was more exploitable. This attack vector, however, is no longer present; since April 2017, Office has disabled the insertion of EPS files into documents [68].
- *Other Resource File Parsing Problems*. In addition to EPS, Office still supports the insertion of various resource files, including 3D models, among others. A substantial number of vulnerabilities remain active in this area. Historically, such issues always proved similar, with updates (e.g., June 1st, 2023) temporarily disabling the functionality to insert SketchUp graphics (.skp files) into Office. However, these types of vulnerabilities are very scattered and small in number and are not included in the statistics.
- *Moniker Problems*. Moniker is an intrinsic feature of the Office suite, designed to facilitate linkage to various lo-

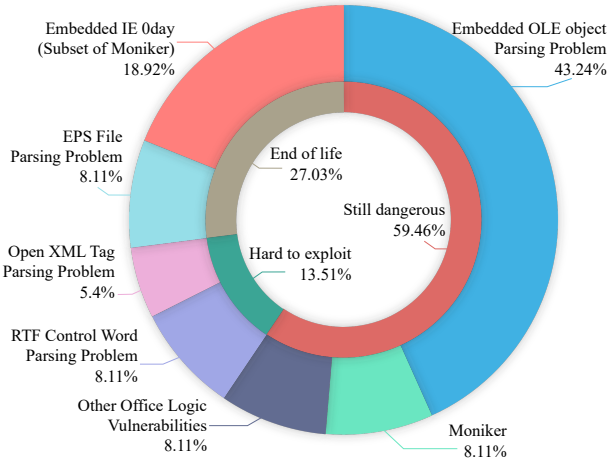


Fig. 12: CVEs Categorization

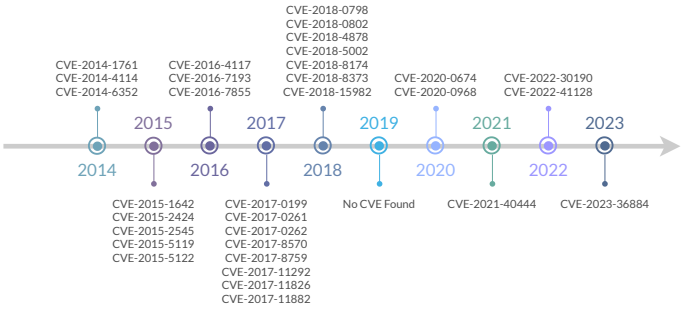


Fig. 13: Exploited vulnerabilities in the last 10 years.

cal or remote objects. Inherently, it does not constitute a vulnerability. Rather, the vulnerability emerges from the execution policy within Office applications pertaining to files fetched from remote links. For instance, opening a remotely loaded Excel file would typically pose no issue. However, when the files involved are of types such as HTA files and scripts, executing them directly can lead to security breaches due to improper handling by the execution policy [59]. Microsoft subsequently disabled some CLSIDs in their fix. Vulnerabilities such as CVE-2017-0199, CVE-2017-8570, and CVE-2017-8579 are related to this issue.

- *Office Embedded IE 0-day Vulnerabilities*. Derived from moniker vulnerabilities these utilize the `StdOleLink` function to load HTML content, which is then processed by the IE parsing module. Exploits like CVE-2018-8174/8373 and CVE-2020-0674/CVE-2021-40444 highlight attackers exploiting IE's VBScript and JavaScript 0-day vulnerabilities. As of August 2019, VBScript was disabled by default, and JavaScript followed in January 2023. The lifecycle of these vulnerabilities has also ended and is no longer a focal point of research.
- *Embedded OLE Object Parsing Problems*. CVE-2017-11882 serves as a classic example involving a vulnerability with the Equation Editor. `eqnedt32.exe` lacked mitigation measures (DEP [69] & ASLR [70]) and was removed in the public update of January 2018. Flash represented a subset of embedded OLE objects, categorized separately due to its significant impact. Numerous security vulnerabilities have been identified, leading Office Monthly Channel to block Flash execution starting in June 2018 [71]. This category includes not only memory corruption vulnerabilities but also logic flaws. Black Hat USA 2015 referenced these issues during "Attacking Interoperability: An OLE Edition" [72]. DLL preloading vulnerabilities could lead to loading DLL files from the current working directory. Our study will place emphasis on this class of vulnerabilities.

Figure 13 shows the exploited vulnerabilities in the last 10 years, from which we can observe a trend: the number of exploitable vulnerabilities in the wild has decreased. One reason is that Office vulnerabilities are becoming harder to find and exploit due to Microsoft's patches. Additionally, many of these vulnerabilities have reached the end of their lifecycle. Attackers urgently need to find a new potential attack surface which can be exploited for Office, and OLE type vulnerabilities cannot be easily overlooked due to their relatively high proportion. Therefore, it is essential to conduct security checks on OLE objects.

B. Weaponization Vulnerabilities

Disguising Document. It is essential to study the activation mechanisms for OLE within `Winword.exe` since attackers often use various document formats to launch attacks. In the context of OLE activation, a distinct difference in user interaction requirements can be observed.

- For documents with the *.doc* extension, a user must explicitly click the OLE object to activate it.
- Documents saved in the *.docx* format prompt a warning message upon an OLE object's attempted activation.
- However, RTF documents (*.rtf*) facilitate a more seamless user experience, where a single click not only opens the document but also loads involved OLE components without eliciting any warning notifications or necessitating a secondary click for OLE activation.

