

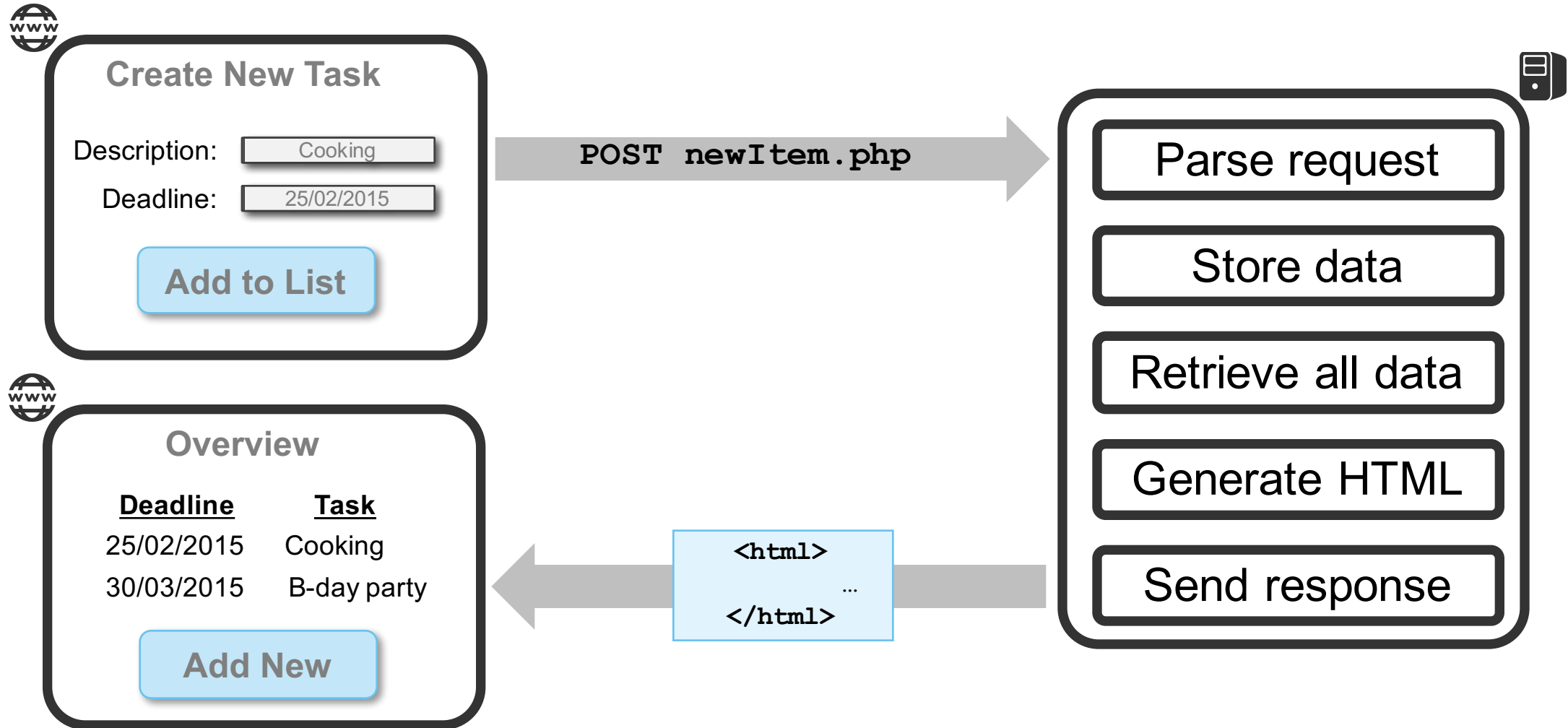
Securing Single Page Applications

Philippe De Ryck

 @PhilippeDeRyck  <https://www.websec.be>



Traditional Web Applications



Traditional Web Applications



Create New Task

Description:

Deadline:

