← Shellcode: In-Memory Execution of DLL

Windows Process Injection: Winsock Helper Functions (WSHX) →

Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL

Posted on July 21, 2019

Introduction

A DynaCall() Function for Win32 was published in the August 1998 edition of Dr.Dobbs Journal. The author, Ton Plooy, provided a function in C that allows an interpreted language such as VBScript to call external DLL functions via a registered COM object. An

Search

Recent Posts

- MiniDumpWriteDump via COM+ Services DLL
- Windows Process Injection: Asynchronous Procedure Call (APC)
- Windows Process Injection: KnownDlls Cache Poisoning
- Windows Process Injection: Tooltip or Common Controls
- Windows Process Injection: Breaking BaDDEr
- Windows Process Injection: DNS Client API

Automation Object for Dynamic DLL Calls published in November 1998 by Jeff Stong built upon this work to provide a more complete project which he called DynamicWrapper. In 2011, Blair Strang wrote a tool called vbsmem that used DynamicWrapper to execute shellcode from VBScript. DynamicWrapper was the source of inspiration for another tool called DynamicWrapperX that appeared in 2008 and it too was used to execute shellcode from VBScript by Casey Smith.

The May 2019 update of Defender Application Control included a number of new policies, one of which is "COM object registration". Microsoft states the purpose of this policy is to enforce "a built-in allow list of COM object registrations to reduce the risk introduced from certain powerful COM objects." Are they referring to DynamicWrapper? Possibly, but what about unregistered COM objects? Robert Freeman/IBM demonstrated in 2007 that unregistered COM objects may be useful for obfuscation purposes. His Virus Bulletin presentation Novel code obfuscation with COM doesn't provide any proof-of-concept code, but does demonstrate the potential to misuse the IActiveScript interface for Dynamic DLL calls without COM registration.

Windows Script Host (WSH)

WSH is an automation technology available since Windows 95 that was popular among developers prior to the release of the .NET Framework in 2002. It was primarily used for generation of dynamic content like Active Server Pages (ASP) written in JScript or VBScript. As .NET superseded this technology, much of the wisdom developers acquired about Active Scripting up until 2002 slowly disappeared from the internet. One post that was recommended quite frequently on developer forums is the Active X FAQ by Mark Baker, which answers most questions developers have about the IActiveScript interface.

- Windows Process Injection: Multiple Provider Router (MPR) DLL and Shell Notifications
- Windows Process Injection: Winsock Helper Functions (WSHX)
- Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL
- Shellcode: In-Memory Execution of DLL
- Windows Process Injection : Windows Notification Facility
- How Red Teams Bypass AMSI and WLDP for .NET Dynamic Code
- Windows Process Injection: KernelCallbackTable used by FinFisher / FinSpy
- Windows Process Injection: CLIPBRDWNDCLASS
- Shellcode: Using the Exception Directory to find GetProcAddress
- Shellcode: Loading .NET Assemblies From Memory
- Windows Process Injection: WordWarping, Hyphentension, AutoCourgette, Streamception, Oleum, ListPlanting, Treepoline
- Shellcode: A reverse shell for Linux in C with support for TLS/SSL
- Windows Process Injection: Print Spooler
- How the Lopht (probably) optimized attack against the LanMan hash.

Enumerating Script Engines

Can be performed in at least two ways.

1. Each Class Identifier in `HKEY_CLASSES_ROOT\CLSID\` that contains a subkey called `OLEScript` can be used with Windows Script Hosting.
2. The Component Categories Manager can enumerate CLSID for category identifiers `CATID_ActiveScript` or `CATID_ActiveScriptParse`.

Below is a snippet of code for displaying active script engines using the second approach. See full version here.

```
void DisplayScriptEngines(void) {
    ICatInformation *pci = NULL;
    IEnumCLSID      *pec = NULL;
    HRESULT         hr;
    CLSID           clsid;
    OLECHAR         *progID, *idStr, path[MAX_PATH], desc[MAX_PATH];

    // initialize COM
    CoInitialize(NULL);

    // obtain component category manager for this machine
    hr = CoCreateInstance(
      CLSID_StdComponentCategoriesMgr,
      0, CLSCTX_SERVER, IID_ICatInformation,
      (void**)&pci);

    if(hr == S_OK) {
```

```
        // obtain list of script engine parsers
        hr = pci->EnumClassesOfCategories(
          1, &CATID_ActiveScriptParse, 0, 0, &pec);

        if(hr == S_OK) {
          // print each CLSID and Program ID
          for(;;) {
            ZeroMemory(path, ARRAYSIZE(path));
            ZeroMemory(desc, ARRAYSIZE(desc));

            hr = pec->Next(1, &clsid, 0);
            if(hr != S_OK) {
              break;
            }
            ProgIDFromCLSID(clsid, &progID);
            StringFromCLSID(clsid, &idStr);
            GetProgIDInfo(idStr, path, desc);

            wprintf(L"\n************************************\n");
            wprintf(L"Description : %s\n", desc);
            wprintf(L"CLSID       : %s\n", idStr);
            wprintf(L"Program ID  : %s\n", progID);
            wprintf(L"Path of DLL : %s\n", path);

            CoTaskMemFree(progID);
            CoTaskMemFree(idStr);
          }
          pec->Release();
        }
        pci->Release();
      }
    }
```


