Transparent COM instrumentation for malware analysis

By David Zimmer <dzzie@yahoo.com>- Talos

- COM automation is a core Windows technology that allows code to access external functionality through well-defined interfaces. It is similar to traditionally loading a DLL, but is class-based rather than function-based. Many advanced Windows capabilities are exposed through COM, such as Windows Management Instrumentation (WMI).
- Scripting and late-bound COM calls operate through the IDispatch interface. This creates a key analysis point that many types of malware leverage when interacting with Windows components. This analysis point is quite complex and hard to safely instrumentate at scale.
- In this article, Cisco Talos presents DispatchLogger, a new open-source tool that closes this gap by delivering high visibility into late-bound IDispatch COM object interactions via transparent proxy interception.
- This blog describes the architecture, implementation challenges, and practical applications of comprehensive COM automation logging for malware analysis. This technique can be utilized on multiple types of malware.

Malware type	Binding type	Est. coverage
Windows Script Host	Always Late	100%
PowerShell COM	Always Late	100%
AutoIT	Always Late	100%
VBA Macros	Mostly Late	95%
VB6 Malware	Mixed	65%
.NET COM Interop	Mixed	60%
C++ Malware	Rarely Late (WMI)	10%

The challenge

Modern script-based malware (e.g., VBScript, JScript, PowerShell) relies heavily on COM automation to perform malicious operations. Traditional dynamic analysis tools capture low-level API calls but miss the semantic meaning of high-level COM interactions. Consider this attack pattern:

```
Set wmi = GetObject("winmgmts:\\.\root\cimv2")
Set processClass = wmi.Get("Win32_Process")
Set startup = wmi.Get("Win32_ProcessStartup").SpawnInstance_
startup.ShowWindow = 0  ' Hide window
processClass.Create "cmd.exe /c evil.exe", null, startup, processId
```

Figure 1. Sample VBScript code to create a process with WMI as its parent.

Behavioral monitoring will detect process creation, but the analyst often loses critical context such as who launched the process. In this scenario WMI spawns new processes with wmic.exe or wmiprvse.exe as the parent.

Technical approach

Interception strategy

DispatchLogger starts with API hooking at the COM instantiation boundary. Every COM object creation in Windows flows through a small set of API functions. By intercepting these functions and returning transparent proxies deep visibility can be achieved without modifying malware behavior.

The core API hooking targets are:

1. **CoCreateInstance**: Primary COM object instantiation (CreateObject in scripts)
2. **CoGetClassObject:** Class factory retrieval
3. **GetActiveObject:** Attachment to running COM instances
4. **CoGetObject/MkParseDisplayName:** Moniker-based binding (GetObject)
5. **CLSIDFromProgID:** ProgID resolution tracking

Why class factory hooking is essential

Initial implementation attempts hooked only CoCreateInstance, filtering for direct IDispatch requests. However, testing revealed that most VBScript CreateObject calls were not being intercepted.

To diagnose this a minimal ActiveX library was created with a MsgBox in Class_Initialize to freeze the process. The VBScript was launched, and a debugger attached to examine the call stack. The following code flow was revealed:

```
vbscript.dll!GetObjectFromProgID+0x123
CoGetClassObject(&clsid, ..., IID_IClassFactory, &ppFactory)
ppFactory->CreateInstance(NULL, IID_IUnknown, &ppunk)// ← IUnknown, not IDispatch!
ppunk->QueryInterface(IID_IDispatch, &pdisp)
```

Figure 2. Call stack showing how VBScript obtains a target IDispatch interface.