Add to List



Overview

<u>Deadline</u>	<u>Task</u>
30/03/2015	B-day party
25/02/2015	Cooking

Add New



Parse request

Store data

Retrieve all data

Generate HTML

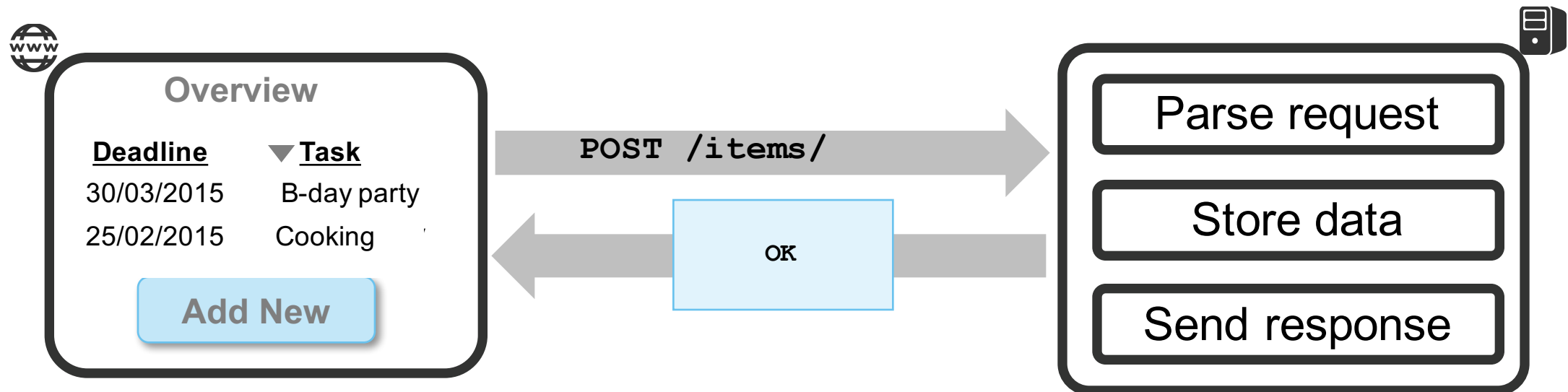
Send response

Sorting API

GET sortBy?col=Task

```
<table>
...
</table>
```

Single Page Applications



What's behind a Single Page Application



<https://items.example.com>

Create New Task

Description:

Deadline:

Add to List

Overview

<u>▼ Deadline</u>	<u>Task</u>
25/02/2015	Cooking
30/03/2015	B-day party

Show completed tasks

AngularJS routing

```
$routeProvider.when('/overview', {  
  templateUrl: 'overview.html',  
  controller: 'OverviewCtrl'  
}).  
$routeProvider.when('/completed',  
{  
  templateUrl: 'completed.html',  
  controller: 'CompletedCtrl'  
});
```



<https://items.example.com/#/completed>

What's behind a Single Page Application



<https://items.example.com>

```
<html ng-app>

  <div ng-controller="NewTaskCtrl">
    ...
  </div>

  <div ng-view>
  </div>

</html>
```

AngularJS controllers

```
myApp.controller('CompletedCtrl',
  ['$scope', function($scope) {
    $scope.completed = ...
  }]);
```

AngularJS templates

```
<h3>Completed Tasks</h3>
<ul>
  <li ng-repeat="task in completed">
    {{task.deadline}} {{task.descr}}
  </li>
</ul>
```

What's behind a Single Page Application

- The backend of an SPA has three general responsibilities
 - Serve static application files
 - Provide access to the business logic through an API
 - Persistent data storage
- Frontend and backend are completely decoupled
 - HTTP is the transport mechanism between both
 - RESTful API is a good match for this scenario
- Decoupled backend needs to stand on its own
 - Validate data
 - Enforce workflows

What's behind a Single Page Application

Route	HTTP Verb	Description
/api/bears	GET	Get all the bears.
/api/bears	POST	Create a bear.
/api/bears/:bear_id	GET	Get a single bear.
/api/bears/:bear_id	PUT	Update a bear with new info.
/api/bears/:bear_id	DELETE	Delete a bear.

What's behind a Single Page Application

- **Properties of a RESTful API**
 - Separation of concern between client and server
 - Stateless on the server-side
 - Clear caching decisions (yes or no)
 - Uniform interface
- **Many concrete implementations available**
 - Heavyweight enterprise frameworks (e.g. Java, .NET)
 - Lightweight JavaScript tools (e.g. NodeJS)
 - Even more lightweight, REST-enabled databases (e.g. CouchDB)

What's behind a Single Page Application

- Consuming a REST API using XHR

Classic XHR

```
var url = "http://.../api/bears/0"
var xhr = new XMLHttpRequest();
xhr.open("DELETE", url, true);
xhr.onreadystatechange = function () {
  if (xhr.readyState == 4) {
    if (xhr.status == 200) {
      //Bye bye bear 0
    }
  }
}
xhr.send();
```

AngularJS \$resource

```
var api = $resource
  ("http://.../api/bears/:id")

api.$delete({id: 0},
  function(v, h) { /* success */ },
  function(res) { /* error */ }
);
```

