Poweliks - Command Line Confusion - Stormshield

By Stormshield Customer Security Lab

Published: 2014-08-20 · Archived: 2026-04-05 14:25:29 UTC

Recently, hFireF0X provided a detailed walkthrough on the reverse engineering forum kernelmode.info about Win32/Poweliks malware. The particularity of this malware is that it resides in the Windows registry and uses rundll32.exe to execute JavaScript code.

I found it funny that we can execute some JavaScript through Rundll32 and obviously I was not the only one.

When we first saw the command line executing JavaScript, we were wondering how it worked.

In this blog post, we analyze how and why JavaScript is executed when calling this simple command line:

Reminder about Rundll32

Rundll32 usage is documented on MSDN; it is used to call an exported function of a DLL file which can be achieved with the following command line:

```
 RUNDLL32.EXE <dllname>,<entrypoint> <optional arguments>

entrypoint is the exported function; its prototype must be:

void CALLBACK EntryPoint(HWND hwnd, HINSTANCE hinst, LPSTR lpszCmdLine, int nCmdShow);
```

The lpszCmdLine parameter is given the <optional arguments> value specified on the rundll32 command line.

We will try to figure out how Rundll32 is able to call the function `RunHTMLApplication` exported by the library mshtml.dll and how the `"javascript:"` prefix is used to execute actual JavaScript code.

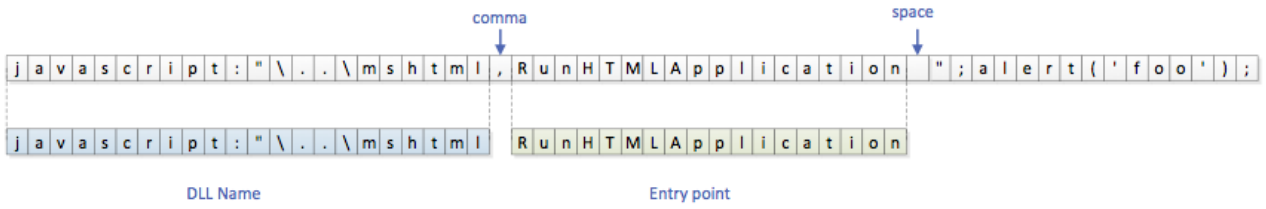
Analysis of Rundll32

Parameters

One of the first things done by Rundll32 is to parse the command line in the internal function `ParseCommand`. This function searches for a comma (‘,’, 0x2C) to locate the DLL name and for a space (‘ ‘, 0x20) to locate the entrypoint name.

```
01001656  >  66:83F9 20    CMP CX,20
0100165A  .v 74 06         JE SHORT 01001662
0100165C  .  66:83F9 2C    CMP CX,2C
01001660  .v 75 0A         JNE SHORT 0100166C
01001662  >  03C3          ADD EAX,EBX
01001664  .  66:8B08       MOV CX,WORD PTR DS:[EAX]
01001667  .  66:3BCF       CMP CX,DI
0100166A  .^ 75 EA         JNE SHORT 01001656
```

When using our sample command line, `ParseCommand` returns `javascript:"\..\mshtml` as the DLL name and `RunHTMLApplication` as the entrypoint. In this context, the space after `RunHTMLApplication` delimits the 'optional arguments' part of the rundll32 command line:



