FEATURE 1

INSIDE THE MICROSOFT SCRIPT ENCODER

Peter Ferrie
Symantec Security Response, USA

When the *Microsoft Script Encoder* was released in 1999, it was predicted that malware authors would use it to obfuscate their code. As a result, tools claiming to be able to decode the files produced by the script encoder started to appear almost instantly.

YOU BREAK IT, YOU BUY IT

Recently, I was given an encoded script to examine, and I was told that it might contain an exploit of some kind. Since I am not in the habit of carrying script decoding tools with me, I downloaded a script decoder tool and used it to decode the file.

The result was a partially decoded file containing a fragment of what looked like shellcode and a lot of garbage bytes. Since the file was in ANSI format, there were three possibilities: that the file contained double-byte character set (DBCS) characters which were not being decoded correctly on the English system; that the script was broken; or that the tool contained a bug.

Assuming the first case, I tried decoding the file using other language formats that the tool supported, but again I was unsuccessful.

There are four languages that use DBCS characters: traditional Chinese, simplified Chinese, Japanese and Korean. Since the tool that I had downloaded supported only three of these languages, I decided to download several other script decoder tools in the hope that one of them would support the remaining language.

When the half-dozen or so tools that I had downloaded also failed to decode the script, I ruled out the third possibility. How likely is it that every copy of a tool would have the same bug? (As a matter of fact this is more likely than one might imagine, as I found out afterwards via a completely unrelated matter: try searching for tools that enable 'Unreal' mode on x86 and x86-64 processors, notice how many people claim to have found it, and notice that none of them enable the A20 line first.)

That left the second possibility – that the script was broken.

SEE SCRIPT RUN, RUN SCRIPT RUN

Although I knew that running the script wouldn't provide a conclusive result, I ran it anyway. Sure enough, the Windows Scripting Host reported that the script was invalid. The question was: why?

Since I had long forgotten the details of encoded scripts, I downloaded the *Microsoft Script Encoder* tool (screnc.exe) and started to reverse-engineer it. Under normal circumstances, one would assume that examination of an encoder would be sufficient to provide an understanding of decoding methods. However, in this instance that is not the case, and it seems that the creators of the decoding tools all made the same mistake.

SCRENC.EXE

The first interesting thing I noticed about the *Microsoft Script Encoder* is that it supports ANSI, UTF-8, Little-Endian and Big-Endian Unicode input file formats. This is interesting because neither *Microsoft*'s own VBScript and JScript scripting engines (before encoding), nor the decoder built into those scripting engines (after encoding), support anything but ANSI and Little-Endian Unicode. Upon attempting to execute files in the other formats, the *Windows Scripting Host* reports that they are invalid.

The second interesting thing is that the encoding is done by the VBScript and JScript scripting engines themselves. The reason for this is that these engines support dynamic encryption, using the EncodeScriptFile method. It seems that this method was not noticed by malware authors.

The rest was fairly straightforward: encoded files begin with the signature '#@~^', followed by the base64-encoded length of the script that follows immediately. After the script is the base64-encoded checksum, and the signature '^#~@'.

The checksum is simply the sum of all of the characters from the script before it was encoded. It is used during the decoding phase to verify that the script has been decoded correctly, rather than to verify that the encoded script has not been altered.

If the script is not already in the Unicode format, it is converted to the Unicode format in memory prior to encoding. If the original file was in the ANSI format, the current code page is used to perform the translation, which causes the DBCS problem described above.

Once in the Unicode format, characters are not encoded if more than seven bits are required to identify them. In those cases, the character is simply copied instead. However, if the resulting encoded script is then saved in the ANSI format, any characters that cannot be represented in seven bits or fewer will be replaced by the system default for untranslatable characters (which is usually the '?'

character). If such a replacement is made, the script can no longer be decoded properly.

COMPOUND INTEREST

Now the fun begins – watching how *Microsoft*'s VBScript and JScript scripting engines deal with encoded files. Immediately we see that what screnc.exe produces is not all that those engines will accept.

The first thing the script decoder does is to search the entire script for the signature '#@~^'. This means that the script is not required to appear at the start of the file. Screnc.exe can produce such files only when the script is inside an HTML file. However, all of the decoding tools that I tried supported this behaviour.

The encoded script is decoded into the same location in the script at which it is found (but not the same location in memory). This means that unencoded script can appear before and/or after encoded script, even though screnc.exe cannot produce such files. None of the tools that I tried were affected by this, since they all found the script no matter where it was. However, one of the tools did not append the unencoded script that appeared after the encoded script.

The entire script is searched for all signatures that exist. This means that multiple encrypted scripts can appear in a single file! Screnc.exe cannot produce such files, and none of the tools that I tried supported it, either. But wait – it gets worse ...

WHAT THE #@~^?!

Any part of a script can be encoded, even down to the level of individual characters. The result is that a script such as:

```
oh_this="bad"
```

can become (unencoded characters are marked in bold):

oh_th#@~^AQAAAA==raQAAAA==^#~@**s="b**#@~^AQAAAA==CYQAAAA==^#~@**d"**

Fortunately, recursive encryption is not allowed, since the script decoder makes only a single pass over the script and decodes it to a different location in memory. If the script decoder had decoded to the same memory location, then it might have been possible to support recursive encoding to arbitrary levels, which would have made the problem much worse.

Given that an encoded script can appear anywhere in a file, it might seem surprising that all of the authors of the decoder tools made the same assumption: that the '#@~^' signature is a guarantee that what follows is an encoded script.

Of course, that's simply not true. Thus, the line:

```
x="#@~^" :#@~^AwAAAA==a{F5gAAAA==^#~@:msgbox(x)
```

was not decoded by any of those tools, yet *Microsoft*'s VBScript and JScript scripting engines decoded and executed it correctly (it prints '1', not '#@~^'). Also note the space that appears after the ' " '. Without it, even *Microsoft*'s VBScript and JScript scripting engines are fooled into believing that what follows the first '#@~^' signature is an encoded script, since the ':' character is a valid entry in the base64 dictionary that is used to decode the script length. The decoded length is an enormous value, and too large for the *Windows Scripting Host*, which reports that the script is invalid and exits without executing any further script.

Despite the fact that unencoded script can appear both before and after encoded script, the decoding is done before any script is interpreted, so string concatenation does not work. For example,

```
a="#@~^AQAAAA=="+"qMQAAAA==^#~@"
```

does not decode to 'a=1'. Additionally, decoding is not done after any script is interpreted, so this line:

```
eval(x)
```

where x is the encoded script

```
'#@~^AQAAAA==CYQAAAA==^#~@'
```

that was read from a file, is not decoded and does not evaluate to 'a'.

(THAT'S A BUG)

As mentioned above, the length and checksum of the decoded script are stored in base64 format. A bug exists in the base64 decoder in *Microsoft*'s VBScript and JScript scripting engines, which does not limit the input values correctly. This can be used to obfuscate the true length and/or checksum from tools which accept only files whose length and checksum are correct.

There is also an integer overflow bug in *Microsoft*'s VBScript and JScript scripting engines that causes a crash while calculating the length of the script. The bug is triggered if the top bit is set in a decoded length that would otherwise point within the file.

CONCLUSION

So what happened to that script? In the end, it was simply broken. There were extra characters inserted throughout the script, so the decoded length did not match; and there were some characters whose value was incorrect ('.' instead of the tab character, for example), so the decoded checksum did not match. After I had identified and removed the extra characters, and corrected the incorrect characters, the script decoded properly on an English system. It even contained an exploit, but it was one that we knew about already.