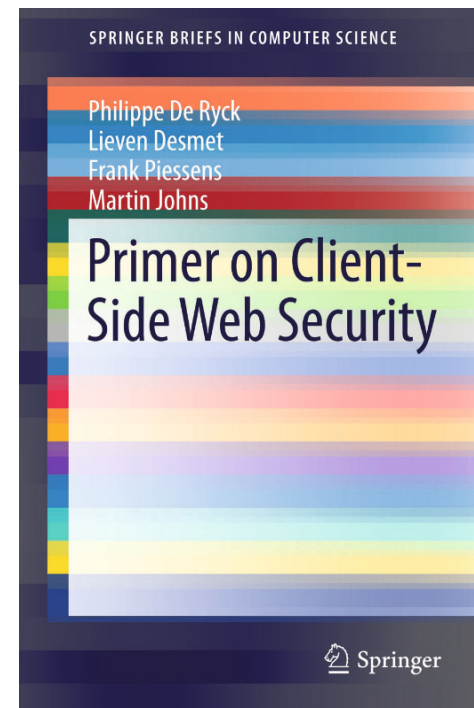
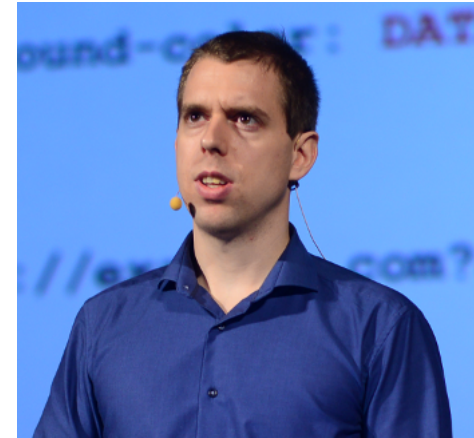
The Agenda for Today

- **Stateless Session Management**
 - Moving from server-side to client-side session management
 - Cookie-based sessions vs token-based sessions
- **Common Authorization Problems**
 - Cross-Site Request Forgery and Direct Object References
- **Cross-Site Scripting**
 - Dealing with XSS in client-side frameworks
- **Conclusion**

About Me – Philippe De Ryck

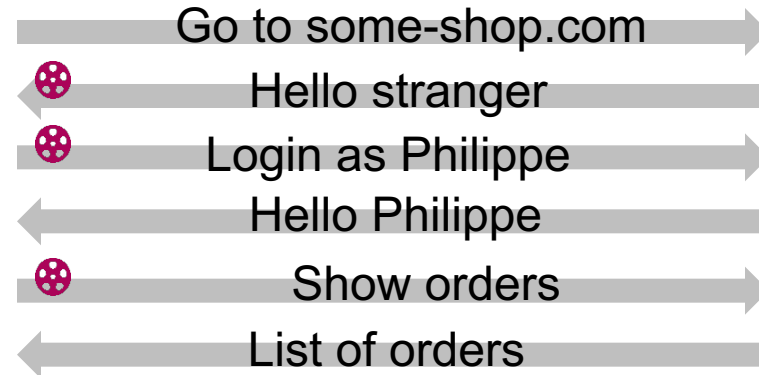
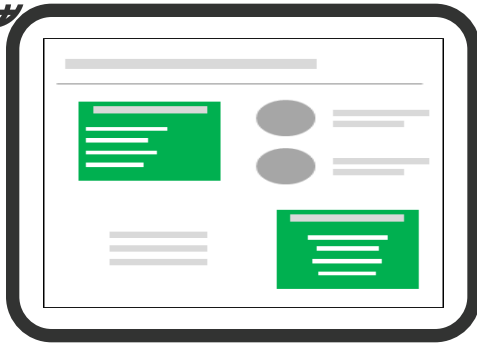
- Postdoctoral Researcher @ DistriNet (KU Leuven)
 - PhD on client-side Web security
 - Expert in the broad field of Web security
 - Main author of the *Primer on Client-Side Web Security*

- Running the Web Security training program
 - Dissemination of knowledge and research results
 - Public training courses and targeted in-house training
 - Target audiences include industry and researchers



Stateless Session Management

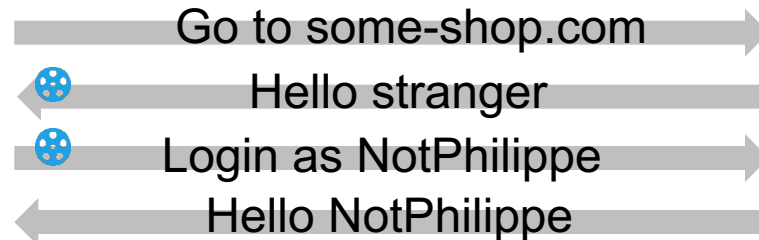
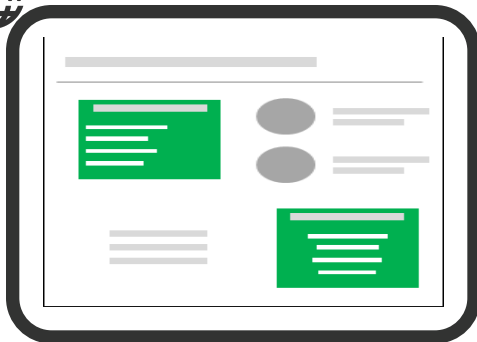
Using Cookies to Manage Sessions



Some-shop.com



```
🍪 3a99a4d1e8f496  
Logged_in: false  
User: Philippe  
Admin: true
```



```
🍪 2ad3e9f78bc808  
Logged_in: false  
User: NotPhilippe  
Admin: false
```

