Vulnerability Advisory

ISEC-V2021-02

WP Bannerize WordPress plugin
SQL injection leading to arbitrary file deletion

INTRINSEC

Innovative by design

Site internet
www.intrinsec.com

Blog
www.intrinsec.com/blog

Twitter
@Intrinsec

TOUR CBX, 20ème ETAGE, 1 PASSERELLE DES REFLETS, 92913 PARIS LA DEFENSE CEDEX • TEL +33 1 41 91 77 77
INTRINSEC SECURITE • SAS AU CAPITAL DE 2 760 680 € • SIREN : 812 535 284 • APE : 6203Z

ADVISORY ISEC-V2021-02

Vendor: Gfazioli

Product: WP-Bannerize

Title: SQLI injection leading to arbitrary file deletion

Intrinsec ID: ISEC-V2021-02

Published: 15/10/2021

Last updated: 13/07/2021

Risk: Moderate

Exploitation: remote with authenticated access

Impact:

- Security feature bypass
- Data corruption
- Sensitive data exposure
- Denial of service

Description:

By sending a malicious payload as a privileged user (other than subscriber), an attacker can first perform an SQL injection and then delete arbitrary files on the server. This may lead to a loss of data confidentiality and a system denial.

Vulnerable versions: 4.0.2 - Older versions are also affected since 2.0.0

Solutions: Validate user inputs.

Credits:

Vulnerability discovered by Margaux DABERT from Intrinsec.

History:

2021-07-12: Vulnerability identified

2021-07-16: Communication with the plugin editor

2021-07-19:

- Communication of the advisory to WordPress
- Plugin removed from the official wordpress plugin repository

2021-10-05: Obtaining the CVE number CVE-2021-39351

DETAILED APPROACH

While reviewing the source code of the WP Bannerize plugin (https://wordpress.org/plugins/wp-bannerize/), we identified that the following functions in the */Classes/wpBannerizeAdmin.php* file use the values of POST requests without filtering their content in order to perform SQL queries:

- "updateBannerFromLocal": function to modify the banner;

```
  // Classes/wpBannerizeAdmin.php : updateBannerFromLocal : Lines 1681 to 1687
  function updateBannerFromLocal()
  {
    global $wpdb;

    // Retrive image info
    $sql = sprintf( "SELECT * FROM `%s` WHERE id = %s", $this-
>table_bannerize, $_POST['id'] );
    $row = $wpdb->get_row( $sql );
```

- "setBannerToTrash": function to put the banner in the trash;

```
  // Classes/wpBannerizeAdmin.php : setBannerToTrash : Lines 1558 to 1563
  function setBannerToTrash( $id = null )
  {
    global $wpdb;
    $id  = ( is_null( $id ) ) ? $_POST['id'] : $id;
    $sql = sprintf( "UPDATE `%s` SET trash = '1' WHERE id IN(%s)", $this-
>table_bannerize, $id );
    $wpdb->query( $sql );
```

- "unsetBannerToTrash": function to retrieve the banner from the trash;

```
  // Classes/wpBannerizeAdmin.php : setBannerToTrash : Lines 1575 to 1580
  function unsetBannerToTrash( $id = null )
  {
    global $wpdb;
    $id  = ( is_null( $id ) ) ? $_POST['id'] : $id;
    $sql = sprintf( "UPDATE `%s` SET trash = '0' WHERE id IN(%s)", $this-
>table_bannerize, $id );
    $wpdb->query( $sql );
```

- "deleteBanner": function to remove the banner from the trash.

```
// Classes/wpBannerizeAdmin.php : deleteBanner : Lines 1593 to 1606
function deleteBanner( $id = null )
{
  global $wpdb;

  $id = ( is_null( $id ) ) ? $_POST['id'] : $id;

  // Delete from disk only local banner.
  $wpBannerizeBannerType = $wpdb->get_var( "SELECT `banner_type` FROM `" . $this-
>table_bannerize .
  "` WHERE `id` = " . $id );

  if ( $wpBannerizeBannerType == '1' ) {
    $filename = $wpdb->get_var( "SELECT `realpath` FROM `" . $this-
>table_bannerize . "` WHERE `id` = " . $id );
    @unlink( $filename );
  }
```

In all these functions, the POST "id" parameter is used without checking that it is a numeric format and not within a prepared SQL statement, which enables an attacker to enter a string, like an SQL query. This leads to an SQL injection attack.

Moreover, in the "deleteBanner" function, the value is used in two places, once to retrieve the "banner_type" (line 1600) and a second time to retrieve the "realpath" (line 1604) from the banner file corresponding to this identifier. When the first request returns the value "1", then we enter a condition which executes again a vulnerable request and which result is sent to the ''unlink" function deleting a file. This explains the double evaluation of the SQL sleep query in the previous capture.
Thus, when the injected request first returns the value 1 and then the path of a file, the file is deleted. To exploit this, we used the following approach:

- We have a single payload and we want to return two different values;
- The only difference we have identified between the two requests is the time at which they are executed;
- So we can choose the behavior depending on when the request is executed;
- We can control the time difference between the two requests with a "sleep" function;
- So we define a reference time of 10 seconds in the future and we make an if condition:
  - If the current time is less than the reference time, then we return "1" and we make a sleep of 20 seconds;
  - If the current time is greater than the reference time, then we return the path of a file to delete.

Any authenticated user other than a subscriber can perform this operation. However, the files that can be deleted are only those for which the user running the web server (www-data by default) has write access on.

This attack could enable an attacker for example to make a site unavailable (Denial Of Service - DOS) or to bypass some authorization restrictions by deleting the ".htaccess" file in the case of an Apache server.