MALWARE ANALYSIS

$$$_+$$+$$__+_$+$$_$+$$$_+$$_$

Peter Ferrie
Microsoft, USA

Imagine a JavaScript encoding method that produces files that contain no alphanumeric characters, only symbols such as '$', '_', and '+'. It would be difficult to imagine how it could possibly work, but unfortunately one such encoder exists. It is called 'JJEncode'. A demonstration version is freely available from the author's website, and has already been used in malware. This article provides a detailed description of how it works.

_$+"\\"+__$+$_$+$_$+"\\"+__$+$__+$$$

We start with this:

```
$=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_
$_:(![]+"")[$],_$_:++$,$_$$:({}+"")[$],$$_
$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],$__:++$,$_
$:++$,$$__:({}+"")[$],$$_:++$,$$$:++$,$___:++$,$_
_$:++$};$.$_=($.$_=$+"")[$.$_$]+($._$=$.$_[$.__
$])+($.$$=($.$+"")[$.__$])+((!$)+"")[$._$$]+($.__
=$.$_[$.$$_])+($.$=(!""+"")[$.__$])+($._=(!""+"")[$._
$_])+$.$_[$.$_$]+$.__+$._$+$.$;$.$$=$.$+(!""+"")
[$._$$]+$.__+$._+$.$+$.$$;$.$=($.___)[$.$_][$.$_
];$.$($.$($.$$+"\""+ENCODED+"\"")())();
```

Note that the 'ENCODED' above does not appear in encoded files, rather it is the location where the encoded host code would appear. Also note that this algorithm does not work in direct mode (that is, putting it in a .js won't work) because it requires a feature that was introduced in HTML 4.0. As a result, it must appear in an HTML page, and that HTML page must declare its need for HTML 4.0 or later using a declaration like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0//EN">
```

The 'HTML 4.0' string can be replaced by later versions, such as 'HTML 4.1' or 'XHTML 1.0', etc.

On to the code...

$_

```
$=~[]
```

The expression '[]' returns a reference to an empty array. The operator '~' accesses the value at that reference ('0' in this case), and then inverts that value, resulting in the value '-1'. This value is assigned to the variable '$'.

```
$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_
:(![]+"")[$],_$_:++$,$_$$:({}+"")[$],$$_
$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],$__:++$,$_
$:++$,$$__:({}+"")[$],$$_:++$,$$$:++$,$___:++$,$__
$:++$}
```

This line can also be written as follows:

```
$=
        {
                ___:++$,
                $$$$:(![]+"")[$],
                __$:++$,
                $_$_:(![]+"")[$],
                _$_:++$,
                $_$$:({}+"")[$],
                $$_$:($[$]+"")[$],
                _$$:++$,
                $$$_:(!""+"")[$],
                $__:++$,
                $_$:++$,
                $$__:({}+"")[$],
                $$_:++$,
                $$$:++$,
                $___:++$,
                $__$:++$
        }
```

The '{' and '}' signify the creation of an object, and each line between the braces creates a property and assigns it a value during the object construction. We'll examine the lines one at a time.

```
___:++$
```

The expression '++$' sets the value in the variable '$' to '0' (specifically, it is incremented by 1, from '-1' to '0'). The ':' assigns a value to a property, and the property name is '___', so the property '___' is set to '0'.

```
$$$$:(![]+"")[$],
```

The expression '[]' returns a reference to an empty array, as above. The operator '!' tests if the reference is zero, which it is not, resulting in the Boolean value 'false'. The expression '+""' causes the Boolean value to be converted to a string, after which the empty string is appended to it. The result is the string 'false'. The expression '[$]' causes the string to be converted to an array[1], and a single character to be returned. The variable '$' has the value '0' from above, so the first character of the string 'false' ('f') is returned. That value is assigned to the property '$$$$'.

```
__$:++$,
```

The expression '++$' sets to '1' the value of the variable '$', and assigns that value to the property '__$'.