Disassembly of vbscript.dll!GetObjectFromProgID (see Figure 3) confirmed the pattern. VBScript's internal implementation requests IUnknown first, then queries for IDispatch afterward:

```
1   HRESULT __userpurge GetObjectFromProgID@<eax>(
2     ...
137   result = CoGetClassObject(&clsid, v14, v17, &IID_IClassFactory, &ppv);
138   if ( result < 0 )
139     return result;
140   if ( (**ppv)(**ppv, ppv, &IID_IClassFactory3, &v39) >= 0 )
141   {
142   ...
154   }
155   else
156   {
157     v19 = (*(*ppv + 12))(*(*ppv + 12), ppv, 0, &IID_IUnknown, &ppunk);// // <--- returne
158     (*(*ppv + 8))(*(*ppv + 8), ppv);
```

Figure 3. Disassembly of vbscript.dll!GetObjectFromProgID.

The key line is `CreateInstance(NULL, IID_IUnknown, &ppunk)`. Here, VBScript explicitly requests IUnknown, not IDispatch. This occurs because VBScript needs to perform additional safety checks and interface validation before accessing the IDispatch interface.

If we only wrap objects when IDispatch is directly requested in CoCreateInstance, we miss the majority of script instantiations. The solution is to also hook CoGetClassObject and wrap the returned IClassFactory:

```
HRESULT Hook_CoGetClassObject(..., IID_IClassFactory, &ppv) {
    HRESULT hr = Original CoGetClassObject(...);
    if (SUCCEEDED(hr)) {
        IClassFactory* pFactory = (IclassFactory*)*ppv;
        *ppv = new ClassFactoryProxy(pFactory);  // Wrap the factory
    }
    return hr;
}
```

Figure 4. Returning a Class Factory proxy from the CoGetClassObject API Hook.

The ClassFactoryProxy intercepts CreateInstance calls and handles both cases:

Figure 5.

```
HRESULT ClassFactoryProxy::CreateInstance(...) {
    HRESULT hr = m_pOriginal->CreateInstance(pUnkOuter, riid, ppv);
    if (SUCCEEDED(hr)) {
        // Case 1: VBScript pattern - IUnknown requested
        if (riid == IID_IUnknown) {
            IDispatch* pDisp;
            if (SUCCEEDED(pUnk->QueryInterface(IID_IDispatch, &pDisp))) {
                *ppv = WrapDispatch(pDisp);  // Replace with proxy
            }
        }
        // Case 2: Direct IDispatch request
        else if (riid == IID_IDispatch) {
            *ppv = WrapDispatch((IDispatch*)*ppv);
        }
    }
    return hr;
}
```

Returning an IDispatch Proxy from ClassFactoryProxy::CreateInstance if possible.

This ensures coverage regardless of which interface the script engine initially requests.

Architecture

Proxy implementation

The DispatchProxy class implements IDispatch by forwarding all calls to the wrapped object while logging parameters, return values, and method names. If the function call returns another object, we test for IDispatch and automatically wrap it.

```
class DispatchProxy : public IDispatch {
private:
    IDispatch* m_pOriginal;
    char m_objectName[256];
    DWORD m_proxyId;

public:
    STDMETHOD(Invoke)(DISPID dispIdMember, REFIID riid, LCID lcid,
        WORD wFlags, DISPPARAMS* pDispParams,
        VARIANT* pVarResult, EXCEPINFO* pExcepInfo,
        UINT* puArgErr) {

        // Log method call with parameters
        SendDebug(dispIdMember, pDispParams);

        // Forward to real object
        HRESULT hr = m_pOriginal->Invoke(dispIdMember, ...);

        // Log result
        SendDebug(dispIdMember, pVarResult, hr);

        // Recursively wrap returned IDispatch objects
        if (SUCCEEDED(hr) && pVarResult) {
            if (pVarResult->vt == VT_DISPATCH && pVarResult->pdispVal) {
                pVarResult->pdispVal = WrapDispatch(pVarResult->pdispVal);
            }
        }

        return hr;
    }
};
```

Figure 6. Simplified flow of IDispatch::Invoke hook. Full hook is around 300 loc.

The proxy is transparent, meaning it implements the same interface, maintains proper reference counting, and handles QueryInterface correctly. Malware cannot detect the proxy through standard COM mechanisms.