Dll loader

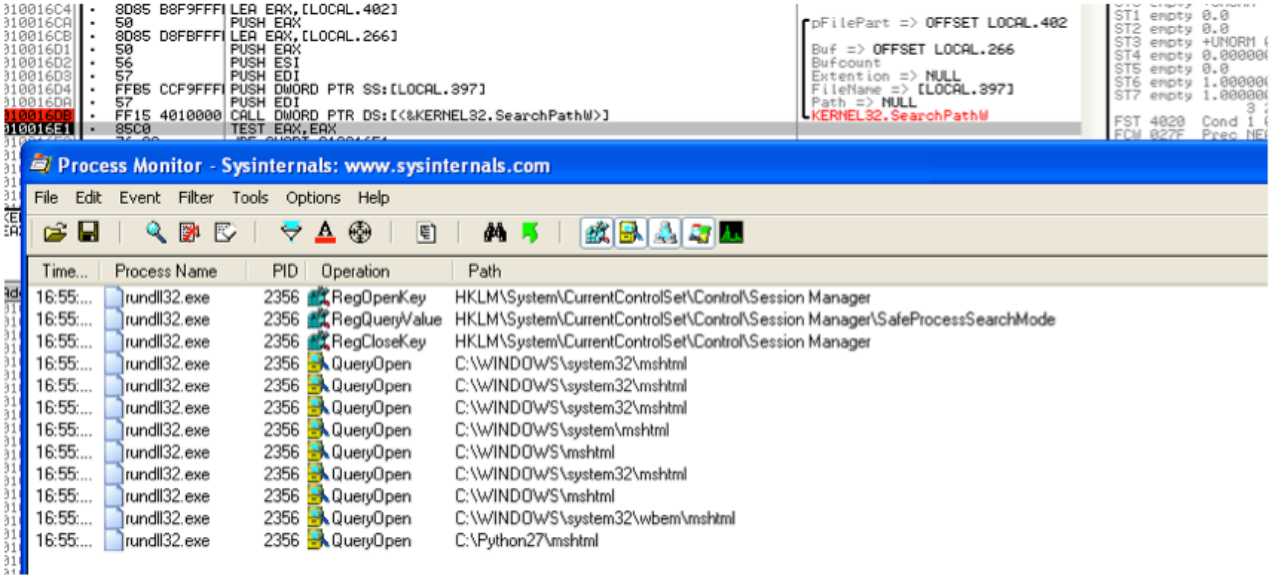
Rundll32 will perform several tries to load the actual DLL from the initial specification `javascript:"\..\mshtml`.

The first test uses the function `GetFileAttributes("javascript:"\..\mshtml")`. This function eventually accesses `C:\Windows\system32\mshtml`. As this file is not found, the function returns -1.

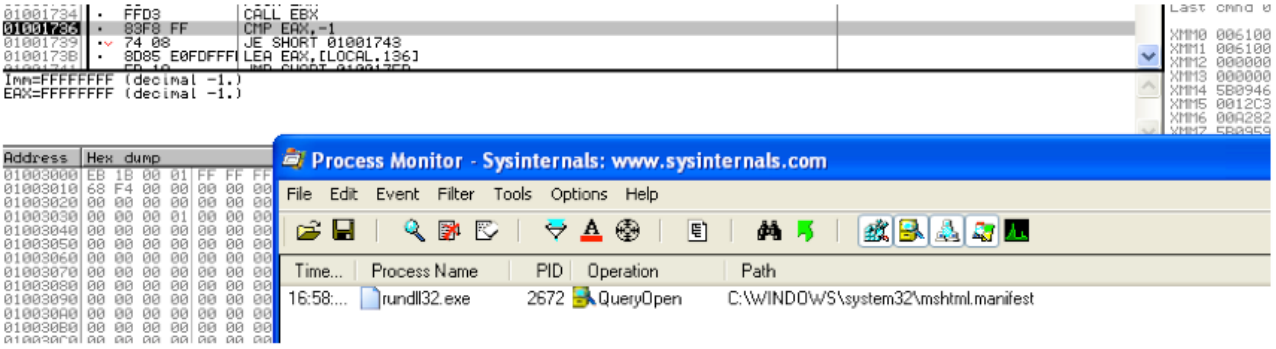
`SearchPath` is then invoked to resolve the DLL name. This function reads the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\SafeProcessSearchMode`. The Microsoft definition of this key is:

When the value of this REG_DWORD registry value is set to 1, SearchPath first searches the folders that are specified in the system path, and then searches the current working folder. When the value of this registry value is set to 0, the computer first searches the current working folder, and then searches the folders that are specified in the system path. The system default value for this registry key is 0.

By default this registry key doesn't exist (on Windows XP / 7 / 8) so `SearchPath` tries to load the file `mshtml` in the current directory of rundll32 (c:\windows\system32) prior to trying locating it in the system path.