```
$_$_:(![]+"")[$],
```

The second character of the string 'false' ('a') is assigned to the property '$_$_'. The use of the expression '![]'

[1] This is HTML-specific behaviour. Normally, a string cannot be converted to an array.

appears to be an oversight on the part of the author, since the expression '!$' could have been used instead, now that the value of the variable '$' is no longer zero. This change would have saved one byte.

```
_$_:++$,
```

The expression '++$' sets to '2' the value of the variable '$', and assigns that value to the property '_$_'.

```
$_$$:({}+"")[$],
```

The expression '{}' returns a reference to an empty object. As above, this reference is converted to a string. The result is the string '[object Object]'. The third character of the string '[object Object]' ('b') is assigned to the property '$_$$'.

```
$$_$:($[$]+"")[$],
```

The expression '$[$]' would access the third entry in the array specified by the variable '$' if that array existed. However, since the variable '$' is not an array, the value 'undefined' is returned. As above, this value is converted to the string 'undefined'. The third character of the string 'undefined' ('d') is assigned to the property '$$_$'.

```
_$$:++$,
```

The expression '++$' sets to '3' the value of the variable '$', and assigns that value to the property '_$$'.

```
$$$_:(!""+"")[$],
```

The expression '""' returns an empty string. The operator '!' tests if the string is zero, which it is, resulting in the Boolean value 'true'. As above, this value is converted to the string 'true'. The fourth character of the string 'true' ('e') is assigned to the property '$$$_'. The use of the expression '!""+""' appears to be an oversight, since the string 'object' contains the letter 'e' immediately before the letter 'c'. Thus, this line could have been moved below the following line, and the expression '!""+""' could have been replaced with the expression '{}+""', to save one byte.

```
$__:++$,
```

The expression '++$' sets to '4' the value of the variable '$', and assigns that value to the property '$__'.

```
$_$:++$,
```

The expression '++$' sets to '5' the value of the variable '$', and assigns that value to the property '$_$'.

```
$$__:({}+"")[$],
```

The sixth character of the string '[object Object]' ('c') is assigned to the property '$$__'. The expression '({}+"")' is duplicated because it is not possible to reference a newly created property during the construction of an object. To reduce the size of the code, it would be necessary to use a second variable, where the 'object' string could be stored prior to the construction of the '$' object. Then the property assignment line would become '$$__:var2[$]', where 'var2' is the example name of the second variable.

```
$$_:++$,
```

The expression '++$' sets to '6' the value of the variable '$', and assigns that value to the property '$$_'.

```
$$$:++$,
```

The expression '++$' sets to '7' the value of the variable '$', and assigns that value to the property '$$$'.

```
$___:++$,
```

The expression '++$' sets to '8' the value of the variable '$', and assigns that value to the property '$___'.

```
$__$:++$
```

The expression '++$' sets to '9' the value of the variable '$', and assigns that value to the property '$__$'.

At this point, we have the properties '___', '$$$$', '__$', '$_$_', '_$_', '$_$$', '$$_$', '_$$', '$$$_', '$__', '$_$', '$$__', '$$_', '$$$', '$___', and '$__$'. They contain the values '0', 'f', '1', 'a', '2', 'b', 'd', '3', 'e', '4', '5', 'c', '6', '7', '8', and '9'.

$$+"\""+$_+"\""

```
$.$_=($.$_=$+"")[$.$_$]+($._$=$.$_[$.__
$])+($.$$=($.$+"")[$.__$])+((!$)+"")[$._$$]+($.__
=$.$_[$.$$_])+($.$=(!""+"")[$.__$])+($._=(!""+"")[$._
$_])+$.$_[$.$_$]+$.__+$._$+$.$
```

This line contains multiple assignments to properties, and character concatenation. It can also be written as follows:

```
$.$_=
          ($.$_=$+"")[$.$_$]
          +($._$=$.$_[$.__$])
          +($.$$=($.$+"")[$.__$])
          +((!$)+"")[$._$$]
          +($.__=$.$_[$.$$_])
          +($.$=(!""+"")[$.__$])
          +($._=(!""+"")[$._$_])
          +$.$_[$.$_$]
          +$.__
          +$._$
          +$.$
```

The references to '$' refer to the object now, not the value '9'. The '$.' in front of each property is required to access an existing property.