Recursive object wrapping

The key capability is automatic recursive wrapping. Every IDispatch object returned from a method call is automatically wrapped before being returned to the malware. This creates a fully instrumented object graph.

```
Set wmi = GetObject("winmgmts:")          ' Wrapped via CoGetObject hook
Set query = wmi.ExecQuery("...")          ' ExecQuery returns object → wrapped
For Each item In query                    ' Items enumerated → wrapped
      item.Terminate()                    ' Method calls logged
Next                                      ' We can also wrap Byref out parameters
```

Figure 7. Sample VBScript code detailing hooking capabilities.

Object relationships are tracked:

1. `GetObject("winmgmts:")` triggers hook, returns wrapped WMI service object
2. Calling `.ExecQuery()` goes through proxy, logs call with SQL parameter
3. Returned query result object is wrapped automatically
4. Enumerating with For Each retrieves wrapped IEnumVARIANT
5. Each enumerated item is wrapped as it's fetched
6. Calling `.Terminate()` on items logs through their respective proxies

Enumerator interception

VBScript/JScript For Each constructs use IEnumVARIANT for iteration. We proxy this interface to wrap objects as they're enumerated:

```
class EnumVARIANTProxy : public IEnumVARIANT {
    STDMETHOD(Next)(ULONG celt, VARIANT* rgVar, ULONG* pCeltFetched) {
        HRESULT hr = m_pOriginal->Next(celt, rgVar, pCeltFetched);

        if (SUCCEEDED(hr)) {
            for (ULONG i = 0; i < *pCeltFetched; i++) {
                if (rgVar[i].vt == VT_DISPATCH) {
                    rgVar[i].pdispVal = WrapDispatch(rgVar[i].pdispVal);
                }
            }
        }
        return hr;
    }
};
```

Figure 8. Implementation of IEnumVariant.Next that wraps child objects in the IDispatch proxy.

Moniker support

VBScript's `GetObject()` function uses monikers for binding to objects. We hook CoGetObject and MkParseDisplayName, then wrap returned moniker objects to intercept `BindToObject()` calls:

```
class MonikerProxy : public IMoniker {
    STDMETHOD(BindToObject)(IBindCtx* pbc, IMoniker* pmkToLeft,
        REFIID riidResult, void** ppvResult) {
        HRESULT hr = m_pOriginal->BindToObject(pbc, pmkToLeft, riidResult, ppvResult);

        if (SUCCEEDED(hr) && riidResult == IID_IDispatch) {
            *ppvResult = WrapDispatch((IDispatch*)*ppvResult);
        }
        return hr;
    }
};
```

Figure 9. Implementation of IMoniker.BindToObject that wraps the returned object with an IDispatch Proxy.

This ensures coverage of WMI access and other moniker-based object retrieval.

Implementation details

Interface summary

While standard API hooks can be implemented on a function-by-function basis, COM proxies require implementing all functions of a given interface. The table below details the interfaces and function counts that had to be replicated for this technique to operate.

Interface	Total Methods	Logged	Hooked/Wrapped	Passthrough
IDispatch	7	4	1	2
IEnumVARIANT	7	1	1	5
IClassFactory	5	2	1	2
IMoniker	26	1	1	24

During execution, a script may create dozens or even hundreds of distinct COM objects. For this reason, interface implementations must be class-based and maintain a one-to-one relationship between each proxy instance and the underlying COM object it represents.

While generating this volume of boilerplate code by hand would be daunting, AI-assisted code generation significantly reduced the effort required to implement the complex interface scaffolding.

The real trick with COM interface hooking is object discovery. The initial static API entry points are only the beginning of the mission. Each additional object encountered must be probed, wrapping them recursively to maintain logging.

