Seguridad Informática: Modelos de Seguridad permisivos como mecanismos de prevención de vulnerabilidades

Los modelos de seguridad permisivos, a pesar de lo complejo de su análisis, son los únicos capaces de generar aplicaciones invulnerables, ya que al no estar basados en la prevención de riesgos conocidos no dan lugar a descubrir «nuevos agujeros», pues los propios modelos, no son más que «ilusiones ópticas».

Cuando comenzamos a programar, una de las primeras cosas que aprendemos en materia de seguridad de aplicaciones, es a filtrar entradas del usuario, quitándoles todo aquel caracter prohibido. Hoy me pregunto **¿de verdad en algún momento nos hemos sentido más listos que el resto del mundo?**

Prohibir es una tarea tediosa e infinita; cuanto más se analizan e investigan los posibles «flancos de ataque» en una aplicación, la tarea de prohibir puede llegar a hacerse interminable, ya que en la Ingeniería de Software aplica, indirectamente, el mismo principio que en el derecho penal: «no existe prohibición sin pena».

En los sistemas informáticos, al igual que un código penal, cada prohibición debe especificarse minuciosa y detalladamente. Pero la peor parte -para la informática- es que al prohibir, necesariamente, se debe indicar «cuál será la pena», solo que para el Software, esto es metafórico. Y cuando hablamos de «establecer una pena», nos estamos refiriendo a **implementar complejos algoritmos que respondan frente a cada acto prohibido**.

Prohibimos el ingreso de comillas simples, entonces, al igual que en el derecho, no basta con decirle a un individuo que «está prohibido matar», porque en realidad, no está prohibido, está «penado». **Con colocar un cartel que diga «por favor, no ingrese comillas simples» no alcanza; debemos crear un algoritmo que se encargue de erradicarlas.** Pero esto no termina aquí y debe repetirse el mismo esquema de procedimientos, con cada «cosa» que deseemos prohibir en nuestro sistema.

> *Cuando implementamos modelos de seguridad por prohibición,* ***no estamos teniendo en cuenta que no somos omnipotentes***

En 1999, la mayoría de las aplicaciones Web que implementaban sistemas de registro de usuarios, buscadores basados en bases de datos y afines, no efectuaban ningún tipo de filtro como los de hoy en día. Hasta que algún día alguien habrá visto que un maldito `%` en un buscador, podía romper el `LIKE` de una sentencia SQL. Y eso mismo, sucede todos los días.

> *Lamentablemente, cada vez más gente invierte su tiempo en mejorar técnicas de pentesting y en crear herramientas de exploiting, en vez de ampliar sus capacidades cognitivas intentando* ***hallar verdaderas soluciones mediante investigación científica***

Todos los días se «descubren» nuevas formas de destruir los sistemas informáticos. Se les suele llamar «vulnerabilidades» pero desde mi punto de vista, todo sistema informático aunque se encuentre libre de toda vulnerabilidad, tendrá al menos una única vulnerabilidad si se implementa un modelo de seguridad prohibitivo: **la falsa omnipotencia de sus desarrolladores**.

> *El modelo de seguridad prohibitivo es la primera vulnerabilidad de un sistema informático*

La humanidad suele no tomarse la molestia de analizar y descubrir conexiones lógicas entre dos o más hechos que en apariencia no se encontrarían relacionados. Dicho de forma más simple, **la mayoría de las personas no se molesta en «atar cabos»**. Pues de hacerlo, la seguridad informática debería ser una ciencia exacta ya que **en nuestra vida cotidiana, tenemos sobrados ejemplos de cómo «el prohibir» siempre se queda detrás de los acontecimientos**, ya que como marca un viejo dicho popular *«hecha la Ley, hecha la trampa»*. Pero a diferencia del derecho, en el caso de la Ingeniería de Software, esto tiene solución y consiste en implementar **«modelos de seguridad permisivos»**.

¿Qué son y en qué consisten los modelos de seguridad permisivos?

No es nada difícil deducir. Se trata de modelos de seguridad que para establecer sus políticas, se basan en aquello que se permite en una aplicación en vez de invertir cientos y miles de bytes definiendo algoritmos centrados en lo que se prohíbe.

