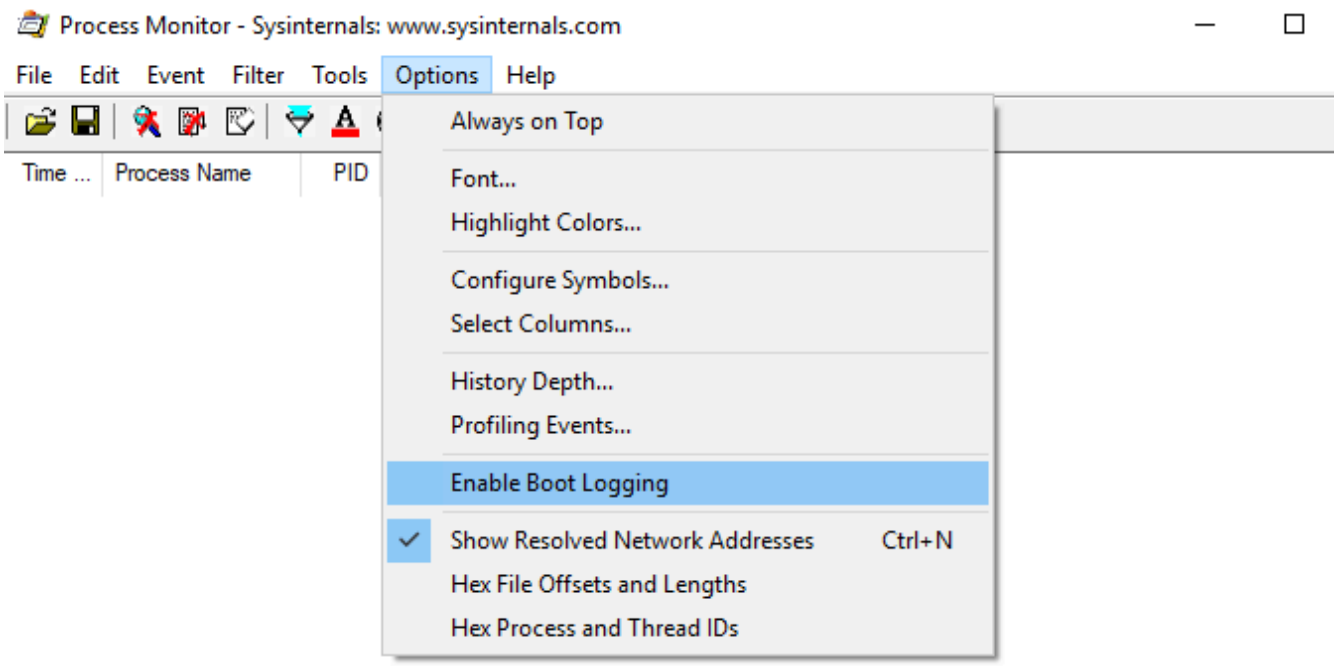
Persistence: Component Object Model (COM) hijacking

stmxcsr.com/persistence/com-hijacking.html

May 4, 2026



This post discusses COM hijacking and how COM can be abused and utilized as a means of payload execution and persistence on a target system during the post-exploitation stage. The technique is documented as T1546.015 in the MITRE ATT&CK knowledge base [6] and the related research was first published in virusBULLETIN [1]. The technique has been observed in malware [7]. The article is not intended to be a complete guide on how COM actually works, as the inner machinations of COM are an enigma to the author.

The post is divided in the following sections:

Information about COM

COM [2] is a platform-independent, distributed, object-oriented system for creating binary software components that can interact and consists of the following parts: host system, interfaces, servers, registry, Service Control Manager and structure storage protocol.

On its host system, COM maintains a registration database of all the CLSIDs for the COM objects installed on the system. The registration database is a mapping between each CLSID and the location of the DLL or EXE that houses the corresponding class. COM queries this database whenever a caller wants to create an instance of a COM class. The caller needs to know only the CLSID to request a new instance of the class. A CLSID Identifies a COM class

with a unique 128-bit Class ID and associates a class with a particular deployment in the file system, which for Windows is a DLL or EXE.