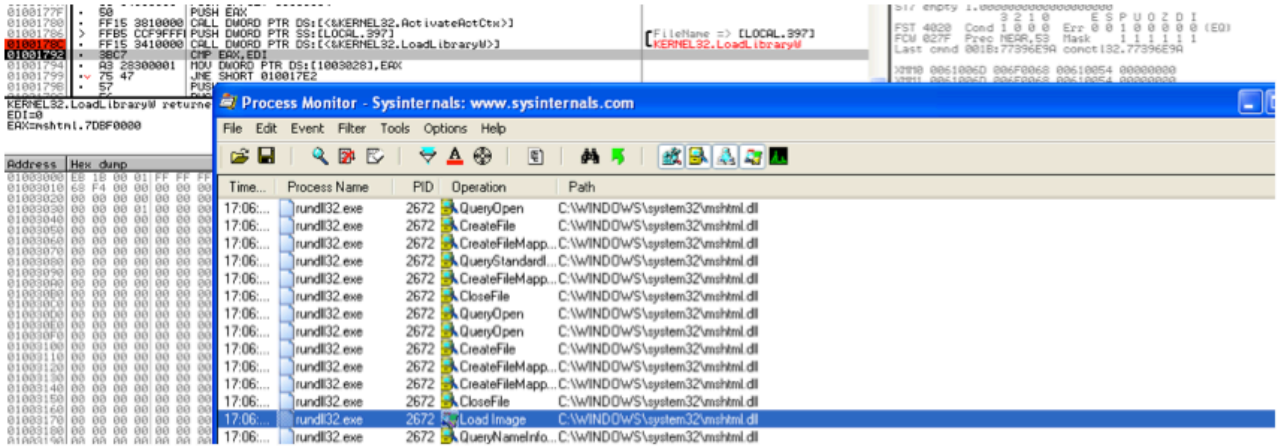
All these attempts fail and rundll32 moves to the next step. `GetFileAttributes` is called again searching for the manifest for the module: `javascript:"\..\mshtml.manifest`



Since all the previous steps failed, Rundll32 eventually calls `LoadLibrary("javascript:"\..\mshtml")`.

`LoadLibrary` is just a thin wrapper around `LdrLoadDll` located in `ntdll.dll`. Internally, `LdrLoadDll` adds the default extension .dll and parses the resulting string `javascript:"\..\mshtml.dll` as a path. The token `..` instructs to go one folder up: it resolves to `mshtml.dll` (think of `foo\..\mshtml.dll` resolved as `mshtml.dll`).

With `mshtml.dll` specification, `LdrLoadDll` is able to load the library in the system directory.



Rundll32 then calls `GetProcAddress` with the previously extracted entry point name `RunHTMLApplication`.

For the moment, the `javascript:` prefix seems pretty useless: `LoadLibrary("foobar:\"\..\mshtml")` works fine. So, why prefixing with `javascript:` ?

Protocols Handler

Once the entry point address has been resolved, Rundll32 calls the function `mshtml.dll!RunHTMLApplication`.

Even if not documented, the actual `RunHTMLApplication` can be inferred from the call made by c:\windows\system32\mshta.exe (the application dedicated to launch an `.hta` file):

```
 HRESULT RunHTMLApplication(
HINSTANCE hinst,
HINSTANCE hPrevInst,
LPSTR szCmdLine,
int nCmdShow
);
```

This is not far from the function prototype expected for a rundll32 entry point:

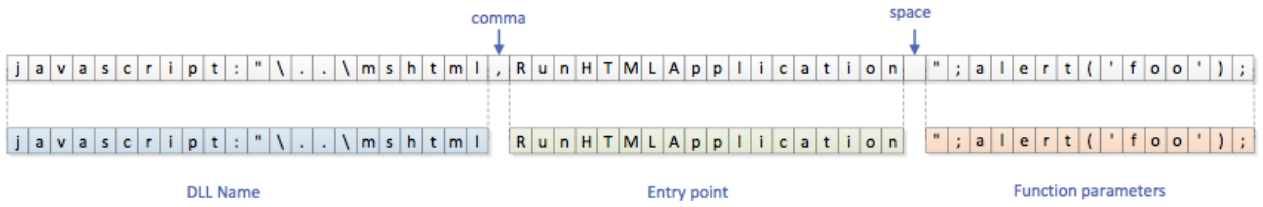
```
 void CALLBACK EntryPoint(
HWND hwnd,
HINSTANCE hinst,
LPSTR lpszCmdLine,
int nCmdShow
);
```

`RunHTMLApplication` receives a handle to a window instead of a handle to a module as the first parameter. This parameter is used when mshml registers for a window class and creates a window of this new class. Passing a value not corresponding to an actual instance doesn't seem to disturb user32 very much…

The second parameter is not used at all, so the mismatch is not important.

The last parameter, `nCmdShow`, is used by the `RunHTMLApplication` function to display the window hosting the HTML application. Rundll32 always calls the entry point function with the value `SW_SHOWDEFAULT` to instruct any potential opened window to use window default placement.