```
($.$_=$+"")[$.$_$]
```

The string '[object Object]' is assigned to the property '$_'. The sixth character ('$_$' is '5') of the string '[object Object]' ('c') is returned. There is a missed opportunity by the author here, since the '$$__' property also contains the character 'c'. Five bytes could have been saved by using that property instead, and moving the assignment of the '$_' property to the following line.

```
+($._$=$.$_[$.__$])
```

The second character ('__$' is '1') of the string '[object Object]' ('o') is assigned to the property '_$' and also returned.

```
+($.$$=($.$+"")[$.__$])
```

The non-existent property '$' is accessed, so the value 'undefined' is returned. This value is converted to a string, as above. The second character ('__$' is '1') of the string 'undefined' ('n') is assigned to the property '$$' and also returned.

```
+((!$)+"")[$._$$]
```

The expression '!$' tests if the reference to the object '$' is zero, which it is not, resulting in the Boolean value 'false'. The value is converted to a string, as above, and the fourth character ('_$$' is '3') of the string 'false' ('s') is returned. The parentheses surrounding the expression '!$' are unnecessary and could have been removed to save two bytes.

```
+($.__=$.$_[$.$$_])
```

The seventh character ('$$_' is '6') of the string '[object Object]' ('t') is assigned to the property '__' and also returned. This line could have been written as '$.__' if the line '__:(!$+"")[$]' were placed in the object construction before the second '++$'. However, this alternative saves no bytes.

```
+($.$=(!""+"")[$.__$])
```

The string 'true' is constructed, as above. The second character ('__$' is '1') of the string 'true' ('r') is assigned to the property '$' and also returned.

```
+($._=(!""+"")[$._$_])
```

The string 'true' is constructed, as above. The third character ('_$_' is '2') of the string 'true' ('u') is assigned to the property '_' and also returned. In this case, the entire line is poorly thought out, since the '_' property could be assigned in the object constructor in a shorter way. Three bytes could have been saved by placing the line '_:($[$]+"")[$]' before the second '++$', which would access the first character of the string 'undefined'.

```
+$.$_[$.$_$]
```

The sixth character ('$_$' is '5') of the string '[object Object]' ('c') is returned. This appears to be another oversight since, as above, five bytes could have been saved by using the '$$__' property instead. If the 'c' and 'o' characters were constructed using the alternative method, then the first assignment to the '$_' property would be completely unnecessary, resulting in the saving of another five bytes.

```
+$.__
```

The value of the property '__' ('t') is returned.

```
+$._$
```

The value of the property '_$' ('o') is returned.

```
+$.$
```

The value of the property '$' ('r') is returned. The result is that the string 'constructor' is assigned to the property '$_'.

$$+"\""+$$+"\""

```
$.$$=$.$+(!""+"")[$._$$]+$.__+$._+$.$+$.$$
```

This line contains more character concatenation. It can also be written as follows:

```
$.$$=
        $.$
        +(!""+"")[$._$$]
        +$.__
        +$._
        +$.$
        +$.$$
```

Once again, we will look at each line in turn:

```
$.$
```

The value of the property '$' ('r') is returned.

```
+(!""+"")[$._$$]
```

The string 'true' is constructed, as above. The fourth character ('_$$' is '3') of the string 'true' ('e') is returned. Again, the entire line appears to have been poorly thought out by the author, since nine(!) bytes could have been saved by using the '$$$_' property instead.

```
+$.__
```

The value of the property '__' ('t') is returned.

```
+$._
```

The value of the property '_' ('u') is returned.

```
+$.$
```

The value of the property '$' ('r') is returned.