Thread safety

Multiple threads may create COM objects simultaneously. Proxy tracking uses a critical section to serialize access to the global proxy map:

```cpp
std::map<IDispatch*, DispatchProxy*> g_ProxyMap;
CRITICAL_SECTION g_ProxyCS;

IDispatch* WrapDispatch(IDispatch* pOriginal, const char* name) {
    EnterCriticalSection(&g_ProxyCS);

    // Check if already wrapped to prevent double-wrapping
    if (g_ProxyMap.find(pOriginal) != g_ProxyMap.end()) {
        LeaveCriticalSection(&g_ProxyCS);
        return pOriginal;  // Already wrapped
    }

    DispatchProxy* pProxy = new DispatchProxy(pOriginal, name);
    g_ProxyMap[pOriginal] = pProxy;

    LeaveCriticalSection(&g_ProxyCS);
    return pProxy;
}
```

Figure 10. Thread safety checks in the WrapDispatch function.

Reference counting

Proper COM lifetime management is critical. The proxy maintains separate reference counts and forwards QueryInterface calls appropriately:

```cpp
STDMETHOD_(ULONG, AddRef)() {
    return InterlockedIncrement(&m_refCount);
}

STDMETHOD (ULONG, Release)() {
    LONG count = InterlockedDecrement(&m_refCount);
    if (count == 0) {
        if (m_pOriginal) {
            m_pOriginal->Release();
        }
        delete this;
        return 0;
    }
    return count;
}
```

Figure 11. The IDispatch proxy maintains proper reference counts.

Output analysis

When script code executes with DispatchLogger active, comprehensive logs are generated. Here are excerpts from an actual analysis session:

Object creation and factory interception:

```
[CLSIDFromProgID] 'Scripting.FileSystemObject'-> {0D43FE01-F093-11CF-8940-00A0C9054228}
[CoGetClassObject] FileSystemObject ({0D43FE01-F093-11CF-8940-00A0C9054228}) Context=0x00000015
[CoGetClassObject] Got IClassFactory for FileSystemObject – WRAPPING!
[FACTORY] Created factory proxy for FileSystemObject
[FACTORY] CreateInstance: FileSystemObject requesting Iunknown
[FACTORY] CreateInstance SUCCESS: Object at 0x03AD42D8
[FACTORY] Object supports IDispatch – WRAPPING!
[PROXY] Created proxy #1 for FileSystemObject (Original: 0x03AD42D8)
[FACTORY] !!! Replaced object with proxy!
```

Method invocation with recursive object wrapping

```
[PROXY #1] >>> Invoke: FileSystemObject.GetSpecialFolder (METHOD PROPGET) ArgCount=1
[PROXY #1] Arg[0]: 2
[PROXY #1] <<< Result: IDispatch:0x03AD6C14 (HRESULT=0x00000000)
[PROXY] Created proxy #2 for FileSystemObject.GetSpecialFolder (Original: 0x03AD6C14)
[PROXY #1] !!! Wrapped returned IDispatch as new proxy
[PROXY #2] >>> Invoke: FileSystemObject.GetSpecialFolder.Path (METHOD PROPGET) ArgCount=0
[PROXY #2] <<< Result: "C:\Users\home\AppData\Local\Temp" (HRESULT=0x00000000)
```

WScript.Shell operations

```
[CLSIDFromProgID] 'WScript.Shell'-> {72C24DD5-D70A-438B-8A42-98424B88AFB8}
[CoGetClassObject] WScript.Shell ({72C24DD5-D70A-438B-8A42-98424B88AFB8}) Context=0x00000015
[FACTORY] CreateInstance: WScript.Shell requesting IUnknown
[PROXY] Created proxy #3 for WScript.Shell (Original: 0x03AD04B0)
[PROXY #3] >>> Invoke: WScript.Shell.ExpandEnvironmentStrings (METHOD PROPGET) ArgCount=1
[PROXY #3] Arg[0]: "%WINDIR%"
[PROXY #3] <<< Result: "C:\WINDOWS" (HRESULT=0x00000000)
```