The main parameter of interest would be `lpszCmdLine` (`";alert('foo')`) in our case.



This obviously leads to an issue since this is not a valid JavaScript statement (please note the missing double-quote at the end of the statement). But it works anyway, because `RunHTMLApplication` ignores the given parameter and prefers to request again the original command line from the `GetCommandLine` Windows API (wrapped in a call to the `GetCmdLine` function).

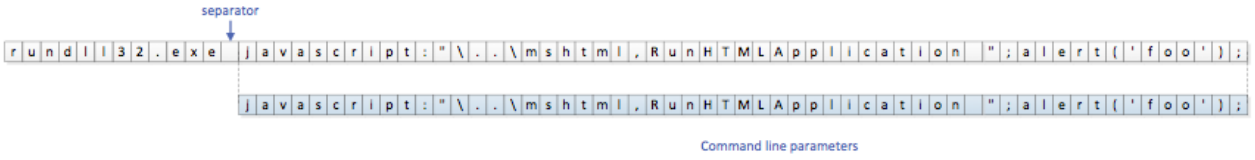
```
_RunHTMLApplication@16 proc near        ; DATA XREF: .text:off_7DCD6EC4↑o

ppmk            = dword ptr  8
hPrevInst       = dword ptr  0Ch
szCmdLine       = dword ptr  10h
nCmdShow        = dword ptr  14h

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    esi
                call    ?GetCmdLine@@YGPAGXZ ; GetCmdLine(void)
                push    [ebp+nCmdShow]  ; int
                mov     esi, offset ?theApp@@3VCHTMLApp@@A ; CHTMLApp theApp
                push    eax             ; Source
                push    [ebp+ppmk]      ; ppmk
                mov     ecx, esi
                call    ?Init@CHTMLApp@@QAEJPAUHINSTANCE__@@PAGH@Z ; CHTMLApp::Init(HINSTANCE__ *,ushort *,int)
                test    eax, eax
                jnz     short loc_7DDA0B92
                mov     ecx, esi
                call    ?Run@CHTMLApp@@QAEXXZ ; CHTMLApp::Run(void)

loc_7DDA0B92:                           ; CODE XREF: RunHTMLApplication(x,x,x,x)+20↑j
                mov     ecx, esi
                call    ?Terminate@CHTMLApp@@QAEJXZ ; CHTMLApp::Terminate(void)
                pop     esi
                pop     ebp
                retn    10h
_RunHTMLApplication@16 endp
```

The full command line contains the name of the executable and the parameters: `GetCmdLine` extracts the parameters by cleaning up the executable specification:



After that, `RunHTMLApplication` calls `CreateUrlMoniker`:

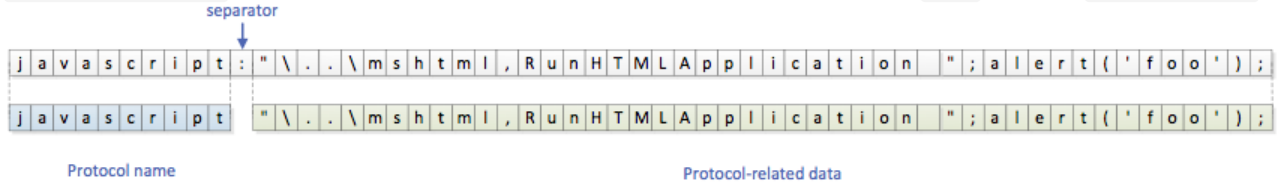
```
__stdcall CHTMLApp__CreateHTAMoniker(LPMONIKER *ppmk)
=HTAMoniker@CHTMLApp@@QAEJPAPAUIMoniker@@@Z proc near
                                ; CODE XREF: CHTMLApp::Init(HINSTANCE__ *,ushort *,int)+D8↓p

= dword ptr -4
= dword ptr  8

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
and     [ebp+szURL], 0
add     ecx, 9Ch
push    esi
push    ecx
lea     ecx, [ebp+szURL]
call    ?Set@CStr@@QAEJABV1@@Z ; CStr::Set(CStr const &)
push    [ebp+szURL]     ; pszPath
call    ds:__imp__PathRemoveArgsW@4 ; PathRemoveArgsW(x)
push    [ebp+szURL]     ; lpsz
call    ds:__imp__PathUnquoteSpacesW@4 ; PathUnquoteSpacesW(x)
push    1               ; dwFlags = 1
push    [ebp+ppmk]      ; ppmk
push    [ebp+szURL]     ; szURL = javascript:"\..\mshtml,RunHTMLApplication ";alert('foo');
push    0               ; pMkCtx = 0
call    _CreateURLMonikerEx@16 ; CreateURLMonikerEx(x,x,x,x)
lea     ecx, [ebp+szURL]
mov     esi, eax
call    ?_Free@CStr@@AAEXXZ ; CStr::_Free(void)
mov     eax, esi
pop     esi
leave
retn    4
```