Properties of Cookie-Based Sessions

- **Session identifiers and objects are bearer tokens**
 - The token represents ownership of the session
- **Cookies are managed by the browser**
 - Stored automatically
 - Automatically attached to every request, if the domain matches
- **Common threats against cookie-based session management**
 - Brute forcing a session identifier
 - Session hijacking and session fixation
 - Cross-Site Request Forgery

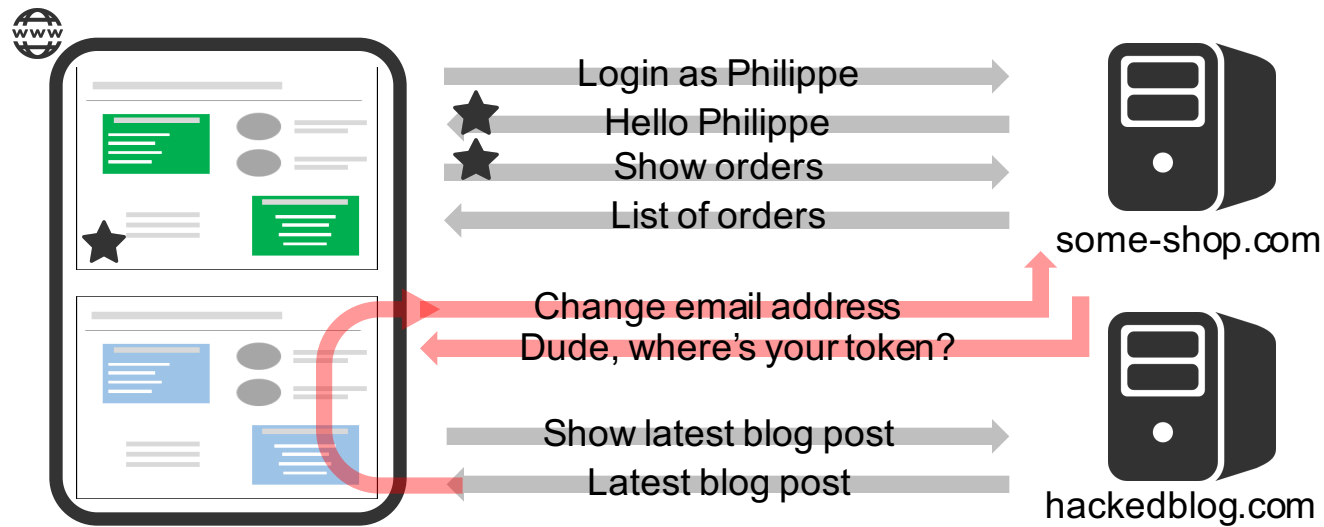
Moving Towards Stateless APIs

- **On the server**
 - Results in a stateful API
 - Gives the server full control over the session
 - Track active sessions, invalidate expired sessions
 - Requires the use of a session identifier (bearer token)
- **On the client**
 - Stateless API pushes all session information to the client
 - Server has no control over active sessions
 - Requires additional protection of the session data at the client

Client-Side Sessions

- A session object is sent back and forth
 - Contains data about the current session
 - Needs to be stored by the client
 - Needs to be sent to the server when needed
- Cookies are merely a medium, not a requirement
 - Cookies are stored automatically by a browser
 - Cookies are attached automatically by a browser
 - Still vulnerable to CSRF attacks
- Token-based approaches are an equally valid medium

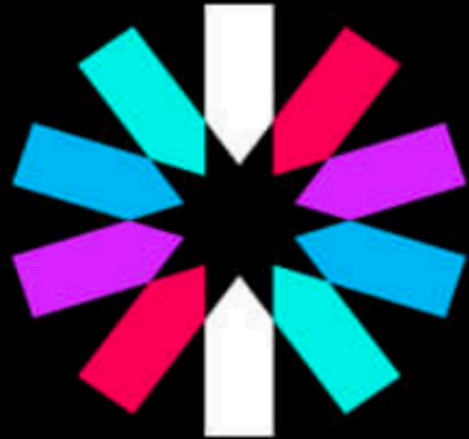
Tokens as an Alternative to Cookies



- Tokens are sent to the client
 - In an HTTP header or in the body of the response
 - The client-side application attaches them to outgoing requests

Client-Side Sessions with Tokens

- The session data is the value of the token
 - Sent to the client by any means
 - Often with the *Authorization* HTTP header, using *Bearer* as value
- Browsers do not handle tokens automatically
 - Client-side application will have to extract and store the token
 - Client-side application needs to attach the token to requests
 - Often taken care of by client-side application frameworks
 - Easier to share across domains than cookies
- Mitigates CSRF attacks by design



JWT

JSON Web Tokens are an open, industry standard **RFC 7519** method for representing claims securely between two parties.

