Increasingly, cybercriminals are turning to commodity software, sold on the black market or even open sourced. These kits allow attackers with relatively basic skills to launch malicious campaigns at scale. Exploit kits, for example, can be installed on compromised websites to exploit a wide range of vulnerabilities in a user’s web browser. Phishing kits provide most of the necessary components to run phishing schemes from development environments to graphics and code to create passable copies of legitimate websites. In some cases, the kits may even come with email lists, along with, of course, spamming software for delivering the emails.

These phishing kits are increasingly sophisticated and often include methods to avoid detection by client software, email providers, and gateways. Many of these obfuscation techniques aren’t particularly new, but the following six examples demonstrate popular (and, too often, effective) methods for hiding their code and malicious intent.

This post analyzes the following obfuscation techniques observed by Proofpoint researchers in multiple phishing campaigns:

- AES 256 with JavaScript in the browser
- Base64 refresh
- Flipped Base64 JavaScript encoding
- Combination Encoding
- Custom Encoding
- Xor Encoding in JavaScript
- Multibyte XOR Phishing Landing Obfuscation

**AES 256 with JavaScript in the browser**

In multiple campaigns, Proofpoint researchers have observed phishing pages that use legitimate AES encryption in JavaScript to encode their pages. In this case, the browser performs all of the decoding so that no normal HTML content for the landing can be observed on the wire.

In the example below, the code of the phishing web page attempts to fool the user into giving up their information. The page loads a JavaScript resource called 'hee.js,' which contains the AES decryption code. The variable hea2t contains the encrypted phishing landing page HTML code (Fig. 1).
Figure 1: Encrypted JavaScript

Below is the bottom of the same page. The document.write method is called on the output variable (Fig. 2), which will decrypt the content of the hea2t variable, effectively rendering the web page.

Figure 2: Document.write method calling the AES decryption routine on the hea2t variable

hee.js is a publicly available, open source implementation of AES (Fig. 3).
Figure 3: AES decryption routine within hee.js

The result is the decoded page shown below (Fig. 4).
This technique makes use of data URIs to obfuscate the phishing landing page by instructing the browser to load the base64 code as the page content. The browser will render the base64 code as html if it is a supported feature. If done correctly, the initial HTML content of the phishing page will not be observed on the wire. Proofpoint researchers have also observed this technique in multiple campaigns.
Figure 5: Data URI encoded HTML variable in phishing landing page

In this case, the code simply instructs the browser to render the base64 code as text/HTML data (Fig. 6).

Figure 6: Rendering base64 code as text/HTML

Once decoded, it is evident that the base64 encoded content is simple HTML.
An end-user could watch for the unusual URL structure, as seen in the URL bar below, even if the rendered page looks legitimate (Fig. 8):

In one interesting variant, we spotted JavaScript was embedded inside another data URI.
This nesting of data URIs will show a somewhat legitimate looking Google URL in the browser bar while the page contains actual phishing code.

The decoded base64 inside the data URI shows that an iframe is being loaded which contains the content of the phishing page.
Flipped Base64 JavaScript encoding

Multiple campaigns were observed making use of JavaScript and ‘backwards’ base64 to hide the phishing code. The document starts off defining a variable ‘OIO’ (Fig. 12):

```javascript
var OIO = 'oQKpyjBDCKB1GwNnLm8ZulGf0VmczSWaFtRHDoxHzuV3y2aJFmYwIfyVmcyVmZlJHf052m1D
Vxqcz3YbdHldGfImVcitb8xM1xHddowldHf5WwNtYRHYwY3dlmmWwDF4BHMLlEM9GwIDf3Wwaw3BjM8V2Zht
2d3cwWjM8VBMeyWYiYfMyxPyWYx8SUX3c5mcXnxyDFuWi9w8Fwy69ZU0GywnbD2bMjM8xWaH1WZyDFlWYyDwYU
VldDFrDFwyxyYjM8Mrwcd3cZGFwxDFk3Jb3N3chBnMHyZ5Gd4RmblbHchxHrthRhaDdowbSjMBzMyDFv9G
R3boxWx1JMBTSXIMBXWah1GFzV2YyV3bxxWY19Gbx3NyiJMB83ci5MynwYnYEdf1JMywHMSIjMBHwyLHIF2HjMyw
F3nyDFI1Mjy3M2EdF0Wibv3cYDFj3JbmnDHuWwbXrbLRxy13YBI3WsdvLY cywuyYnHf69Wb8RXJrJWzXHerGFk3J
LWytJWzXw3eMjyEyMyw7NkfxjJMywH11jMB1Hrz3GF5R2BFEMwXah1wRwWd0hZ2pVgauZLzyFwFWDFkFZ
RYxVhCvbnMywMyxYbxy9M512NYxZsUHdbfNMBTfM1WiwBVGbo1Gd8VGCHmH2c5LwDnZxVmcwJMBRHcpJ3yzN0M8RXDw5K
NwczZXYqnxBpdmch1GdmGw6Vg6V3FJ35Y8wanJXyTXt2R3BjM8SNanXYcLncmiDfRoXa3Bj
R3c8Vgdp3j3d82s3m2cL3bwof65BFw5RZw2bJjMBR8XcyZG2pmQwZwHTFRwXzXwHd8VGbo1GdNDNDkFVG
```