Dictionary operations

```
[CLSIDFromProgID] 'Scripting.Dictionary'-> {EE09B103-97E0-11CF-978F-00A02463E06F}
[PROXY] Created proxy #4 for Scripting.Dictionary (Original: 0x03AD0570)
[PROXY #4] >>> Invoke: Scripting.Dictionary.Add (METHOD) ArgCount=2
[PROXY #4] Arg[0]: "test"
[PROXY #4] Arg[1]: "value"
[PROXY #4] <<< Result: (void) HRESULT=0x00000000
[PROXY #4] >>> Invoke: Scripting.Dictionary.Item (METHOD PROPGET) ArgCount=1
[PROXY #4] Arg[0]: "test"
[PROXY #4] <<< Result: "value" (HRESULT=0x00000000)
```

This output provides:

- Complete object instantiation audit trail with CLSIDs
- All method invocations with method names resolved via ITypeInfo
- Full parameter capture including strings, numbers, and object references
- Return value logging including nested objects
- Object relationship tracking showing parent-child relationships
- log post processing allows for high fidelity command retrieval

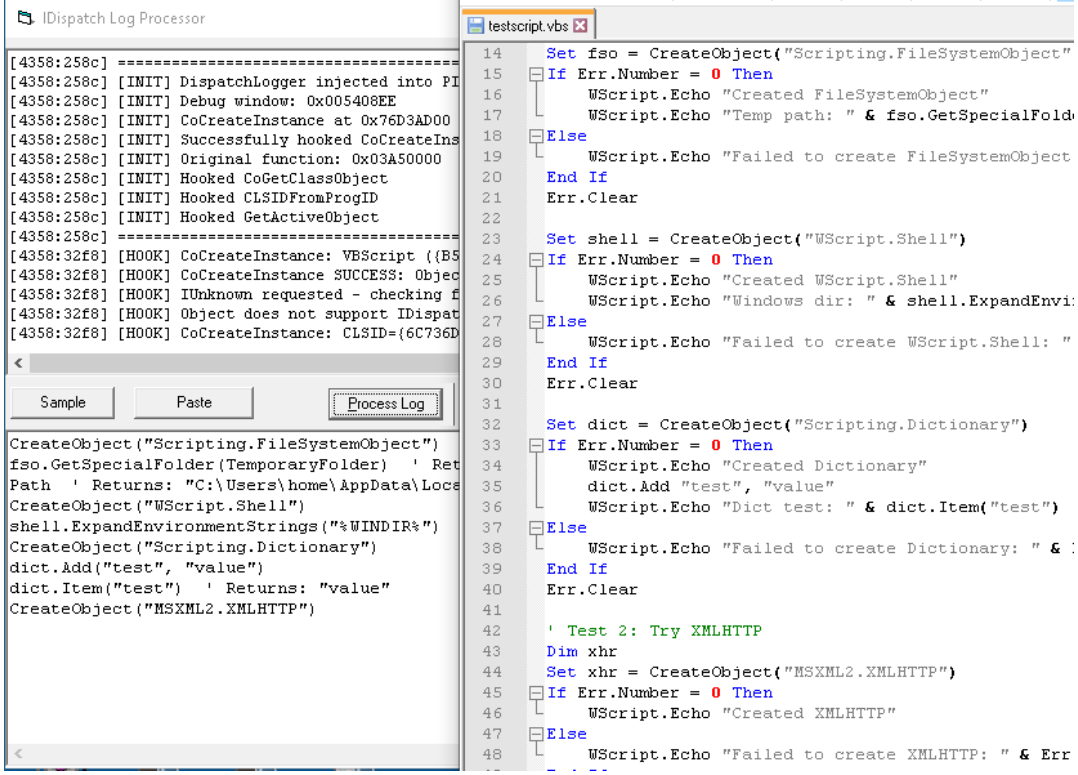

Figure 12. Raw log output, parsed results, and original script.