- Shellcode: x86 optimizations part 1
- WanaCryptor File Encryption and Decryption
- Shellcode: Dual Mode (x86 + amd64) Linux shellcode
- Shellcode: Fido and how it resolves GetProcAddress and LoadLibraryA
- Shellcode: Dual mode PIC for x86 (Reverse and Bind Shells for Windows)
- Shellcode: Solaris x86
- Shellcode: Mac OSX amd64
- Shellcode: Resolving API addresses in memory
- Shellcode: A Windows PIC using RSA-2048 key exchange, AES-256, SHA-3
- Shellcode: Execute command for x32/x64 Linux / Windows / BSD
- Shellcode: Detection between Windows/Linux/BSD on x86 architecture
- Shellcode: FreeBSD / OpenBSD amd64
- Shellcode: Linux amd64
- Shellcodes: Executing Windows and Linux Shellcodes
- DLL/PIC Injection on Windows from Wow64 process
- Asmcodes: Platform Independent PIC for Loading DLL and Executing Commands

The output of this code on a system with ActivePerl and ActivePython installed :

```
************************************
Description : JScript Language
CLSID       : {16D51579-A30B-4C8B-A276-0FF4DC41E755}
Program ID  : JScript
Path of DLL : C:\Windows\System32\jscript9.dll

************************************
Description : XML Script Engine
CLSID       : {989D1DC0-B162-11D1-B6EC-D27DDCF9A923}
Program ID  : XML
Path of DLL : C:\Windows\System32\msxml3.dll

************************************
Description : VB Script Language
CLSID       : {B54F3741-5B07-11CF-A4B0-00AA004A55E8}
Program ID  : VBScript
Path of DLL : C:\Windows\System32\vbscript.dll

************************************
Description : VBScript Language Encoding
CLSID       : {B54F3743-5B07-11CF-A4B0-00AA004A55E8}
Program ID  : VBScript.Encode
Path of DLL : C:\Windows\System32\vbscript.dll

************************************
Description : JScript Compact Profile (ECMA 327)
```

```
CLSID       : {CC5BBEC3-DB4A-4BED-828D-08D78EE3E1ED}
Program ID  : JScript.Compact
Path of DLL : C:\Windows\System32\jscript.dll

************************************
Description : Python ActiveX Scripting Engine
CLSID       : {DF630910-1C1D-11D0-AE36-8C0F5E000000}
Program ID  : Python.AXScript.2
Path of DLL : pythoncom36.dll

************************************
Description : JScript Language
CLSID       : {F414C260-6AC0-11CF-B6D1-00AA00BBBB58}
Program ID  : JScript
Path of DLL : C:\Windows\System32\jscript.dll

************************************
Description : JScript Language Encoding
CLSID       : {F414C262-6AC0-11CF-B6D1-00AA00BBBB58}
Program ID  : JScript.Encode
Path of DLL : C:\Windows\System32\jscript.dll

************************************
Description : PerlScript Language
CLSID       : {F8D77580-0F09-11D0-AA61-3C284E000000}
Program ID  : PerlScript
Path of DLL : C:\Perl64\bin\PerlSE.dll
```

The PerlScript and Python scripting engines are provided by ActiveState. I would recommend using {16D51579-A30B-4C8B-A276-0FF4DC41E755} for JavaScript.