Figure 12: Defining a variable with a backwards base64 string for later reversal and decoding

Functions are defined at the end of the page. Function ‘01l’ handles the base64 decoding, while function ‘001’ takes care of reversing the string. The evaluation statement will reverse the contents of the 0l0 variable and then base64 decode it.
Figure 13: Function for decoding and reversing a string which will render a phishing page

Often, the resulting decoded base64 is further encoded, as can be seen in the next example “Combination encoding”.

```javascript
function Oll(data) {
    var 001l0I = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+=/";
    var o1, o2, o3, h1, h2, h3, h4, bits, i = 0,
    enc = '';
    do {
        h1 = 001l0I.indexOf(data.charAt(i++));
        h2 = 001l0I.indexOf(data.charAt(i++));
        h3 = 001l0I.indexOf(data.charAt(i++));
        h4 = 001l0I.indexOf(data.charAt(i++));
        bits = h1 << 18 | h2 << 12 | h3 << 6 | h4;
        o1 = bits >> 16 & 0xff;
        o2 = bits >> 8 & 0xff;
        o3 = bits & 0xff;
        if (h3 == 64) {
            enc += String.fromCharCode(o1)
        } else if (h4 == 64) {
            enc += String.fromCharCode(o1, o2)
        } else {
            enc += String.fromCharCode(o1, o2, o3)
        }
    } while (i < data.length);
    return enc;
}

function 001(string) {
    var ret = '';
    i = 0;
    for (i = string.length - 1; i >= 0; i--) {
        ret += string.charAt(i);
    }
    return ret;
}

eval(Oll(001(010)));
```
This process is invisible to the end user who will be presented with a legitimate looking phishing page.

**Combination encoding**

This particular encoding method takes the previous encodings and puts them all together, while adding a few tricks. It starts with the data URI method (Fig. 15)

Upon base64 decoding this we are presented with some a hex-encoded string (Fig. 16).
Figure 16: Decoded base64 presents hex encoding

Upon escaping the hex characters we are presented with the flipped base64 encoding method.
Figure 17: The now-familiar flipped base64 encoding

Upon flipping and base64 decoding, we are presented with a nested dean edwards JavaScript packer. This packer is very popular and easily decoded. Websites like http://jsbeautifier.org/ or tools like JSDetox (http://www.relentless-coding.org/projects/jsdetox) have no problem decoding it.

The initial packed code is shown below (Fig. 18):
Figure 18: The initial packed code utilizing a dean edwards JavaScript packer

After the first round of unpacking:
After the second round of unpacking, the code is starting to emerge:

**Figure 19: The code after initial unpacking**
Figure 20: Two rounds of unpacking

The last step to make it readable is to decode the URL encoding. Finally, we have the normalized phishing landing page.
Another phishing landing obfuscation technique to discuss here is a custom character replacement that Proofpoint researchers observed associated with an Apple Account phishing scheme. Initially we are presented with a page that consists of two eval statements and two arrays at the end of the second eval statement. Looking closely at the array, it appears that it could be useful in decoding.
Figure 22: The encoded phishing landing

Figure 23: The character key that exists at the end of the phishing landing

Figure 24: The unescaped content of the first unescape section in the encoded phishing landing
If we decode the first eval statement we observe that the JavaScript “unescape” variable is rewritten, so that when the second section evals the code, it runs the “new unescape” rather than the normal JavaScript unescape command.

The first variable in the function is the code to deobfuscate, the second is the encoded characters, and the third is the key. If the variables were rewritten to make more sense, the code would look something like this.