The interaction between a COM object and its callers is modeled as a client/server relationship. The client is the caller that requests a COM object from the system, and the server is the module that houses COM objects that provides services to clients. A COM client is whatever code or object gets a pointer to a COM server and uses its services by calling the methods of its interfaces. There are two main types of servers, in-process and out-of-process. In-process servers are implemented in a dynamic linked library (DLL), and out-of-process servers are implemented in an executable file (EXE).

Abuse options

To abuse the COM functionality there are three options:

- Register a (COM) user object that will get loaded prior to (COM) machine objects
- Replace a DLL or EXE on existing COM classes
- Register a class that has not yet been defined

The first two options may affect system's functionality and stability, since functional objects are replaced. However, we may look for objects that are not backed by a resource present on the system. This may happen when for example an application has been uninstalled but COM Classes are left behind (ghost COM classes).

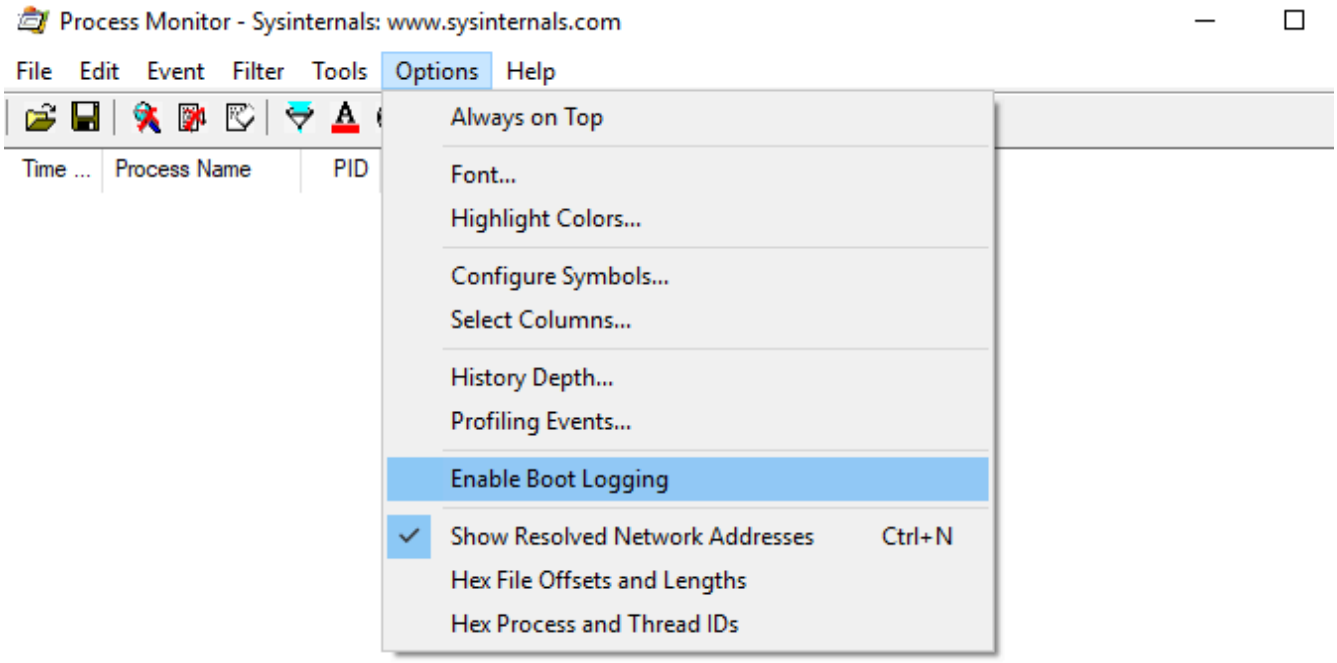
In relation to the first option, COM loads user COM objects registered in *HKEY_CURRENT_USER\Software\Classes\CLSID\ prior* to machine objects located in *HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID*. The *HKEY_LOCAL_MACHINE\SOFTWARE\Classes* key corresponds to the *HKEY_CLASSES_ROOT*. By registering a user object we can achieve the loading of this object first and thus abuse the loading order.

The second option is pretty straightforward: replace a DLL or an EXE with the payload.

The third option requires further research. In this article the focus is on this option. Various components of the operating system request COM objects that haven't yet been defined in registry. Let's call these requests orphaned requests. Using SysInternals' *Procmon* we can harvest the orphaned requests, register COM objects for them, map them to our payload (either DLL or EXE) and eventually achieve persistent execution.