Given these characteristics, attackers exhibit a preference for utilizing RTF files as the medium of choice for their malicious endeavors. Consequently, for the purpose of conducting our experimental attack analysis, we have opted to select the RTF file format as the carrier for our test attack. However, due to the limited popularity of RTF, such attack activities can easily attract attention, leading attackers to disguise their format. Examples in the wild include: 1) changing *.rtf* to *.doc*, i.e., by modifying the file extension, or 2) converting *.rtf* into *.docx*: i.e., by embedding an RTF within a *.docx* file, as seen in the PoC for CVE-2023-36884 [73].

Heap Feng Shui. In the context of Type-1 and Type-3 vulnerabilities, heap feng shui [74] is crucial. Our objective is to ensure that the heap spraying is properly aligned with controlled data at a known address, such as the canonical `0x0a0a0a0a`, thereby circumventing mitigation strategies such as ASLR. Recent techniques leverage the use of ActiveX [75] and OLE objects [1] for heap spraying [76]. This can be achieved through the utilization of Microsoft Common Controls COM objects, including `Toolbar` objects or other OLE controls, exemplified by `TabStrip` objects (as demonstrated in malware samples like CVE-2013-3906) and bitmap images. Heap memory allocation is facilitated by embedding a specific number of these objects into documents.

Stealthy DLL Delivery. However, for Type-2 vulnerabilities, the exploitation methods are notably more diverse and innovative. We have chosen the following two methods for carrying malicious files to conceal their presence for imperceptible attacks:

- **In a local archive scenario**: An attacker might compress a malicious DLL file along with the document. When the victim decompresses it, the malicious document and DLL file are both loaded. With Windows' default settings, hidden files are not displayed in File Explorer, allowing attackers to exploit this feature to conceal the malicious DLL file from users' notice.

- **In remote shared directories**: When a user opens a malicious document in a remote shared directory, the malicious DLL file is also loaded from the shared directory. Attackers can modify protocols or control the shared directory to further disguise the malicious DLL file, making it undetectable to users. Since the shared directory is under the attacker's control, protocol modifications can also render the malicious DLL files invisible to users.

In the context of Type-2 DLL Preloading Attacks, successful exploitation requires the co-location of a malicious document and a corresponding malevolent DLL in the same directory. The `Winword.exe` process acknowledges two distinct types of Current Working Directory (CWD):

1) **Where the document is located**: This directory serves as the CWD for instances of `Winword.exe` that are initiated through `explorer.exe`. Positioning a malevolent DLL in this directory proves to be an effective strategy for exploitation.
2) **C:\Users\name\Documents**: An internal timer within Office applications alters the execution context after a time span of 10 seconds. Subsequently, `Winword.exe` modifies the current directory to `C:\Users\name\Documents`, which represents the default local file location according to Office settings. Post the directory transition, attempts at exploitation will no longer succeed.

Bypassing the Protected View Mode. Irrespective of the vulnerability type, whether it be Type-1 and Type-3 memory vulnerabilities or Type-2 logic vulnerabilities, a common challenge persists: to exploit these vulnerabilities, it is imperative for an OLE object to be loaded by Office. However, the Protected View Mode hinders the loading of OLE objects. Herein, we introduce an innovative technique to circumvent the Protected View Mode—exploiting the *AutoRecover* feature intrinsic to Office components. The built-in save functionality of Office components is designed to prevent user data loss; Office will still open any previously unintentionally closed documents alongside any new document that is opened after an abnormal termination. However, the Protected View Mode only offers protection during the first opening of a document. Thus, a program crash could be beneficial to us. The crash terminates the running process, and the next time a new process starts, it will not enter Protected Mode. The vulnerability we uncovered, when combined with a DoS flaw, can easily circumvent the Protected View Mode to enable the execution of malicious functions. In fact, there are numerous DoS vulnerabilities in Office (including our findings in Table I), but Microsoft states that DoS-type vulnerabilities do not meet the criteria for security update services and will not receive security updates.

C. Software Fault Isolation as Defense

Common mitigations include implementing stringent checks, such as allowlisting and sandbox isolation [31] in Office. However, such mitigations have not been strictly implemented for OLE. We can observe a more robust approach from Internet Explorer's handling of ActiveX controls.

Unlike with OLE, when the IE11 browser prepares to load an ActiveX control, it first verifies an "allowlist" within the Registry key under the path name `HKLM\Software\Microsoft\Windows\CurrentVersion\Ext\PreApproved`. The associated DLL is only loaded into the IE process if the ActiveX control's CLSID is present on this list. This mechanism indicates that the loading process for ActiveX controls in IE is selective, avoiding the loading of unvetted components. This highlights a clear disparity between Windows' method of addressing the security of OLE objects and IE's handling of ActiveX controls. The latter offers a more secure and filtered mechanism to prevent the execution of unaudited code. In summary, using third-party code without adequate verification constitutes an insecure design practice. Conducting security tests on these types of data is highly effective, and employing sandboxing techniques to completely isolate them until they have been thoroughly validated by the system or deemed trustworthy by users is advisable.