Características de los modelos de seguridad permisivos

Estos modelos suelen dar libertades al usuario que comúnmente se le niegan y hasta incluso, podrían considerarse impensables como sería el caso de permitirle elegir un nombre de usuario conformado por caracteres no alfanuméricos como podrían ser comillas simples. Es por ello, que **son considerados modelos de seguridad «liberales»** en los cuales la criptografía -tanto desde la codificación hasta la encriptación- juegan un papel protagónico, puesto que son el pilar del modelo.

> *La **criptografía** y la **codificación** son el pilar de los modelos de seguridad permisivos*

Es entonces, que como principales características podemos listar las siguientes:

1. Dan amplia libertad de acción y movimiento al usuario;
2. Emplean sistemas criptográficos con mucha frecuencia;
3. Utilizan la (re)codificación como base del modelo liberal.

Ventajas y desventajas de los modelos de seguridad permisos

Los modelos de seguridad permisivos (o liberales) suelen traer aparejadas muchísimas más ventajas y por consiguiente beneficios, que desventajas.

Las desventajas suelen ser más bien de índole social ya que las aplicaciones que implementan modelos de seguridad permisivos, suelen ser blanco fácil de los «aprendices de *crackers*» o de conductas delictivas semejantes ya que inicialmente, los futuros delincuentes suelen dejarse llevar por el entusiasmo de creer que «el exceso de libertades» se debe a una conducta negligente de los programadores.

Sin embargo, **el cese de los intentos de ataque suele producirse mucho más rápido** que en las aplicaciones que implementan modelos de seguridad restrictivos (o prohibitivos) ya que en su mayoría, **este tipo de delincuentes buscan satisfacer sus fantasías de forma inmediata y al no conseguirlo, abandonan para buscar una víctima más vulnerable que les facilite el placer inmediato**.

No obstante es válido recordar que si bien todo perfil criminal busca la satisfacción inmediata de sus fantasías, existe porción más reducida de delincuentes con una psiquis mucho más compleja que podrían permanecer años tratando de violar la seguridad de un sistema por el mero hecho de sentir el placer que no logran alcanzar en otros aspectos de sus vidas. Por consiguiente, **sería un error asumir que los modelos de seguridad permisivos evitan por completo los intentos de ataques a corto plazo**.

Si bien hasta aquí toda desventaja parece estar compensada por una ventaja o beneficio, existe un factor que convierte a los modelos de seguridad permisivos en algo difícil de ser implementados pues solo serán realmente seguros si la política de permisos es delineada por profesionales que poseen un conocimiento profundo del funcionamiento interno de un sistema informático completo y de la teoría de ordenadores, pues **una política permisiva mal pensada podría generar una aplicación mucho más vulnerable** que aquella que implemente un modelo restrictivo.

Ejemplos prácticos de políticas de seguridad basadas en modelos permisivos

Aunque todo parezca -en principio- demasiado novedoso, algunas políticas resultarán familiares ya que en algún momento se habrán puesto en práctica y seguramente, en más de una oportunidad. Sin embargo, **implementar políticas permisivas de forma aislada en un modelo restrictivo, puede ser tan riesgoso como no implementar ninguna.** Por lo tanto, los siguientes ejemplos solo se exponen a fin de alcanzar una mejor comprensión sobre los modelos de seguridad permisivos y no deben ser interpretados como una guía de seguridad única.

Sin duda, la política permisiva más implementada en aplicaciones es el paso de nombres de archivo por la URI, frecuentemente utilizado para la muestra o descarga de contenido estático. Sin embargo, **la misma política podría estar planteada tanto en un marco permisivo como en uno restrictivo, dependiendo de la óptica desde la cual se lo haya elaborado**.

```
$solicitud = isset($_GET['page']) : strtolower($_GET['page']) : 'default';
$archivo = SOME_PATH . "/html/{$solicitud}.html";
$contenido_interno = file_get_contents($archivo);
print str_replace('{contenido}', $contenido_interno, SOME_TEMPLATE);
```