JSON Web Token

- A JWT just looks like a blob of data
 - Contains three sections of base64-encoded data

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJkaXN0cm1uZXQuY3Mua3VsZXV2ZW4uYmUiLCJleHAiOjI0MjUwNzgwMDAwMDAsIm5hbWUiOiJwaGlscXBwZSIsImFkbWwIjp0cnVlfQ.dIi1OguZ7K3ADFnPOsmX2nEpF2Asq89g7GTuyQuN3so
```

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

Header

```
{  
  "iss": "distrinet.cs  
.kuleuven.be",  
  "exp": 1425078000000,  
  "name": "philippe",  
  "admin": true  
}
```

Payload

```
HMACSHA256(  
  base64UrlEncode(header)  
  + "." +  
  base64UrlEncode(payload),  
  "secret"  
)
```

Signature

JSON Web Token

- The standardized way to exchange session data
 - Part of a JSON-based Identity Protocol Suite
 - Together with specs for encryption, signatures and key exchange
 - Used by OpenID Connect, on top of OAuth 2.0
- Requires explicit handling by the client-side application
 - Difficult in traditional HTML Web applications
 - Easy with modern client-side JavaScript frameworks
- Easy to pass around between services

Client-Side Sessions with JWT

- JWT data is base64-encoded, so the client can read it
 - Use the information in the token to reflect access control in UI
 - E.g. hide or show admin features
- Tokens are sent in headers or URI parameters
 - No automatic browser involvement
 - Client-side application needs to extract and attach tokens
 - Session persistency needs to be implemented as well
- In modern JS frameworks, this is not a big deal

JWT In Practice

- JWT are base64-encoded JSON objects
 - By default, no confidentiality, so careful with sensitive data
 - Integrity is built in through the signature
- Widely supported, with libraries for almost every language
 - Well suited to exchange identity information between microservices
- JWT has gotten a bad reputation lately
 - Two implementation vulnerabilities in the libraries

Session Management Wrap Up

- Traditional server-side give the highest level of control
 - Session identifiers generally carried by cookies
 - Requires keeping state on the server
- Client-side sessions are more flexible
 - Server does not need to keep state, but has less control
 - Requires integrity and/or confidentiality checks to be in place
- Commonly accepted best practice in modern applications
 - Client-side sessions in a token-based approach
 - Standardized technology: JWT

Common Authorization Problems

The Impact of Client-side Sessions



- Server-side session object contains user information
 - Referred to with the long and unique session identifier
 - Sent in an HTTP-Only and Secure cookie over HTTPS

```
if(!req.session.Admin) {  
    throw new Error("No admin privileges")  
}  
else {  
    // Do admin stuff  
}
```

The Impact of Client-side Sessions



- Client-side session object contains user information
 - Unserialized into a session object at the server-side
 - Sent in an HTTP-Only and Secure cookie over HTTPS