```
+$.$$
```

The value of the property '$$' ('n') is returned. The result is that the string 'return' is assigned to the property '$$'. This assignment is completely unnecessary (see below).

$$$_+”\\”+__$+$$_+$$_+$_$_+(![]+””)[_$_]

```
$.$=($.___)[$.$_][$.$_]
```

This translates to the expression ‘(0)[“constructor”][“constructor”]’, and is assigned to the property ‘$’. The use of the expression ‘($.___)’ appears to be an oversight by the author, since the expression ‘[]’ could have been used instead. This change would have saved five bytes. Further, by assigning to the property ‘$_’ instead of ‘$’ at a cost of two bytes, the five bytes that are required to assign the property ‘$$’ and the four bytes that are required to reference it can be removed for an overall saving of seven bytes.

The expression ‘(0)[“constructor”][“constructor”]’ is equivalent to the expression ‘0.constructor.constructor’, but the brackets are required to delimit the two strings. Otherwise, the expression would appear to reference a single property several levels deep (‘$.$_.$.$_’). The expression ‘<number>.constructor’ is a reference to the constructor of a numeric object, while the expression ‘<object>.constructor.constructor’ is a reference to the constructor of a generic object.

```
$.$($.$($.$$+”\””+ENCODED+”\””)())()
```

This line decodes and executes the encoded host code using two constructor calls. The first constructor call decodes the encoded host code, and the second constructor call executes it, in this way:

```
$.$$+”\””+ENCODED+”\””
```

The value of the property ‘$$’ (‘return’, however as noted above, this property reference can be replaced by the ‘return’ concatenation from above) is used in the first constructor call to return the decoded host code that is bounded by the ‘"’s and represented here by ‘ENCODED’.

```
$.$(return”ENCODED”)()
```

This line translates to the expression ‘0.constructor.constructor(return“ENCODED”)()’, an anonymous function that returns the decoded host code as a string object. This is equivalent to executing the ‘eval()’ function.

```
$.$(DECODED)()
```

This line translates to the expression ‘0.constructor.constructor(DECODED)()’, an anonymous function that executes the decoded host code.

$$$_+$$+$$__+_$+$$_$+$$$_

Each character of the host code is encoded separately in one of several ways:

If the character is ‘"’ or ‘\’, then the character is prepended with the ‘\’ character (so ‘"’ becomes ‘\"’, and ‘\’ becomes ‘\\’).

If the character is a symbol already – that is, any of the following:

!"#$%&’()*+,-./:;<>=?@[]^_`{|}~

then the character is used exactly as it appears.

If the character is numeric, or one of the letters ‘a’ to ‘f’, ‘o’, ‘t’, or ‘u’ (and the check is case-sensitive), then the appropriate property is used.

If the character is the letter ‘l’, then the expression ‘(![]+"")[_$_]’ (that is, the third character of the string ‘false’) is used.

If the value of the character is less than 128, then the expression ‘\<val>’ is used, where ‘<val>’ is the decimal value of the character.

Otherwise, the expression ‘\u<val>’ is used, where ‘<val>’ is the hexadecimal value of the character.

It is interesting that some seemingly obvious encoding opportunities were missed. For example, since the numbers ‘0’–‘9’ are all available, it would be possible to use one of them to index the entire string for the special texts ‘false’, ‘true’, ‘[object Object]’ and ‘undefined’ (the string ‘constructor’ exists, but all of the characters in that string are present in the other four strings; the string ‘return’ also exists, but all of the characters in that string are present in the strings ‘true’ and ‘undefined’). Those four strings offer six more lower case alphabetic characters (‘ijlnrs’, leaving only ‘ghkmpqvwxyz’), and only the numbers ‘0’–‘5’ are needed to access the entire set. If the host does not require all of the numbers, then several lines could be removed from the object construction code. That would allow the code to be shortened further, since some variables could then use shorter names.

CONCLUSION

As it stands, JJEncode carries a relatively large constant body (even after applying the suggested size optimizations), which makes it easy to recognize. It would remain easy to recognize even if some ‘polymorphism’ were applied by using alternative indexes for the shared characters in the special texts (for example, the letter ‘c’ is at position ‘0’ in the word ‘constructor’, and at position ‘6’ in the string ‘[object’). It would also remain easy to recognize even if the variables were renamed as a result of discarding the unused numeric assignments. The only difficulty is in knowing at a glance what the encoded host does. However, the first constructor call (‘$.$(...)’) can be replaced with a function to display the result, or even to write it to disk, instead of executing it. Fortunately, the only way that someone could defend against that would be to change the code to the point where it is no longer JJEncode.