Deployment

DispatchLogger is implemented as a dynamic-link library (DLL) that can be injected into target processes.

Once loaded, the DLL:

1. Locates debug output window or uses OutputDebugString
2. Initializes critical sections for thread safety
3. Hooks COM API functions using inline hooking engine
4. Begins transparent logging

No modifications to the target script or runtime environment are required.

Advantages over alternative approaches

Approach	Coverage	Semantic visibility	Detection risk
Static analysis	Encrypted/obfuscated scripts missed	No runtime behavior	N/A
API monitoring	Low-level calls only	Missing high-level intent	Medium
Memory forensics	Point-in-time snapshots	No call sequence context	Low
Debugger tracing	Manual breakpoints required	Analyst-driven, labor-intensive	High
DispatchLogger	Complete late bound automation layer	Full semantic context	None

DispatchLogger provides advantages for:

- **WMI-based attacks**: Complete query visibility, object enumeration, method invocation tracking
- **Living-off-the-land (LOTL) techniques**: Office automation abuse, scheduled task manipulation, registry operations
- **Fileless malware**: PowerShell/COM hybrid attacks, script-only payloads
- **Persistence mechanisms**: COM-based autostart mechanisms, WMI event subscriptions
- **Data exfiltration**: Filesystem operations, network object usage, database access via ADODB
- **Obsfuscation bypass**: Working at the COM layer, object, method names, and arguments are already fully resolved.

Performance considerations

Proxy overhead is minimal:

- Each Invoke call adds one virtual function dispatch.
- In the demo, logging I/O occurs via IPC.
- Object wrapping is O(1) with hash map lookup.
- There is no performance impact on non-COM operations.

In testing with real malware samples, execution time differences were negligible.

Limitations

Current implementation constraints:

- **IDispatchEx**: Not currently implemented (not used by most malware)
- **IClassFactory2+**: Not currently implemented (may impact browser/HTA/WinRT)
- **Out-of-process COM**: DCOM calls require separate injection into server process
- **Multi-threaded race conditions**: Rare edge cases in concurrent object creation
- **Type library dependencies**: Method name resolution requires registered type libraries
- **Process following:** This sample code does not attempt to follow child processes
- **64bit Support**: Has only had minimal testing with 64 bit targets

The sample code included with this article is a general purpose tool and proof of concept. It has not been tested at scale and does not attempt to prevent logging escapes.

Operational usage

Typical analysis workflow:

1. Prepare isolated analysis VM
2. Inject DispatchLogger into target process
3. Execute malware sample
4. Review comprehensive COM interaction log
5. Identify key objects, methods, and parameters
6. Extract IOCs and behavioral signatures

The tool has been tested against:

- VBScript & Jscript using Windows Script Host (cscript/wscript)
- PowerShell scripts
- basic tests against .NET and Runtime Callable Wrappers (RCW)
- VB6 executables with late bound calls and Get/CreateObject

Background and Prior Work

The techniques presented in this article emerged from earlier experimentation with IDispatch while developing a JavaScript engine capable of exposing dynamic JavaScript objects as late-bound COM objects. That work required deep control over name resolution, property creation, and IDispatch::Invoke handling. This framework allowed JavaScript objects to be accessed and modified transparently from COM clients.

The experience gained from that effort directly informed the transparent proxying and recursive object wrapping techniques used in DispatchLogger.

Conclusion

DispatchLogger addresses a significant gap in script-based malware analysis by providing deep, semantic-level visibility into COM automation operations. Through transparent proxy interception at the COM instantiation boundary, recursive object wrapping, and comprehensive logging, analysts gain great insight into malware behavior without modifying samples or introducing detection vectors.

The implementation demonstrates that decades-old COM architecture, when properly instrumented, provides powerful analysis capabilities for modern threats. By understanding COM internals and applying transparent proxying patterns, previously opaque script behavior becomes highly observable.