```
if(!req.session.Admin) {  
    throw new Error("No admin privileges")  
}  
else {  
    // Do admin stuff  
}
```

Protecting Client-Side Session Data

- Session data is provided by the client
 - This data should be considered untrusted until verified
 - Protection can be offered using encryption or signatures
- Encryption offers confidentiality on the client
 - Recommended if sensitive data is stored inside the session object
- A signature offers integrity on the server
 - If a client tampered with the data, the server will be able to notice it

Client-Side Sessions with Cookies



 `session=TG9nZ2VkX2luOiB0cnVlClVzZXI6IFBoaWxpcHBlCkFkbWluOiB0cnVlCg==`

 `session-sig=7699bf4963dbec0e66a9d8e213dfe3c0ca07ee87`

- The session cookie is base64 encoded
 - This is no encryption, merely a transformation
- Signature is generated using a server secret and HMAC function
 - The client should never be able to generate a valid signature

Do Not Rely on Untrusted Data

- In a Web application, untrusted data is everywhere
 - Everything that comes from the client should not be trusted
- Common authorization vulnerabilities
 - Relying on cookie values without scrutiny
 - Relying on hidden form parameters
 - Using hidden paths
- Only trust data if you are sure it's not been tampered with
 - The knowledge of a random identifier or token
 - Verification of a signature of the data

2015-05-12

All Artisan State User-Uploaded Photos are Publicly Accessible

Update 2015-05-15: It appears this one specific issue has been fixed as of today.

Artisan State is a photo printing service that specializes in flush mount books and other photo books. They are managed from San Francisco with fulfillment in Hong Kong and Houston and manufacturing in Shanghai. They have a pretty website and reputation seems somewhere between iPhone-level travel books and professionally-bound books you would get with an in-person event photographer. I was preparing a book for one of my clients and as I am uploading the photos, which are personal, the first thought was... *should I really be uploading these photos to this website, we just met?*



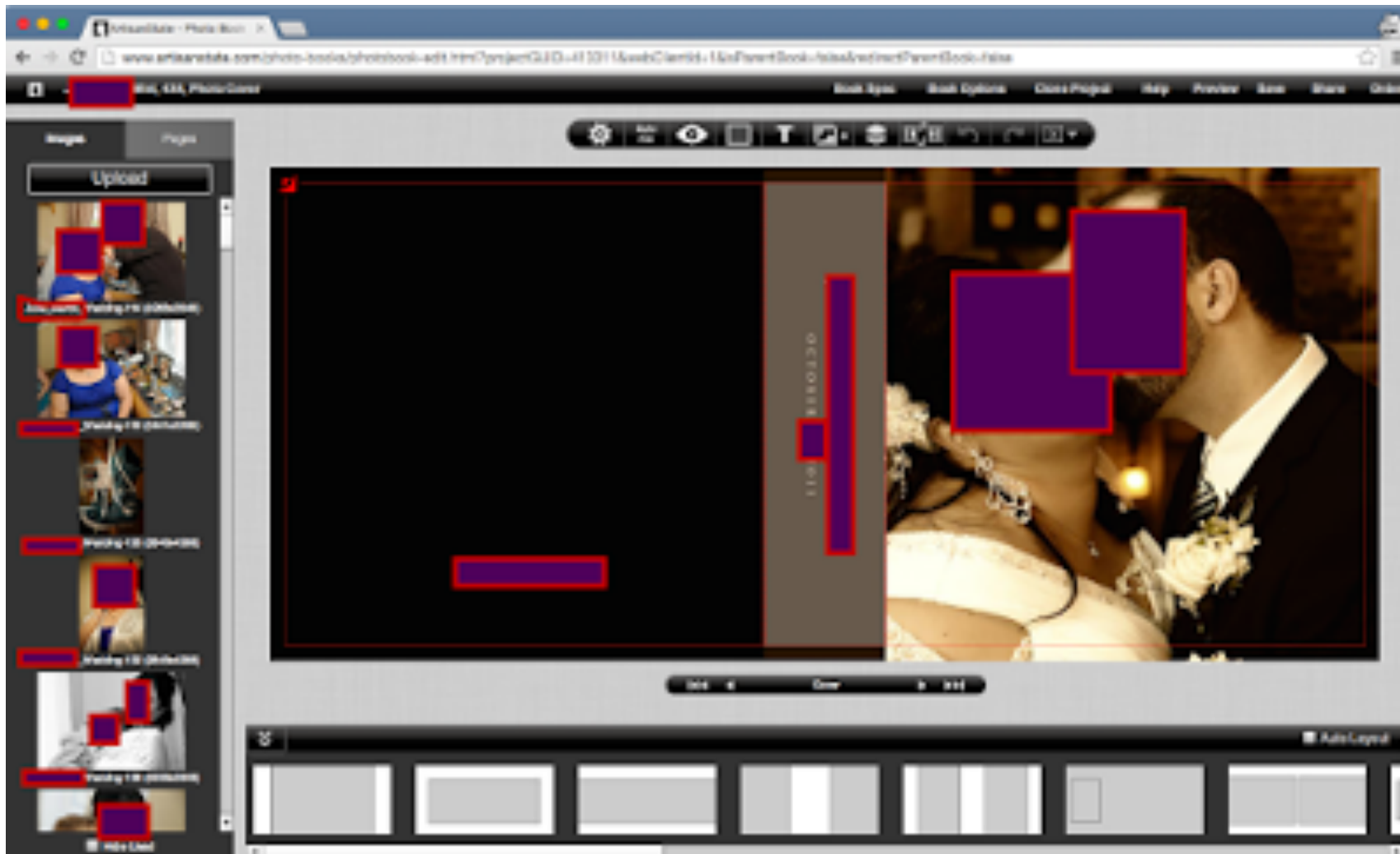
Direct Object References

<http://www.artisanstate.com/photo-books/photobook-edit.html?projectGUID=413312>



Direct Object References

<http://www.artisanstate.com/photo-books/photobook-edit.html?projectGUID=413311>



<http://privacylog.blogspot.be/2015/05/all-artisan-state-user-uploaded-photos.html>

Citigroup hack exploited easy-to-detect web flaw

Brute force attack exposes 200,000 accounts

14 Jun 2011 at 21:25, [Dan Goodin](#)



106



30



39

Hackers who stole bank account details for 200,000 Citigroup customers infiltrated the company's system by exploiting a garden-variety security hole in the company's website for credit card users, according to a report citing an unnamed security investigator.

The New York Times reported that the technique allowed the hackers to leapfrog from account to account on the Citi website by changing the numbers in the URLs that appeared after customers had entered valid usernames and passwords. The hackers wrote a script that automatically repeated the exercise tens of thousands of times, the *NYT* said in [an article](#) published Monday.

“Think of it as a mansion with a high-tech security system – that the front door wasn't locked tight,” reporters Nelson D. Schwartz and Eric Dash wrote.

The underlying vulnerability, known as an [insecure direct object reference](#), is so common that it's included in the [Top 10 Risks list](#) compiled by the Open Web Application Security Project. It results when developers expose direct references to confidential account numbers instead of using substitute characters to ensure the account numbers are kept private.

DOR Are Very Common in REST APIs

Route	HTTP Verb	Description
/api/bears	GET	Get all the bears.
/api/bears	POST	Create a bear.
/api/bears/:bear_id	GET	Get a single bear.
/api/bears/:bear_id	PUT	Update a bear with new info.
/api/bears/:bear_id	DELETE	Delete a bear.

- API relies on the use of IDs
 - Data storage also relies on the use of IDs
 - Common to simply use the object ID as the REST identifier
 - Easy to forget access control checks, leading to vulnerabilities

ObjectIDs in MongoDB

- MongoDB uses 12-byte object identifiers