![Reformatted and rewritten code](image)

**Figure 25: Reformatted and rewritten code**

This is simply a character replace using a cipher that looks something like this:

```
60; - f
61; - m
62; - C
63; - H
64; - I
65; - d
66; - M
67; - E
68; - c
69; - V
610; - {  
611; - G
612; - r
613; - v
614; - ~
615; - a
616; - !
617; - S
618; - u
619; - Y
620; - <
621; - i
622; - I
623; - U
624; - g
625; - x
626; - @
627; - p
628; - B
629; - N
630; - h
631; - 
632; - 
633; - b
```

![Cipher for text replace obfuscation](image)

**Figure 26: Cipher for text replace obfuscation**
When we replace these characters on the page, we are presented with a mostly-decoded page:

![Decoded page after character replace](image)

However, this page still contains some decimal and \u00 encoded strings on it.

The Unicode-encoded strings appear below (Fig. 28):
Figure 28: Unicode-encoded strings in Apple Account phishing scheme

The decimal-encoded strings follow (Fig. 29):

Figure 29: Decimal-encoded strings in Apple Account phishing scheme
Simple Xor Encoding in JavaScript

This phishing landing we examined xor decodes charcode stored in a variable and then writes out the page via document.write. The obfuscated landing page begins as follows by defining an encoded string:

```

```Figure 30: First section of the obfuscated landing page with an excerpt of the encoded string

The JavaScript which will xor the string with 2 appears below (Fig. 31):

```

```Figure 31: JavaScript code that will xor decode

The resulting code after the xor still needs a another round of decoding:
Figure 32: Phishing page excerpt after xor decoding

After URL decoding, the normalized Dropbox phishing site looks like this (Fig. 33):
Figure 33: Fully decoded Dropbox phishing site
Multibyte XOR Phishing Landing Obfuscation

This method is among the more sophisticated phishing obfuscations we’ve observed. In this case, the initial landing is essentially two chunks of data that are unescaped and eval’ed.
Decoding the first eval statement (hex decode) yields the brains of the decoding (Fig. 35):

```javascript
function fa57e0be7e1(s) {
    var r = "";
    var tmp = s.split("17864328");
    s = unescape(tmp[0]);
    k = unescape(tmp[1] + "817390");
    for (var i = 0; i < s.length; i++) {
        r += String.fromCharCode((parseInt(k.charAt(i%k.length))^s.charCodeAt(i))+-7);
    }
    return r;
}
```

*Figure 35: First eval statement after hex-decoding*

While it doesn’t involve much code, this is a fairly sophisticated obfuscation method as far as phishing goes. The second block of code decodes to eval the large chunk of data as the s variable in the above code.

The tmp variable becomes an array by splitting the data into two bits of information where “17864328” occurs in the variable. tmp[0] holds the encoded data, while tmp[1] holds what will be used as a key for decoding.

Next, k.charAt(0) will evaluate to the first character in the key variable which is 4.

Evaluate further, i%k.length for the first loop will be 0.

String.fromCharCode((parseInt(k.charAt(i%k.length)) ^ s.charCodeAt(i))+-7);

This evaluates to essentially 4 ^ 71 which evaluates to 67.
String.fromCharCode((parseInt(k.charAt(i%k.length))^s.charCodeAt(i))+-7);

The next step just subtracts 7 from 67 and parses as an integer, so the result is 60.

String.fromCharCode((parseInt(k.charAt(i%k.length))^s.charCodeAt(i))+-7);

Finally, the expression converts 60 decimal to ascii, so we end up with “<”, which is saved in the r variable.

Subsequent loop values would look something like this:

String.fromCharCode(5^45+-7) == !
String.fromCharCode(1^106+-7) == d
String.fromCharCode(5^115+-7) == o
...

The fully decoded value is then written to the page via document.write where we see normal html.

Figure 37: The deobfuscated page
Conclusion

As phishing schemes become more sophisticated, the landing pages to which users are directed via email or social media lures are increasingly obfuscated to avoid detection by endpoints and gateway appliances. With few exceptions, these landing pages are legitimate-looking copies of the sites indicated in the lures, e.g., Dropbox, DHL, or Apple. More importantly, though, while many of the obfuscation techniques we have examined here are extremely sophisticated, they are often being incorporated in phishing kits, meaning that even inexperienced cybercriminals can now stage attacks and build landing pages with commodity tools.

For businesses, individuals, and vendors, the challenge is to implement detection techniques that can decode the obfuscation as well as to increase awareness of the warning signs for phishing campaigns.