Hasta aquí, vemos el concepto sin ninguna medida de seguridad definida. Suponiendo que en la carpeta `SOME_PATH . "/html/"` todos los archivos con extensión `html` que se almacenan puedan ser accesibles por el usuario, a lo sumo se podría incluir una validación de existencia del archivo para evitar un fallo en el código que revelase información del sistema:

```
$solicitud = isset($_GET['page']) : strtolower($_GET['page']) : 'default';
$archivo = SOME_PATH . "/html/{$solicitud}.html";
if(file_exists($archivo)) {
    $contenido_interno = file_get_contents($archivo);
} else {
    $contenido_interno = '';
}
print str_replace('{contenido}', $contenido_interno, SOME_TEMPLATE);
```

E incluso, el algoritmo anterior podría optimizarse aún más definiendo un contenido interno por defecto y haciendo el algoritmo mucho más seguro y legible:

```
$solicitud = isset($_GET['page']) : strtolower($_GET['page']) : 'default';
$archivo = SOME_PATH . "/html/{$solicitud}.html";
$contenido_interno = '';
if(file_exists($archivo)) $contenido_interno = file_get_contents($archivo);
print str_replace('{contenido}', $contenido_interno, SOME_TEMPLATE);
```

Conceptualmente, a pesar de estar «permitiendo» al usuario pasar el nombre de un archivo por la URI, le

estaríamos dando un enfoque «prohibitivo» ya que si la carpeta `SOME_PATH . "/html/"` almacenara un archivo que no se quisiese poder tratar como «contenido interno» habría que «prohibir su acceso»:

```
$solicitud = isset($_GET['page']) : strtolower($_GET['page']) : 'default';
$archivo = SOME_PATH . "/html/{$solicitud}.html";
$contenido_interno = '';
$es_template = ($solicitud == 'template');
$es_menu_admin = ($solicitud == 'menu_admin');
if(file_exists($archivo) && !$es_template && !$es_menu_admin) {
    $contenido_interno = file_get_contents($archivo);
}
print str_replace('{contenido}', $contenido_interno, SOME_TEMPLATE);
```

Y en el mejor de los casos, el almacenamiento de archivos que no se quisiesen tratar como contenido interno, dentro de esa carpeta **se estarían restringiendo «solo por políticas de seguridad y no por cuestiones arquitectónicas»**. Esto es una clara consecuencia de las políticas restrictivas.

Sin embargo, no debe confundirse «permisivo» con «descuidado». **Una política permisiva, es aquella que «define lo que se puede hacer en vez de definir aquello que está prohibido»**. Entonces, definir cuáles archivos están permitidos, será mucho más *«limpio»* y seguro que crear algoritmos para custodiar «las prohibiciones»:

```
$archivos_permitidos = array('default', 'contacto', 'newsletter', 'noticias', 'login');
$solicitud = isset($_GET['page']) : strtolower($_GET['page']) : 'default';
if(!in_array($solicitud, $archivos_permitidos)) $solicitud = 'default';
$archivo = SOME_PATH . "/html/{$solicitud}.html";
$contenido_interno = file_get_contents($archivo);
print str_replace('{contenido}', $contenido_interno, SOME_TEMPLATE);
```

Como se puede ver en el último ejemplo, estableciendo una política permisiva obtenemos:

- **un código que no requiere de mantenimientos futuros** ya que el algoritmo se mantiene intacto incluso aunque se modifique el contenido de la carpeta en cuestión. De hecho, el *array* de «archivos permitidos» debería ser configurable desde un *settings*;

- **algoritmos con menos instrucciones** ya que no será necesario verificar la existencia del archivo en nombre de la seguridad. Si se decidiese efectuar dicha validación, sería por cuestiones de *usabilidad* pero no de seguridad.

Anteriormente, también había comentado que se podía llegar a extremos impensables como permitir todo tipo de caracteres en el nombre de usuario de un sistema de registro y autenticación. Ese es otro ejemplo de política permisiva. En este caso, no solo nos ahorramos largas validaciones en el código sino que además, evitamos tener que «limpiar» la entrada del usuario. Con solo encriptar (*hashear*) el nombre del usuario, guardarlo encriptado y luego, al momento de autenticar, compararlo mediante su *hash* al igual que se hace con las contraseñas, sería suficiente:

```
$username = md5($_POST['username']);
```

Aquí lo permisivo, a pesar de lo que se podría creer en un principio, no es el ingreso de caracteres «extraños». En realidad, de forma indirecta, estamos definiendo que solo se permiten caracteres alfanuméricos, puesto que un *hash* MD5 cifrará la cadena resultando en otra que solo es alfanumérica.