Harvesting orphaned CLSIDs

To collect requests for orphaned CLSIDs we utilize Procmon. To collect as much data as possible, it's suggested to enable boot logging. With this option, all events taking place during boot will be recorded.

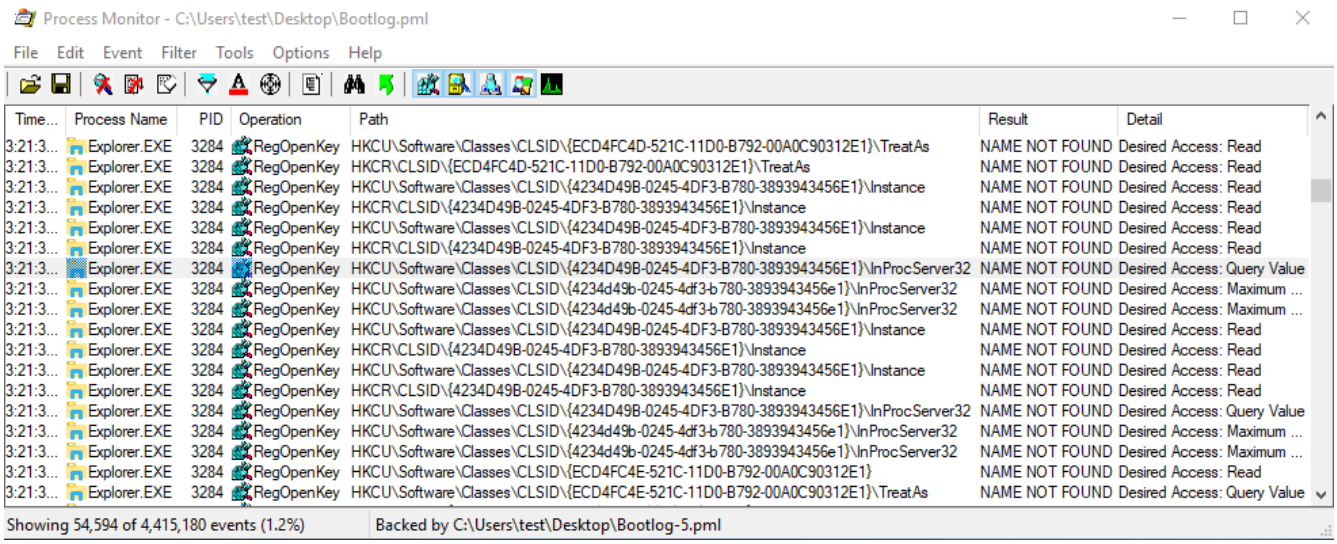


Enable boot logging on Procmon

After the system boots, open Procmon and see the recorded results. We are using three filters in order to limit down the results to what is important to us. We can do this by looking in the results for the following three:

- Operation is RegOpenKey
- Path contains the string CLSID
- Result is NAME NOT FOUND

After we apply the filters we get a view like the following:



Results for Procmon filters

Results show that we got 54,594 RegOpenKey events that match the set filters. Take some time to go through the results and observe the processes that generated the events as well as the Paths. The recorded results are actually the requests made by a client to the COM server. What they have in common? They all are for orphaned CLSIDs.

Within the results we see Paths that end with *InprocServer32*. According to Microsoft documentation [3], *InprocServer32* registers a 32-bit in-process server and specifies the threading model of the apartment the server can run in. *InprocServer32* keys should be pointing to DLLs. If they don't exist (aka *NAME NOT FOUND*) here is our chance to create them…and abuse COM!

Knowing this, we can further filter the results so that we focus more on keys that end with *InprocServer32*.