C Implementation of IActiveScript

During research into IActiveScript, I found [COM in plain C, part 6] by Jeff Glatt to be helpful. The following code is the bare minimum required to execute VBS/JS files and does not support WSH objects. See [here] for the full source.

```
VOID run_script(PWCHAR lang, PCHAR script) {
    IActiveScriptParse     *parser;
    IActiveScript          *engine;
    MyIActiveScriptSite    mas;
    IActiveScriptSiteVtbl  vft;
    LPVOID                 cs;
    DWORD                  len;
    CLSID                  langId;
    HRESULT                hr;

    // 1. Initialize IActiveScript based on language
    CLSIDFromProgID(lang, &langId);
    CoInitializeEx(NULL, COINIT_MULTITHREADED);

    CoCreateInstance(
      &langId, 0, CLSCTX_INPROC_SERVER,
      &IID_IActiveScript, (void **)&engine);

    // 2. Query engine for script parser and initialize
    engine->lpVtbl->QueryInterface(
        engine, &IID_IActiveScriptParse,
        (void **)&parser);

    parser->lpVtbl->InitNew(parser);
```

```
// 3. Initialize IActiveScriptSite interface
vft.QueryInterface      = (LPVOID)QueryInterface;
vft.AddRef              = (LPVOID)AddRef;
vft.Release             = (LPVOID)Release;
vft.GetLCID             = (LPVOID)GetLCID;
vft.GetItemInfo         = (LPVOID)GetItemInfo;
vft.GetDocVersionString = (LPVOID)GetDocVersionString;
vft.OnScriptTerminate   = (LPVOID)OnScriptTerminate;
vft.OnStateChange       = (LPVOID)OnStateChange;
vft.OnScriptError       = (LPVOID)OnScriptError;
vft.OnEnterScript       = (LPVOID)OnEnterScript;
vft.OnLeaveScript       = (LPVOID)OnLeaveScript;

mas.site.lpVtbl     = (IActiveScriptSiteVtbl*)&vft;
mas.siteWnd.lpVtbl  = NULL;
mas.m_cRef          = 0;

engine->lpVtbl->SetScriptSite(
    engine, (IActiveScriptSite *)&mas);

// 4. Convert script to unicode and execute
len = MultiByteToWideChar(
  CP_ACP, 0, script, -1, NULL, 0);

len *= sizeof(WCHAR);

cs = malloc(len);

len = MultiByteToWideChar(
  CP_ACP, 0, script, -1, cs, len);
```

```
    parser->lpVtbl->ParseScriptText(
         parser, cs, 0, 0, 0, 0, 0, 0, 0, 0);

    engine->lpVtbl->SetScriptState(
         engine, SCRIPTSTATE_CONNECTED);

    // 5. cleanup
    parser->lpVtbl->Release(parser);
    engine->lpVtbl->Close(engine);
    engine->lpVtbl->Release(engine);
    free(cs);
}
```

x86 Assembly

Just for illustration, here's something similar in x86 assembly with some limitations imposed: The script should not exceed 64KB, the UTF-16 conversion only works with ANSI(latin alphabet) characters, and the language (VBS or JS) must be predefined before assembling. When declaring a local variable on the stack that exceeds 4KB, compilers such as GCC and MSVC insert code to perform stack probing which allows the kernel to expand the amount of stack memory available to a thread. There are of course compiler/linker switches to increase the reserved size if you wanted to prevent stack probing, but they are rarely used in practice. Each thread on Windows initially has 16KB of stack available by default as you can see by subtracting the value of StackLimit from StackBase found in the Thread Environment Block (TEB).

```
0:004> !teb
TEB at 000000f4018bf000
    ExceptionList:        0000000000000000
    StackBase:            000000f401c00000
    StackLimit:           000000f401bfc000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 000000f4018bf000
    EnvironmentPointer:   0000000000000000
    ClientId:             0000000000001940 . 000000000000067c
    RpcHandle:            0000000000000000
    Tls Storage:          0000000000000000
    PEB Address:          000000f40185a000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0

0:004> ? 000000f401c00000 - 000000f401bfc000
Evaluate expression: 16384 = 00000000`00004000
```