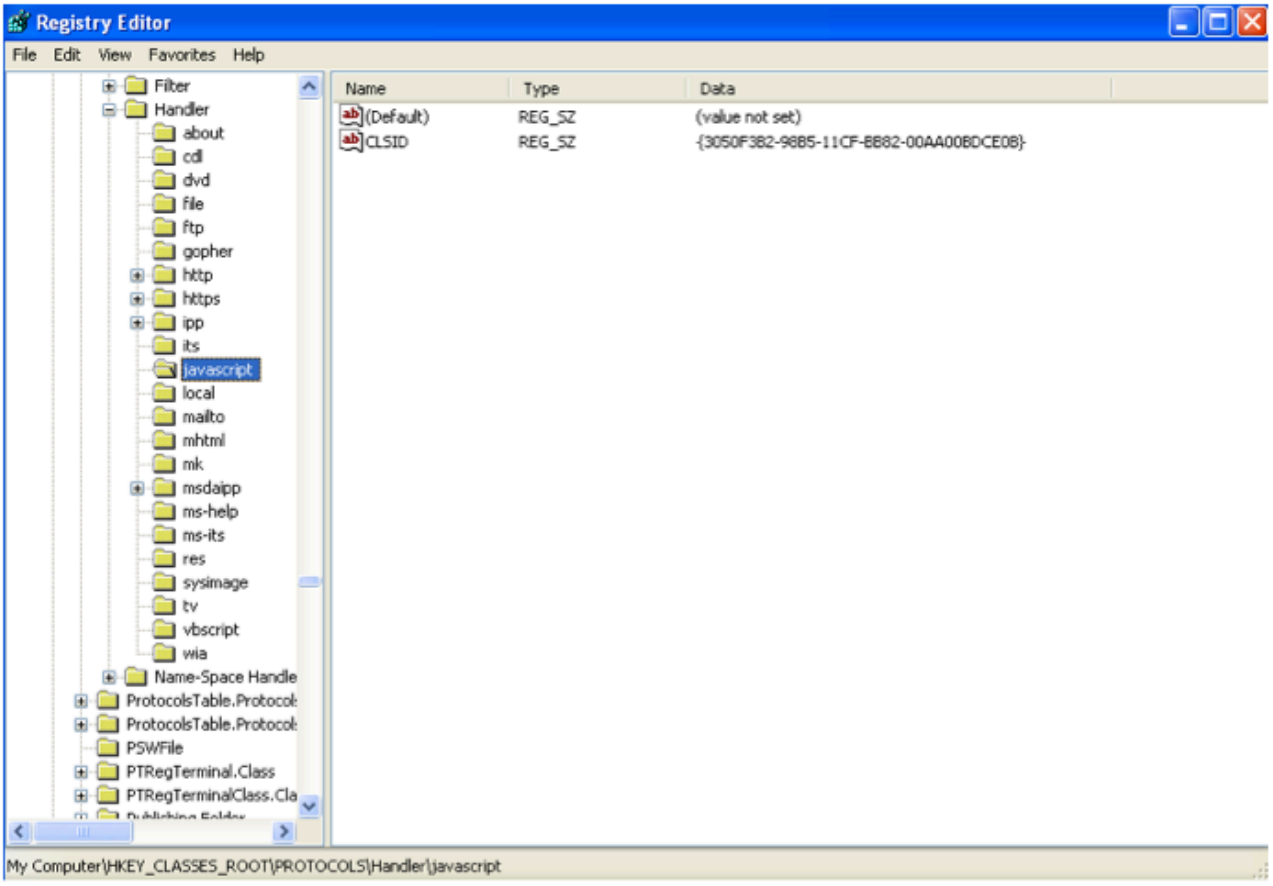
This is where the string `« javascript: »` is essential.

`CreateUrlMoniker` parses the command line to extract the string before the char `":"` (0x3A): `"javascript"`.