Un buen programador es aquel que mejor engaña al usuario

Convertirse en un timador suena trágicamente espantoso. Sin embargo, como todo lo «virtual» en nuestros queridos resultantes «ceros y unos», no necesariamente convertirá un «timo» en algo malo, sino que será aquello que otorgue gran libertad de movimiento a los usuarios generando una significativa reducción en la curva de aprendizaje que un usuario transita hasta entender la aplicación, que lo haga sentir cómodo pero que al mismo tiempo, no ponga en riesgo la aplicación. Se trata de **«ofrecer espejos de colores»** (pero sin el cruel objetivo de la colonización).

Me sucede con frecuencia que al momento de analizar requerimientos con mis alumnos, al igual que los usuarios se dejan llevar solo y únicamente por aspectos visuales. **Hace décadas atrás, era prácticamente impensable que un programador analizase un requerimiento con «ojos de usuario».** Sin embargo, desde la llegada de «las ventanas» en modo gráfico y dispositivos como el *mouse*, la informática se hizo accesible a todos nosotros mediante ordenadores personales y en definitiva, **la *«visión amigable e intuitiva»* que los gráficos otorgan al común de las personas, facilitaron el acceso a la programación, de gran parte de la población**.

Desde entonces, se ha convertido en «moneda corriente» observar a programadores confundir el problema a resolver (requerimiento visual en el 99% de los casos) con la solución del mismo. Y esto, es importante entenderlo ya que justamente, **los modelos de seguridad permisivos se basan en «mostrar al usuario lo que desea ver»** pero aquello que el usuario desea ver, debe ofrecérsele como una ilusión óptica, una fantasía. Para nosotros, lo que el usuario vea, deberá ser el resultado de convertir «aquello que permitimos» en lo que el usuario «crea estar viendo».

Por ejemplo, le hago creer al usuario que está viendo esto:

```
function add_confirm() {
    links = document.getElementsByTagName('a');
    for(i=0; i<links.length; i++) {
        if(links[i].title.indexOf('eliminar') == 0) {
            links[i].onclick = function() {
                partes = this.href.split('/');
                preg = "¿Confirma que desea eliminar este " + partes[4] + "?";
                if(confirm(preg)) {
                    location.href = this.href;
                } else {
                    return false;
                }
            };
        }
    }
}
```

Pero en realidad, lo anterior, es una ilusión óptica, resultado de haber convertido esto otro:

```
functionECODG160ECODCaddECODG95ECODCconfirmECODG40ECODCECODG41ECODCECODG160ECODCECODG123ECODCECODS
ECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODClinksECODG160ECODCECODG61ECODCECODG160ECODCdoc
umentECODG46ECODCgetElementsByTagNameECODG40ECODCECODG39ECODCaECODG39ECODCECODG41ECODCECODG59ECODC
ECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCforECODG40ECODCiECODG61ECODC0ECODG59ECODC
ECODG160ECODCiECODG60ECODClinksECODG46ECODClengthECODG59ECODCECODG160ECODCiECODG43ECODCECODG43ECOD
CECODG41ECODCECODG160ECODCECODG123ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCEC
ODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCifECODG40ECODClinksECODG91ECODCiECODG93ECODCECOD
G46ECODCtitleECODG46ECODCindexOfECODG40ECODCECODG39ECODCeliminarECODG39ECODCECODG41ECODCECODG160EC
ODCECODG61ECODCECODG61ECODCECODG160ECODC0ECODG41ECODCECODG160ECODCECODG123ECODCECODSECODG160ECODCE
CODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160
ECODCECODG160ECODCECODG160ECODCECODG160ECODClinksECODG91ECODCiECODG93ECODCECODG46ECODConclickECODG
160ECODCECODG61ECODCECODG160ECODCfunctionECODG40ECODCECODG41ECODCECODG160ECODCECODG123ECODCECODSEC
ODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160E
CODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECO
DG160ECODCpartesECODG160ECODCECODG61ECODCECODG160ECODCthisECODG46ECODChrefECODG46ECODCsplitECODG40
ECODCECODG39ECODCECODG47ECODCECODG39ECODCECODG41ECODCECODG59ECODCECODSECODG160ECODCECODG160ECODCEC
ODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160E
CODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCpregECODG160ECOD
CECODG61ECODCECODG160ECODCECODG34ECODCECODG191ECODCConfirmaECODG160ECODCqueECODG160ECODCdeseaECODG
160ECODCeliminarECODG160ECODCesteECODG160ECODCECODG34ECODCECODG160ECODCECODG43ECODCECODG160ECODCpa
rtesECODG91ECODC4ECODG93ECODCECODG160ECODCECODG43ECODCECODG160ECODCECODG34ECODCECODG63ECODCECODG34
ECODCECODG59ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECO
DCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG
160ECODCECODG160ECODCECODG160ECODCifECODG40ECODCconfirmECODG40ECODCpregECODG41ECODCECODG41ECODCECO
DG160ECODCECODG123ECODCECODG160ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG
160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECOD
CECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG1
60ECODClocationECODG46ECODChrefECODG160ECODCECODG61ECODCECODG160ECODCthisECODG46ECODChrefECODG59EC
ODCECODG160ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECOD
CECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG1
60ECODCECODG160ECODCECODG160ECODCECODG125ECODCECODG160ECODCelseECODG160ECODCECODG123ECODCECODSECOD
G160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECO
DCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG
160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCreturnECODG160ECODCfalseECODG59ECODCEC
ODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECOD
G160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECO
DCECODG160ECODCECODG125ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODC
ECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG12
5ECODCECODG59ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160EC
ODCECODG160ECODCECODG160ECODCECODG125ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECOD
CECODG125ECODCECODSECODG125ECODC
```