The assembly code initially used VirtualAlloc to allocate enough space, but since this code is unlikely to be used for anything practical, the stack is used instead.

```
; In-Memory execution of VBScript/JScript using 392 bytes of x86 asse
; Odzhan
```

```
        %include "ax.inc"

        %define VBS

        bits   32

        %ifndef BIN
          global run_scriptx
          global _run_scriptx
        %endif

run_scriptx:
_run_scriptx:
        pop    ecx             ; ecx = return address
        pop    eax             ; eax = script parameter
        push   ecx             ; save return address
        cdq                    ; edx = 0
        ; allocate 128KB of stack.
        push   32              ; ecx = 32
        pop    ecx
        mov    dh, 16          ; edx = 4096
        pushad                 ; save all registers
        xchg   eax, esi        ; esi = script
alloc_mem:
        sub    esp, edx        ; subtract size of page
        test   [esp], esp      ; stack probe
        loop   alloc_mem       ; continue for 32 pages
        mov    edi, esp        ; edi = memory
        xor    eax, eax
utf8_to_utf16:                 ; YMMV. Prone to a stack overflow.
        cmp    byte[esi], al   ; ? [esi] == 0
        movsb                  ; [edi] = [esi], edi++, esi++
```

```
        stosb                   ; [edi] = 0, edi++
        jnz     utf8_to_utf16   ;
        stosd                   ; store 4 nulls at end
        and     edi, -4         ; align by 4 bytes
        call    init_api        ; load address of invoke_api onto stack
        ; ******************************
        ; INPUT: eax contains hash of API
        ; Assumes DLL already loaded
        ; No support for resolving by ordinal or forward references
        ; ******************************
invoke_api:
        pushad
        push    TEB.ProcessEnvironmentBlock
        pop     ecx
        mov     eax, [fs:ecx]
        mov     eax, [eax+PEB.Ldr]
        mov     edi, [eax+PEB_LDR_DATA.InLoadOrderModuleList + LIST_ENTF
        jmp     get_dll
next_dll:
        mov     edi, [edi+LDR_DATA_TABLE_ENTRY.InLoadOrderLinks + LIST_E
get_dll:
        mov     ebx, [edi+LDR_DATA_TABLE_ENTRY.DllBase]
        mov     eax, [ebx+IMAGE_DOS_HEADER.e_lfanew]
        ; ecx = IMAGE_DATA_DIRECTORY[IMAGE_DIRECTORY_ENTRY_EXPORT].Virt
        mov     ecx, [ebx+eax+IMAGE_NT_HEADERS.OptionalHeader + \
                             IMAGE_OPTIONAL_HEADER32.DataDirectory + \
                             IMAGE_DIRECTORY_ENTRY_EXPORT * IMAGE_DATA_
                             IMAGE_DATA_DIRECTORY.VirtualAddress]
        jecxz   next_dll
        ; esi = offset IMAGE_EXPORT_DIRECTORY.NumberOfNames
        lea     esi, [ebx+ecx+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
        lodsd
```

```
        xchg    eax, ecx
        jecxz   next_dll        ; skip if no names
        ; ebp = IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
        lodsd
        add     eax, ebx        ; ebp = RVA2VA(eax, ebx)
        xchg    eax, ebp        ;
        ; edx = IMAGE_EXPORT_DIRECTORY.AddressOfNames
        lodsd
        add     eax, ebx        ; edx = RVA2VA(eax, ebx)
        xchg    eax, edx        ;
        ; esi = IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
        lodsd
        add     eax, ebx        ; esi = RVA2VA(eax, ebx)
        xchg    eax, esi
get_name:
        pushad
        mov     esi, [edx+ecx*4-4] ; esi = AddressOfNames[ecx-1]
        add     esi, ebx           ; esi = RVA2VA(esi, ebx)
        xor     eax, eax           ; eax = 0
        cdq                        ; h = 0
hash_name:
        lodsb
        add     edx, eax
        ror     edx, 8
        dec     eax
        jns     hash_name
        cmp     edx, [esp + _eax + pushad_t_size]   ; hashes match?
        popad
        loopne get_name             ; --ecx && edx != hash
        jne     next_dll            ; get next DLL
        movzx   eax, word [esi+ecx*2] ; eax = AddressOfNameOrdinals[ecx]
        add     ebx, [ebp+eax*4]      ; ecx = base + AddressOfFunctions[
```

```
        mov     [esp+_eax], ebx
        popad                           ; restore all
        jmp     eax
_ds_section:
        ; ---------------------
        db      "ole32", 0, 0, 0
co_init:
        db      "CoInitializeEx", 0
co_init_len equ $-co_init
co_create:
        db      "CoCreateInstance", 0
co_create_len equ $-co_create
        ; IID_IActiveScript
        ; IID_IActiveScriptParse32 +1
        dd      0xbb1a2ae1
        dw      0xa4f9, 0x11cf
        db      0x8f, 0x20, 0x00, 0x80, 0x5f, 0x2c, 0xd0, 0x64
  %ifdef VBS
        ; CLSID_VBScript
        dd      0xB54F3741
        dw      0x5B07, 0x11cf
        db      0xA4, 0xB0, 0x00, 0xAA, 0x00, 0x4A, 0x55, 0xE8
  %else
        ; CLSID_JScript
        dd      0xF414C260
        dw      0x6AC0, 0x11CF
        db      0xB6, 0xD1, 0x00, 0xAA, 0x00, 0xBB, 0xBB, 0x58
  %endif
_QueryInterface:
        mov     eax, E_NOTIMPL      ; return E_NOTIMPL
        retn    3*4
_AddRef:
```

```
_Release:
      pop    eax                 ; return S_OK
      push   eax
      push   eax
_GetLCID:
_GetItemInfo:
_GetDocVersionString:
      pop    eax                 ; return S_OK
      push   eax
      push   eax
_OnScriptTerminate:
      xor    eax, eax            ; return S_OK
      retn   3*4
_OnStateChange:
_OnScriptError:
      jmp    _GetDocVersionString
_OnEnterScript:
_OnLeaveScript:
      jmp    _Release
init_api:
      pop    ebp