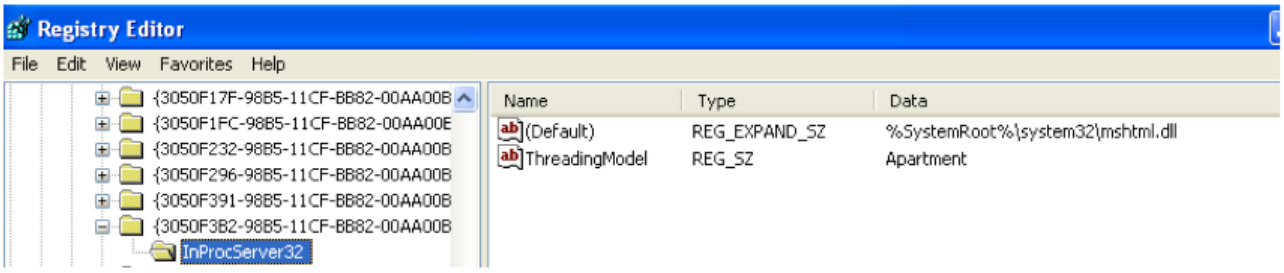


`CreateUrlMoniker` crawls the registry key `HKCR\SOFTWARE\Classes\PROTOCOLS\Handler\`. These keys refer to a set of protocols and their CLSID.

`CreateUrlMoniker` finds an appropriate protocol handler for the JavaScript protocol (`HKCR\SOFTWARE\Classes\PROTOCOLS\Handler\javascript`):

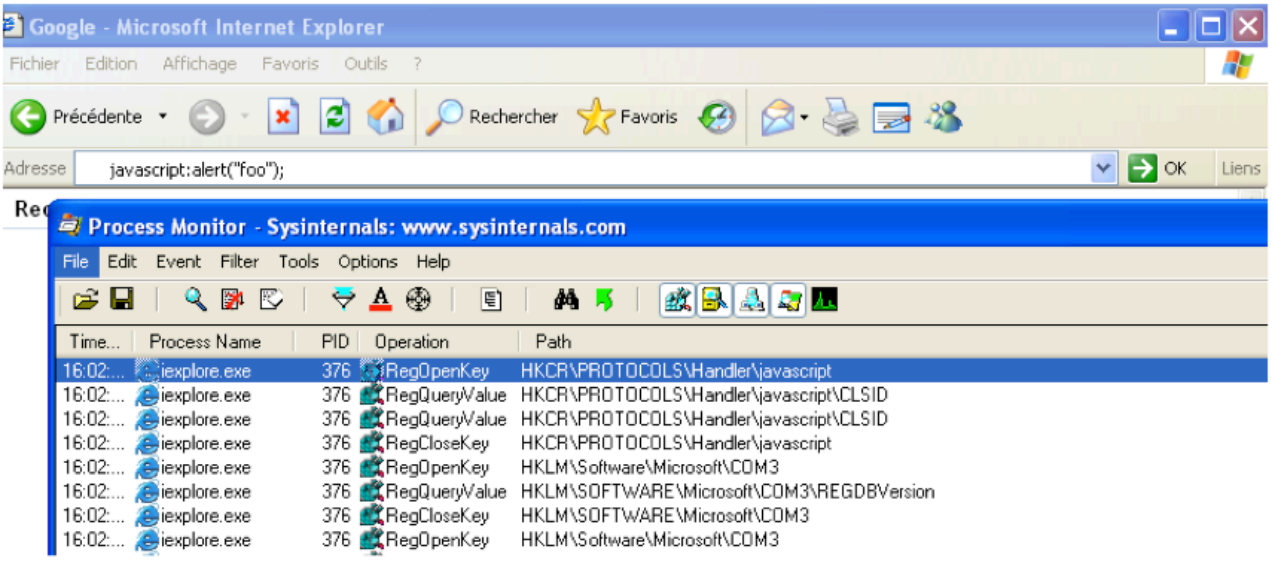


The CLSID `{3050F3B2-98B5-11CF-BB82-00AA00BDCE0B}` matches « `Microsoft HTML Javascript Pluggable Protocol` ».



It is for this reason that the string " `javascript` " is essential in the beginning of the parameters.