```
> ObjectId()  
ObjectId("56daed0d9a0d54b9c7fe354a")
```

- But they are actually sequential identifiers
 - Beware if you use these IDs directly in your application
 - Make sure you perform server-side access control checks!

```
> ObjectId()  
ObjectId("56daed0d9a0d54b9c7fe354b")  
> ObjectId()  
ObjectId("56daed0d9a0d54b9c7fe354c")  
> ObjectId()  
ObjectId("56daed0d9a0d54b9c7fe354d")
```

Direct Object References

- Explicitly listed in the OWASP top 10 as a vulnerability
 - Very natural mistake to make
 - You will not spot this unless someone tells you about it
- Explicitly check ownership/permissions for each request
 - Use the user identifier from the session data to compare with
- Alternatively, avoid the use of direct object references
 - Filter out all objects that the user should not access
 - Use indirect identifiers that refer to this list, and translate to DOR

Best Practices

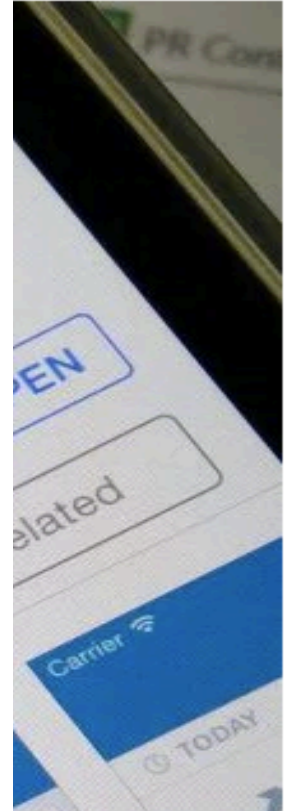
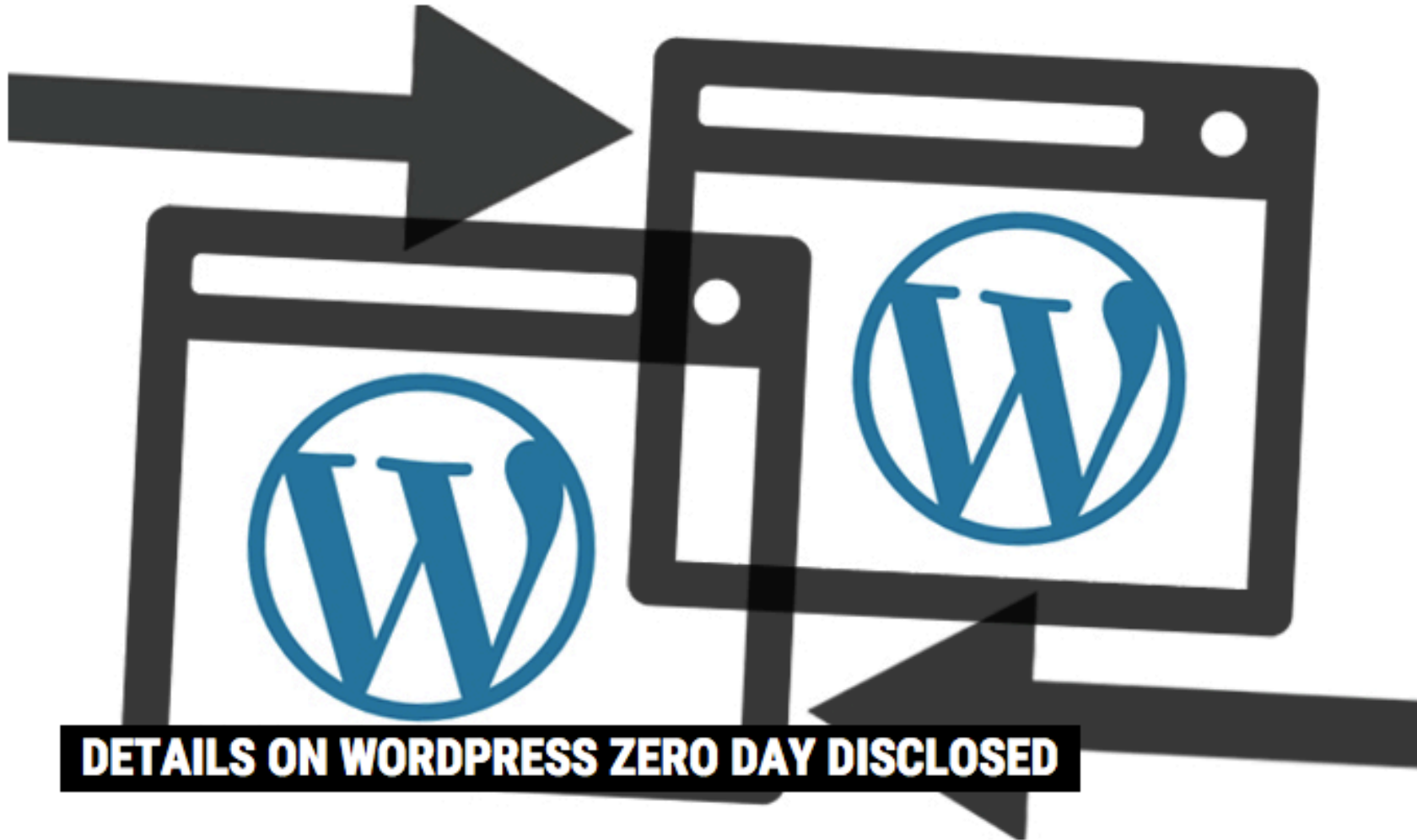
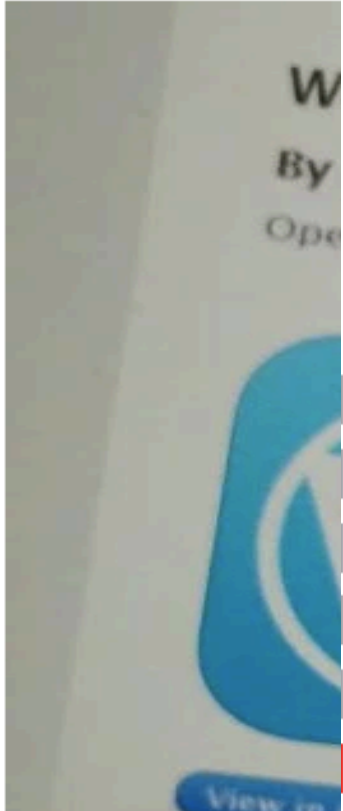
- Do not make security decisions with untrusted data
 - Treat all input as untrusted, unless you have verified the integrity
 - Signed session data is well-suited for this purpose
- Always check ownership/permissions before granting access
 - Use user information from a trusted session object
 - Perform these checks **everywhere**, even for “hidden” functions
- Never ever make security decisions on the client-side
 - Make them in the backend, and test the backend for security
 - Optionally use them at the client-side to improve the UI

Cross-Site Scripting

WordPress now powers 25% of the Web

EMIL PROTALINSKI N

TAGS: AUTOMATTIC, TO



<https://threatpost.com/details-on-wordpress-zero-day-disclosed/112435/>
<http://venturebeat.com/2015/11/08/wordpress-now-powers-25-of-the-web/>