      lea    esi, [ebp + (_ds_section - invoke_api)]

      ; LoadLibrary("ole32");
      push   esi                     ; "ole32", 0
      mov    eax, 0xFA183D4A         ; eax = hash("LoadLibraryA")
      call   ebp                     ; invoke_api(eax)
      xchg   ebx, eax                ; ebp = base of ole32
      lodsd                          ; skip "ole32"
      lodsd

      ; _CoInitializeEx = GetProcAddress(ole32, "CoInitializeEx");
```

```
mov    eax, 0x4AAC90F7        ; eax = hash("GetProcAddress")
push   eax                    ; save eax/hash
push   esi                    ; esi = "CoInitializeEx"
push   ebx                    ; base of ole32
call   ebp                    ; invoke_api(eax)

; 1. _CoInitializeEx(NULL, COINIT_MULTITHREADED);
cdq                           ; edx = 0
push   edx                    ; COINIT_MULTITHREADED
push   edx                    ; NULL
call   eax                    ; CoInitializeEx

add    esi, co_init_len       ; skip "CoInitializeEx", 0

; _CoCreateInstance = GetProcAddress(ole32, "CoCreateInstance")
pop    eax                    ; eax = hash("GetProcAddress")
push   esi                    ; "CoCreateInstance"
push   ebx                    ; base of ole32
call   ebp                    ; invoke_api

add    esi, co_create_len     ; skip "CoCreateInstance", 0

; 2. _CoCreateInstance(
    ; &langId, 0, CLSCTX_INPROC_SERVER,
    ; &IID_IActiveScript, (void **)&engine);
push   edi                    ; &engine
scasd                         ; skip engine
mov    ebx, edi               ; ebx = &parser
push   edi                    ; &IID_IActiveScript
movsd
movsd
movsd
```

```
movsd
push    CLSCTX_INPROC_SERVER
push    0                       ;
push    esi                     ; &CLSID_VBScript or &CLSID_JScri
call    eax                     ; _CoCreateInstance

; 3. Query engine for script parser
; engine->lpVtbl->QueryInterface(
;  engine, &IID_IActiveScriptParse,
;  (void **)&parser);
push    edi                     ; &parser
push    ebx                     ; &IID_IActiveScriptParse32
inc     dword[ebx]              ; add 1 for IActiveScriptParse32
mov     esi, [ebx-4]            ; esi = engine
push    esi                     ; engine
mov     eax, [esi]              ; eax = engine->lpVtbl
call    dword[eax + IUnknownVtbl.QueryInterface]

; 4. Initialize parser
; parser->lpVtbl->InitNew(parser);
mov     ebx, [edi]              ; ebx = parser
push    ebx                     ; parser
mov     eax, [ebx]              ; eax = parser->lpVtbl
call    dword[eax + IActiveScriptParse32Vtbl.InitNew]

; 5. Initialize IActiveScriptSite
lea     eax, [ebp + (_QueryInterface - invoke_api)]
push    edi                     ; save pointer to IActiveScriptSi
stosd                           ; vft.QueryInterface      = (LPVO
add     eax, _AddRef  - _QueryInterface
stosd                           ; vft.AddRef              = (LPVO
stosd                           ; vft.Release             = (LPVO
```

```
add     eax, _GetLCID - _Release
stosd                           ; vft.GetLCID             = (LPVO
stosd                           ; vft.GetItemInfo         = (LPVO
stosd                           ; vft.GetDocVersionString = (LPVO
add     eax, _OnScriptTerminate - _GetDocVersionString
stosd                           ; vft.OnScriptTerminate   = (LPVO
add     eax, _OnStateChange - _OnScriptTerminate
stosd                           ; vft.OnStateChange       = (LPVO
stosd                           ; vft.OnScriptError       = (LPVO
inc     eax
inc     eax
stosd                           ; vft.OnEnterScript       = (LPVO
stosd                           ; vft.OnLeaveScript       = (LPVO
pop     eax                     ; eax = &vft

; 6. Set script site
; engine->lpVtbl->SetScriptSite(
;   engine, (IActiveScriptSite *)&mas);
push    edi                     ; &IMyActiveScriptSite
stosd                           ; IActiveScriptSite.lpVtbl = &vft
xor     eax, eax
stosd                           ; IActiveScriptSiteWindow.lpVtbl
push    esi                     ; engine
mov     eax, [esi]
call    dword[eax + IActiveScriptVtbl.SetScriptSite]

; 7. Parse our script
; parser->lpVtbl->ParseScriptText(
;     parser, cs, 0, 0, 0, 0, 0, 0, 0, 0);
mov     edx, esp
push    8
pop     ecx
```

```
init_parse:
        push    eax                     ; 0
        loop    init_parse
        push    edx                     ; script
        push    ebx                     ; parser
        mov     eax, [ebx]
        call    dword[eax + IActiveScriptParse32Vtbl.ParseScriptText]

        ; 8. Run script
        ; engine->lpVtbl->SetScriptState(
        ;     engine, SCRIPTSTATE_CONNECTED);
        push    SCRIPTSTATE_CONNECTED
        push    esi
        mov     eax, [esi]
        call    dword[eax + IActiveScriptVtbl.SetScriptState]

        ; 9. cleanup
        ; parser->lpVtbl->Release(parser);
        push    ebx
        mov     eax, [ebx]
        call    dword[eax + IUnknownVtbl.Release]

        ; engine->lpVtbl->Close(engine);
        push    esi                     ; engine
        push    esi                     ; engine
        lodsd                           ; eax = lpVtbl
        xchg    eax, edi
        call    dword[edi + IActiveScriptVtbl.Close]
        ; engine->lpVtbl->Release(engine);
        call    dword[edi + IUnknownVtbl.Release]

        inc     eax                     ; eax = 4096 * 32
```

```
    shl     eax, 17
    add     esp, eax
    popad
    ret
```