The same mechanism comes into play when one types `javascript:alert('foo')` ; in the Internet Explorer navigation bar:

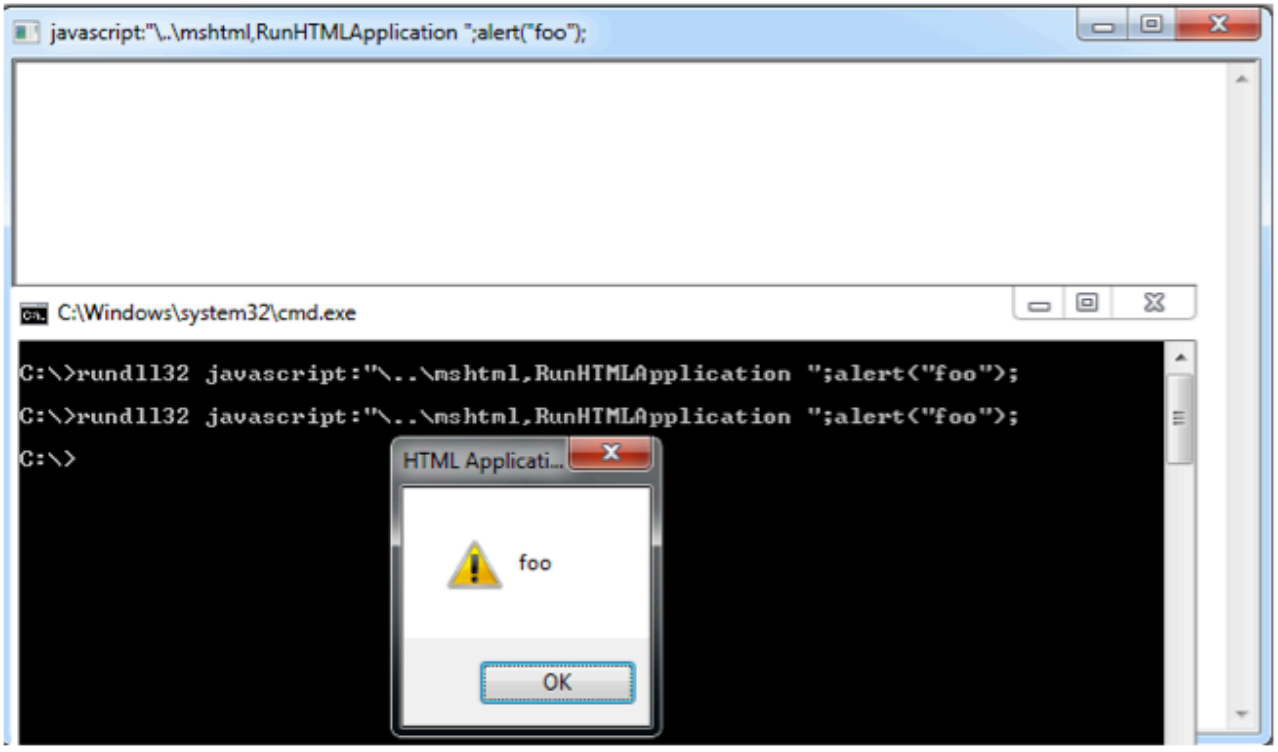


The remaining of the string located after the ':' separator is interpreted by the JavaScript URL moniker as JavaScript instructions:

```
"\..\mshtml,RunHTMLApplication ";alert('foo');
```

This is a valid JavaScript with a string `"\..\mshtml,RunHTMLApplication "` (hence the double-quotes skipped in all the previous steps!) and a function (`alert`).

Finally `RunHTMLApplication` calls `CHTMLApp::Run` and the JavaScript is executed:



Security point

From a security point of view, executing JavaScript through Rundll32 is like executing an HTML Application.

In other words, we can have all the power of Internet Explorer—its object model, performance, rendering power and protocol support—without enforcing the strict security model and user interface of the browser. Zone security is off, and cross-domain script access is allowed, we have read/write access to the files and system registry on the client machine.

With this trick, JavaScript is executed outside the Internet Explorer process and script is not subject to security concept like Protected Mode / Sandbox on Vista and superior.

Conclusion

`RunHTMLApplication` has the perfect prototype to work with Rundll32. Attackers have made great efforts to build a command line using the perfect syntax for passing through all the mechanisms (library loading, command line parsing, URL syntax correctness, valid JavaScript, etc.) leading to JavaScript execution in an uncontrolled environment.

From our understanding, this technique allows bypassing some security products that may trust actions performed by the built-in rundll32 while specifying the script to run without writing any file on the file system.

That’s all folks!
