Unvalidated Redirects and Forwards Cheat Sheet

Introduction

Unvalidated redirects and forwards are possible when a web application accepts untrusted input that could cause the web application to redirect the request to a URL contained within untrusted input. By modifying untrusted URL input to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials.

Because the server name in the modified link is identical to the original site, phishing attempts may have a more trustworthy appearance. Unvalidated redirect and forward attacks can also be used to maliciously craft a URL that would pass the application's access control check and then forward the attacker to privileged functions that they would normally not be able to access.

Safe URL Redirects

When we want to redirect a user automatically to another page (without an action of the visitor such as clicking on a hyperlink) you might implement a code such as the following:

Java

```
response.sendRedirect("http://www.mysite.com");
```

PHP

```
<?php
/* Redirect browser */
header("Location: http://www.mysite.com");
/* Exit to prevent the rest of the code from executing */
exit;
?>
```

ASP .NET

```
Response.Redirect("~/folder/Login.aspx")
```

Rails

```
redirect_to login_path
```

Rust actix web

```
  Ok(HttpResponse::Found()
        .insert_header((header::LOCATION, "https://mysite.com/"))
        .finish())
```

In the examples above, the URL is being explicitly declared in the code and cannot be manipulated by an attacker.

Dangerous URL Redirects

The following examples demonstrate unsafe redirect and forward code.

Dangerous URL Redirect Example 1

The following Java code receives the URL from the parameter named `url` (GET or POST) and redirects to that URL:

```
response.sendRedirect(request.getParameter("url"));
```

The following PHP code obtains a URL from the query string (via the parameter named `url`) and then redirects the user to that URL. Additionally, the PHP code after this `header()` function will continue to execute, so if the user configures their browser to ignore the redirect, they may be able to access the rest of the page.

```
$redirect_url = $_GET['url'];
header("Location: " . $redirect_url);
```

A similar example of C# .NET Vulnerable Code:

```
string url = request.QueryString["url"];
Response.Redirect(url);
```

And in Rails:

```
redirect_to params[:url]
```

Rust actix web

```
  Ok(HttpResponse::Found()
        .insert_header((header::LOCATION, query_string.path.as_str()))
        .finish())
```

The above code is vulnerable to an attack if no validation or extra method controls are applied to verify the certainty of the URL. This vulnerability could be used as part of a phishing scam by redirecting users to a malicious site.

If no validation is applied, a malicious user could create a hyperlink to redirect your users to an unvalidated malicious website, for example:

```
http://example.com/example.php?url=http://malicious.example.com
```

The user sees the link directing to the original trusted site (`example.com`) and does not realize the redirection that could take place

Dangerous URL Redirect Example 2

ASP .NET MVC 1 & 2 websites are particularly vulnerable to open redirection attacks. In order to avoid this vulnerability, you need to apply MVC 3.

The code for the LogOn action in an ASP.NET MVC 2 application is shown below. After a successful login, the controller returns a redirect to the returnUrl. You can see that no validation is being performed against the returnUrl parameter.

ASP.NET MVC 2 LogOn action in `AccountController.cs` (see Microsoft Docs link provided above for the context):

```
[HttpPost]
 public ActionResult LogOn(LogOnModel model, string returnUrl)
 {
   if (ModelState.IsValid)
   {
     if (MembershipService.ValidateUser(model.UserName, model.Password))
     {
       FormsService.SignIn(model.UserName, model.RememberMe);
       if (!String.IsNullOrEmpty(returnUrl))
       {
         return Redirect(returnUrl);
       }
       else
       {
         return RedirectToAction("Index", "Home");
       }
     }
     else
```

```
      {
        ModelState.AddModelError("", "The user name or password provided is
incorrect.");
      }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
  }
```

Dangerous Forward Example

When applications allow user input to forward requests between different parts of the site, the application must check that the user is authorized to access the URL, perform the functions it provides, and it is an appropriate URL request.

If the application fails to perform these checks, an attacker crafted URL may pass the application's access control check and then forward the attacker to an administrative function that is not normally permitted.

Example:

```
http://www.example.com/function.jsp?fwd=admin.jsp
```

The following code is a Java servlet that will receive a `GET` request with a URL parameter named `fwd` in the request to forward to the address specified in the URL parameter. The servlet will retrieve the URL parameter value from the request and complete the server-side forward processing before responding to the browser.

```
public class ForwardServlet extends HttpServlet
{
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException {
    String query = request.getQueryString();
    if (query.contains("fwd"))
    {
      String fwd = request.getParameter("fwd");
      try
      {
        request.getRequestDispatcher(fwd).forward(request, response);
      }
      catch (ServletException e)
      {
        e.printStackTrace();
      }
    }
```

```
    }
}
```

Preventing Unvalidated Redirects and Forwards

Safe use of redirects and forwards can be done in a number of ways:

- Simply avoid using redirects and forwards.
- If used, do not allow the URL as user input for the destination.
- Where possible, have the user provide short name, ID or token which is mapped server-side to a full target URL.
    - This provides the highest degree of protection against the attack tampering with the URL.
    - Be careful that this doesn't introduce an enumeration vulnerability where a user could cycle through IDs to find all possible redirect targets
- If user input can't be avoided, ensure that the supplied **value** is valid, appropriate for the application, and is **authorized** for the user.
- Sanitize input by creating a list of trusted URLs (lists of hosts or a regex).
    - This should be based on an allow-list approach, rather than a block list.
- Force all redirects to first go through a page notifying users that they are going off of your site, with the destination clearly displayed, and have them click a link to confirm.

Validating URLs

Validating and sanitising user-input to determine whether the URL is safe is not a trivial task. Detailed instructions how to implement URL validation is described in Server Side Request Forgery Prevention Cheat Sheet

References

- CWE Entry 601 on Open Redirects.
- WASC Article on URL Redirector Abuse
- Google blog article on the dangers of open redirects.
- Preventing Open Redirection Attacks (C#).