HKLM and HKCR

Would be good to point out that the results contain two root registry keys: *HKEY_CURRENT_USER* (HKCU) and *HKEY_CLASSES_ROOT* (HKCR). As we are going to create registry keys, we have to first check what privileges are required to create these keys. PowerShell's *Get-Acl* command can help get the security descriptor and determine which user accounts can write to these keys.

```
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\test> get-acl HKCU: | format-list


Path   : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\
Owner  : BUILTIN\Administrators
Group  : NT AUTHORITY\SYSTEM
Access : NT AUTHORITY\RESTRICTED Allow  ReadKey
         NT AUTHORITY\SYSTEM Allow  FullControl
         BUILTIN\Administrators Allow  FullControl
         DESKTOP-[redacted]\test Allow  FullControl
         APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES Allow  ReadKey
         S-1-15-3-1024-1065365936-[redacted]-3511738428-1654721687-432734479-3232135806-4053264122-3456934681 Allow
         ReadKey
Audit  :
Sddl   : O:BAG:SYD:P(A;OICI;KR;;;RC)(A;OICI;KA;;;SY)(A;OICI;KA;;;BA)(A;OICI;KA;;;S-1-5-21-4162225231-4122751953-2322023
         677-1001)(A;;KR;;;AC)(A;;KR;;;S-1-15-3-1024-1065365936-[redacted]-3511738428-1654721687-432734479-3232135806-4
         053264122-3456934681)
```

HKCU Security Descriptor

```
PS C:\Users\test> get-acl HKLM: | format-list


Path   : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\
Owner  : NT AUTHORITY\SYSTEM
Group  : NT AUTHORITY\SYSTEM
Access : Everyone Allow  ReadKey
         NT AUTHORITY\RESTRICTED Allow  ReadKey
         NT AUTHORITY\SYSTEM Allow  FullControl
         BUILTIN\Administrators Allow  FullControl
         APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES Allow  ReadKey
         S-1-15-3-1024-1065365936-[redacted]-3511738428-1654721687-432734479-3232135806-4053264122-3456934681 Allow
         ReadKey
Audit  :
Sddl   : O:SYG:SYD:(A;CI;KR;;;WD)(A;CI;KR;;;RC)(A;CI;KA;;;SY)(A;CI;KA;;;BA)(A;CI;KR;;;AC)(A;CI;KR;;;S-1-15-3-1024-10653
         65936-[redacted]-3511738428-1654721687-432734479-3232135806-4053264122-3456934681)
```

HKLM Security Descriptor

We confirm that low privileged users can write to the root key HKCU. As HLCR is just a link to HKLM, we include the security descriptor of HKLM. So, assuming a low privileged user account is compromise, the way to go is to create a registry key in HKCU.

Goal setting

Thus far, we have identified requests for keys that don't exist. We want to target specific keys that map CLSIDs to DLLs, so we're looking for InprocServer32 keys. The next step is to get a CLSID, map it to a DLL then reboot the machine and see if the payload gets executed. The process consists of trial and error. You may choose a CLSID that at the end won't launch your payload.

Create the (registry) keys to heaven

After a few tries, we picked up the CLSID {4234D49B-0245-4DF3-B780-3893943456E1}. Searching on Google or Bing for this CLSID, reveals [4] that this CLSID is used to open the Applications directory. You can simply hit *Ctrl+r* to open the run box and the following, in order to confirm that Applications is associated with the CLSID:

> explorer shell:::{4234d49b-0245-4df3-b780-3893943456e1}

This means that everytime a user opens the Applications OR the operating system itself is querying this CLSID, our payload will get executed too.

Two registry keys and two values are required to achieve our goal. We create the following keys:

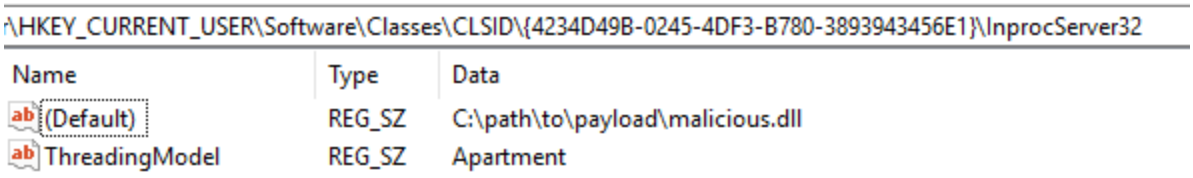
> HKEY_CURRENT_USER\Software\Classes\CLSID\{4234D49B-0245-4DF3-B780-3893943456E1}

> HKEY_CURRENT_USER\Software\Classes\CLSID\{4234D49B-0245-4DF3-B780-3893943456E1}\InprocServer32

And the following values:

Value Name	Value data
(Default)	<path to payload DLL>
ThreadingModel	Apartment

Note that the filename of the DLL does *NOT* have to end with *.dll*. A different extension can be used - even a random one. So we can plant the DLL in a directory with pictures for example or with other system files.