Windows Script Host Objects

Two named objects (WSH and WScript) are added to the script namespace by wscript.exe/cscript.exe that do not require instantiating at runtime. The 'WScript' object is used primarily for console I/O, accessing arguments and the path of script on disk. It can also be used to terminate a script via the Quit method or poll operations via the Sleep method. The IActiveScript interface only provides basic scripting functionality, so if we want our host to support those objects, or indeed any custom objects, they must be implemented manually. Consider the following code taken from ReVBShell that expects to run inside WSH.

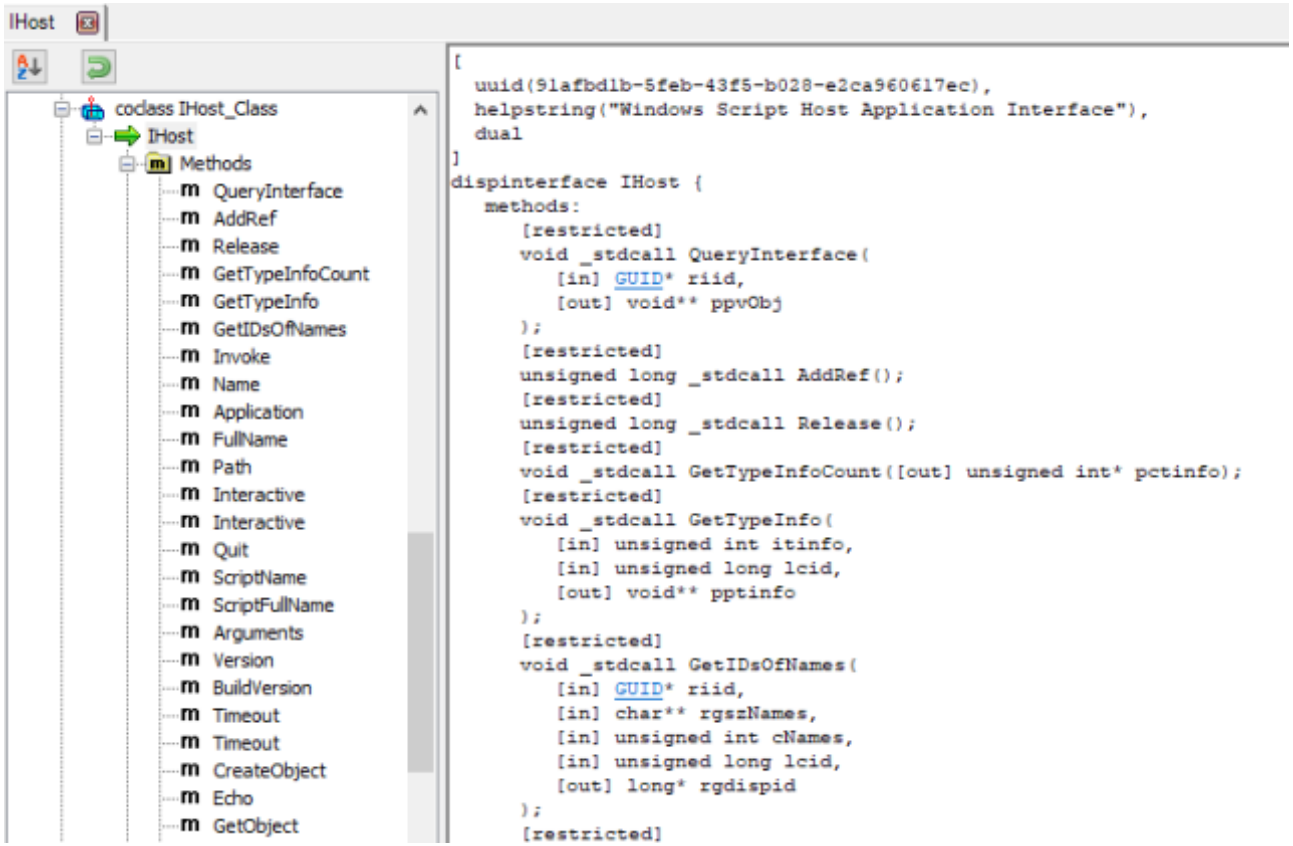
```
While True
  ' receive command from remote HTTP server
  ' other code omitted
  Select Case strCommand
    Case "KILL"
      SendStatusUpdate strRawCommand, "Goodbye!"
      WScript.Quit 0
  End Select
Wend
```