Como puede verse en el ejemplo anterior, es un simple e inocuo conjunto de caracteres alfanuméricos, que solo y únicamente del lado del cliente «adoptan forma», la forma que nosotros como programadores, le hacemos ver al usuario.

Conclusión

Implementar modelos de seguridad permisivos no es algo sencillo. Por el contrario, su estrategia requiere de cierta complejidad de análisis a la que tal vez, la mayoría de los programadores no está acostumbrado. Sin embargo, no es imposible. Las grandes ventajas y sobre todo, la innumerable cantidad de beneficios que los modelos de seguridad permisivos aportan a los sistemas informáticos, justifican la inversión de tiempo y esfuerzo que la definición, desarrollo e implementación de estos modelos, demandan al programador.

No existen fórmulas únicas y por consiguiente, mucho menos las hay mágicas, para definir estos modelos. Toda política de seguridad, toda medida a implementar, deberán ser el producto de un **análisis objetivo, abstracto y cuidado** de lo que realmente se necesita.

Es imposible implementar verdaderos modelos de seguridad permisivos si se analizan las aplicaciones con la misma óptica del usuario. Es importante entender que aquello que el usuario ve, no necesariamente será real. De la misma forma en la que nosotros hallamos 20 objetos complejos en lo que el usuario solo ve como un sencillo formulario con 10 campos en los cuáles escribir (o elegir) datos, debemos ver toda la aplicación. Basarnos en la premisa de que todo, absolutamente todo lo que el usuario vea, será una gran fantasía.

Insisto a mis alumnos de forma constante en que desarrollen sobre la misma plataforma en la que finalmente se ejecutará la aplicación y en que eviten tanto como les sea posible, el uso de herramientas gráficas para trabajar. Y no lo hago por «capricho», sino por el contrario, **lo hago para ir acostumbrando al cerebro a razonar cada vez de forma lo más abstracta, independiente y aislada de «fantasías ópticas» posible.** Porque cuanto más «amigable» sea el entorno de trabajo, menos esfuerzo mental requerirá por parte del programador, pues cuanto menos esfuerzo mental esté haciendo el programador en su trabajo, más estará siendo víctima él mismo, de las «ilusiones ópticas» producidas por un programador más astuto y entonces, la pregunta es:

> *¿cómo logrará un programador, hacer aplicaciones seguras si él mismo es «víctima» de «ilusiones ópticas» que no es capaz de entender ni descifrar?*

Un usuario que programa, no es lo mismo que un programador. Si te animas a ser programador, lograrás darte cuenta de que **en realidad, las aplicaciones invulnerables existen y son posibles**.