Screenshot of CLSID {4234D49B-0245-4DF3-B780-3893943456E1} from Regedit

In this test, we used the PoC DLL payload that spawns Windows Calculator. The effect is immediate, multiple Calculators are spawned as soon as we create the registry key. The parent process of the instances of Calculator is *explorer.exe*, which is what we expect if we consider that the request we captured with Procmon was generated by this process. If this payload DLL is a Meterpreter or a Cobalt Strike Beacon, you'll get back multiple connections back from this host. This happens because the COM client makes multiple requests which are now served by the COM server we set up.

Dealing with the trouble: Multiple Instances

Multiple requests coming from the COM client are now being served and multiple instances of Calculator are being spawned from *explorer.exe*. And this is where the trouble begins. As you can imagine, multiple Calculators will be keep launching consuming system resources. What we can do to address this, is to account for this behavior in our code. That is, to check if our process is already running before we launch it. If it is running, then the code that spawns Calculator won't be executed again. Right, just like a switch.

In the following code, after getting some help from docs.microsoft.com [5], we get a list of active processes using the CreateToolhelp32Snapshot API and if the process we want to launch is already running we don't execute Calculator. This switch-like behavior reduces the noise to the bare minimum and leaves only a single instance of Calculator running.

```
#include <windows.h>
#include <stdio.h>
#include <tlhelp32.h>

#define DllExport __declspec(dllexport)

DllExport void __stdcall SpawnCalc(void)
{
        WinExec("calc", 0);
}

BOOL calc_check()
{
        HANDLE hProcessSnap;
        PROCESSENTRY32 pe32;

        // take a snapshot of all processes in the system
        hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
        if (hProcessSnap == INVALID_HANDLE_VALUE)
        {
                printf("[-] CreateToolhelp32Snaphost has failed: %d", GetLastError());
                return 0;
        }

        pe32.dwSize = sizeof( PROCESSENTRY32 );

        if ( !Process32First(hProcessSnap, &pe32) )
        {
                printf("[-] Process32First has failed!");
                CloseHandle(hProcessSnap);
                return 0;
        }
```

```
        do
        {
                if ( strcmp(pe32.szExeFile, "Calculator.exe") == 0 )
                {
                        return 0;
                }
        } while( Process32Next(hProcessSnap, &pe32) );

        CloseHandle(hProcessSnap);

        return 1;
}

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
        switch (fdwReason)
        {
                case DLL_PROCESS_ATTACH:
                        printf("[+] Process Attached\n");
                        // if Calculator.exe is not running, then launch it
                        if (calc_check())
                        {
                                SpawnCalc();
                        }
                        printf("[+] Function in DLL was executed\n");
                        break;
                case DLL_THREAD_ATTACH:
                case DLL_PROCESS_DETACH:
                case DLL_THREAD_DETACH:
                        break;
        }

        return 1;
}
```

Observations

After reducing the noise (aka multiple instances of Calculator that hang the system), using Process Hacker, we observed that *calc.exe* is indeed spawned by *explorer.exe*. The *calc.exe* is immediately terminated. However, at the same time an instance of *Calculator.exe* is spawned by the parent process *svchost.exe*. This instance is the one that remains active. The image file name of *Calculator.exe* is *C:\Program Files\WindowsApps\Microsoft.WindowsCalculator_10.1905.30.0_x64__8wekyb3d8bbwe\Calculator.exe*.

Tools

- Visual Studio
- Procmon
- Process Hacker
- Regedit

References

[1] https://www.virusbulletin.com/conference/vb2011/abstracts/dangers-user-com-objects-windows/

[2] https://docs.microsoft.com/en-us/windows/win32/com/com-technical-overview

[3] https://docs.microsoft.com/en-us/windows/win32/com/inprocserver32

[4] https://www.tenforums.com/tutorials/3123-clsid-key-guid-shortcuts-list-windows-10-a.html

[5] https://docs.microsoft.com/en-us/windows/win32/toolhelp/taking-a-snapshot-and-viewing-processes

[6] https://attack.mitre.org/techniques/T1546/015/

[7] https://blog.talosintelligence.com/cyber-conflict-decoy-document/

** Any feedback on this article is welcome, method of communication is listed in the Contact page*

tags: #persistence