When this was used for testing Donut shellcode, the script engine stopped running upon reaching the line “WScript.Quit 0” because it didn’t recognize the WScript object. “On Error Resume Next” was enabled, and so the script simply kept executing. Once the name of this object was added to the namespace via IActiveScript::AddNamedItem, a request for ITypeInfo and IUnknown interfaces was made via IActiveScriptSite::GetItemInfo. If we don’t provide an interface for the request, the parser calls IActiveScriptSite::OnScriptError with the message “Variable is undefined ‘WScript'” before terminating.

To enable support for ‘WScript’ requires a custom implementation of the WScript interface defined in type information found in wscript.exe/cscript.exe. First, add the name of the object to the scripting engine’s namespace using AddNamedItem. This makes any methods, properties and events part of this object visible to the script.

```
obj = SysAllocString(L"WScript");
engine->lpVtbl->AddNamedItem(engine, (LPCOLESTR)obj, SCRIPTITEM_I
```

Obtain the type information from wscript.exe or cscript.exe. IID_IHost is simply the class identifier retrieved from aforementioned EXE files. Below is a screenshot of OleWoo, but other TLB viewers may work just as well.




```
ITypeLib  lpTypeLib;
ITypeInfo lpTypeInfo;

LoadTypeLib(L"WScript.exe", &lpTypeLib);
lpTypeLib->lpVtbl->GetTypeInfoOfGuid(lpTypeLib, &IID_IHost, &lpTy
```

Now, when the scripting engine first encounters the ‘WScript’ object and requests an IUnknown interface via [IActiveScriptSite::GetItemInfo](), Donut returns a pointer to a

minimal implementation of the IHost interface.

After this, the IDispatch::Invoke method will be used to call the 'Quit' method requested by the script. At the moment, Donut only implements Quit and Sleep methods, but others can be supported if requested.

Extensible Stylesheet Language Transformations (XSLT)

XSL files can contain interpreted languages like JScript/VBScript. The following code found here is based on this example by TheWover.

```
void run_xml_script(const char *path) {
    IXMLDOMDocument *pDoc;
    IXMLDOMNode     *pNode;
    HRESULT         hr;
    PWCHAR          xml_str;
    VARIANT_BOOL    loaded;
    BSTR            res;

    xml_str = read_script(path);

    if(xml_str == NULL) return;

    // 1. Initialize COM
    hr = CoInitialize(NULL);
    if(hr == S_OK) {
      // 2. Instantiate XMLDOMDocument object
      hr = CoCreateInstance(
        &CLSID_DOMDocument30,
        NULL, CLSCTX_INPROC_SERVER,
        &IID_IXMLDOMDocument,
        (void**)&pDoc);
```

```
        if(hr == S_OK) {
          // 3. load XML file
          hr = pDoc->lpVtbl->loadXML(pDoc, xml_str, &loaded);
          if(hr == S_OK) {
            // 4. create node interface
            hr = pDoc->lpVtbl->QueryInterface(
              pDoc, &IID_IXMLDOMNode, (void **)&pNode);

            if(hr == S_OK) {
              // 5. execute script
              hr = pDoc->lpVtbl->transformNode(pDoc, pNode, &res);
              pNode->lpVtbl->Release(pNode);
            }
          }
          pDoc->lpVtbl->Release(pDoc);
        }
        CoUninitialize();
      }
      free(xml_str);
  }
```

PC-Relative Addressing in C

The linker makes an assumption about where a PE file will be loaded in memory. Most EXE files request an image base address of 0x00400000 for 32-bit or 0x0000000140000000 for 64-bit. If the PE loader can't map at the requested address, it uses relocation information to fix position-dependent code and data. ARM has support for PC-relative addressing via the ADR, ADRP and LDR opcodes, but poor old x86 lacks a similar instruction. x64 does support RIP-relative addressing, but there's no guarantee a compiler will use it even if we tell it to (-fPIC and -fPIE for GCC). Because we're using C for the shellcode, we need to manually calculate the address of a function relative to where the

shellcode resides in memory. We could apply relocations in the same way a PE loader does, but self-modifying code can trigger some anti-malware programs. Instead, the program counter (EIP on x86 or RIP on x64) is read using some assembly and this is used to calculate the virtual address of a function in-memory. The following code stub is placed at the end of the payload and returns the value of the program counter.

```
#if defined(_MSC_VER)
  #if defined(_M_X64)

    #define PC_CODE_SIZE 9 // sub rsp, 40 / call get_pc

    static char *get_pc_stub(void) {
      return (char*)_ReturnAddress() - PC_CODE_SIZE;
    }

    static char *get_pc(void) {
      return get_pc_stub();
    }

  #elif defined(_M_IX86)
    __declspec(naked) static char *get_pc(void) {
      __asm {
          call   pc_addr
        pc_addr:
          pop    eax
          sub    eax, 5
          ret
      }
    }
  #endif
```

```
#elif defined(__GNUC__)
  #if defined(__x86_64__)
    static char *get_pc(void) {
        __asm__ (
        "call   pc_addr\n"
      "pc_addr:\n"
        "pop    %rax\n"
        "sub    $5, %rax\n"
        "ret");
    }
  #elif defined(__i386__)
    static char *get_pc(void) {
        __asm__ (
        "call   pc_addr\n"
      "pc_addr:\n"
        "popl   %eax\n"
        "subl   $5, %eax\n"
        "ret");
    }
  #endif
#endif
```

With this code, the linker will calculate the Relative Virtual Address (RVA) by subtracting the offset of our target function from the offset of the get_pc() function. Then at runtime, it will subtract the RVA from the program counter returned by get_pc() to obtain the Virtual Address of the target function. The position of get_pc() must be placed at the end of a payload, otherwise this would not work. The following macro (named after the ARM opcode ADR) is used to calculate the virtual address of a function in-memory.

```
#define ADR(type, addr) (type)(get_pc() - ((ULONG_PTR)&get_pc -
```

To illustrate how it's used, the following code from the payload shows how to initialize the IActiveScriptSite interface.

```
// initialize virtual function table
static VOID ActiveScript_New(PDONUT_INSTANCE inst, IActiveScriptS
    MyIActiveScriptSite *mas = (MyIActiveScriptSite*)this;

    // Initialize IUnknown
    mas->site.lpVtbl->QueryInterface      = ADR(LPVOID, ActiveScr
    mas->site.lpVtbl->AddRef              = ADR(LPVOID, ActiveScr
    mas->site.lpVtbl->Release             = ADR(LPVOID, ActiveScr

    // Initialize IActiveScriptSite
    mas->site.lpVtbl->GetLCID             = ADR(LPVOID, ActiveScr
    mas->site.lpVtbl->GetItemInfo         = ADR(LPVOID, ActiveScr
    mas->site.lpVtbl->GetDocVersionString = ADR(LPVOID, ActiveScr
    mas->site.lpVtbl->OnScriptTerminate   = ADR(LPVOID, ActiveScr
    mas->site.lpVtbl->OnStateChange       = ADR(LPVOID, ActiveScr
    mas->site.lpVtbl->OnScriptError       = ADR(LPVOID, ActiveScr
    mas->site.lpVtbl->OnEnterScript       = ADR(LPVOID, ActiveScr
    mas->site.lpVtbl->OnLeaveScript       = ADR(LPVOID, ActiveScr

    mas->site.m_cRef                      = 0;
    mas->inst                             = inst;
}
```

Dynamic Calls to DLL Functions

After implementing support for some WScript methods, providing access to DLL functions directly from VBScript/JScript using a similar approach is much easier to understand. The initial problem is how to load type information directly from memory. One solution to this can be found in A lightweight approach for exposing C++ objects to a hosted Active Scripting engine. Confronted with the same problem, the author uses CreateDispTypeInfo and CreateStdDispatch to create the ITypeInfo and IDispatch interfaces necessary for interpreted languages to call C++ objects. The same approach can be used to call DLL functions and doesn't require COM registration.

Summary

v0.9.2 of Donut will support in-memory execution of JScript/VBScript and XSL files. Dynamic calls to DLL functions without COM registration will be supported in a future release.

Related

Shellcode: Loading .NET Assemblies From Memory
In "assembly"

Shellcode: A reverse shell for Linux in C with support for TLS/SSL
In "assembly"

MiniDumpWriteDump via COM+ Services DLL
In "windows"

This entry was posted in assembly, programming, security, shellcode, windows and tagged assembly, javascript, jscript, perl, python, shellcode, vbscript, x86. Bookmark the permalink.

← Shellcode: In-Memory Execution of DLL

Windows Process Injection: Winsock Helper Functions (WSHX) →

Leave a Reply

Enter your comment here...