The Samy Worm

Firefox

marks Tools Help

http://mail.myspace.com/index.cfm?fuseaction=mail.friendRequests&Mytoken=[redacted]

MySpace.com | Home The Web MySpace Search Help | SignOut

classmates-com

I graduated in: State: [MD] Year: [90] GO! Springfield High (1084) Martin Luther King High (676) Trinity High School (328) NEW YORK High School (820)

Home | Browse | Search | Invite | Rank | Mail | Blog | Favorites | Forum | Groups | Events | Games | Music | Classifieds

Mail Center Friend Request Manager Approve or Deny Your Friend Requests Here [help]

Inbox Saved Sent Trash Bulletin Friend Requests Pending Requests Event Invites

Fly Fishing Trip in Mexico All inclusive package in Ascension Bay, Mexico, from US\$1,600... www.pescamaya.com




Yellow Dog Flyfishing Adventures Specializing in destination angling packages throughout the U... www.yellowdogfl...

Elk River Guiding Company - Fernie, BC Fly fish the Elk River in the Canadian Rockies.

KICK ASS I RULE

PLEASE DONT PRESS CHARGES MAD PHOTOSHOP SKILLS SHE WANTS ME

Listing 1-10 of 919664 1 2 3 4 5 >> of 91967 Next >

Date:	From:	Confirmation:
Oct 4, 2005 10:22 PM	 Online Now!	PLEASE DONT PRESS CHARGES Lulu the Loveable Freak wants to be your friend! Approve Deny Send Message
Oct 4, 2005 10:21 PM		AlysOn!! wants to be your friend! Approve Deny Send Message
Oct 4, 2005 10:20 PM	 Online Now!	Erika wants to be your friend! Approve Deny Send Message

http://samy.pl/popular/

XSS Vulnerabilities Make You Money

How I got a \$3,500 USD Facebook Bug Bounty

I recently found a Stored XSS on Facebook, which resulted in a Bug Bounty Reward. If you want to know how an XSS could be exploited, you can read my colleague Mathias' [blog post about it](#). Anyway, here's how it went down.

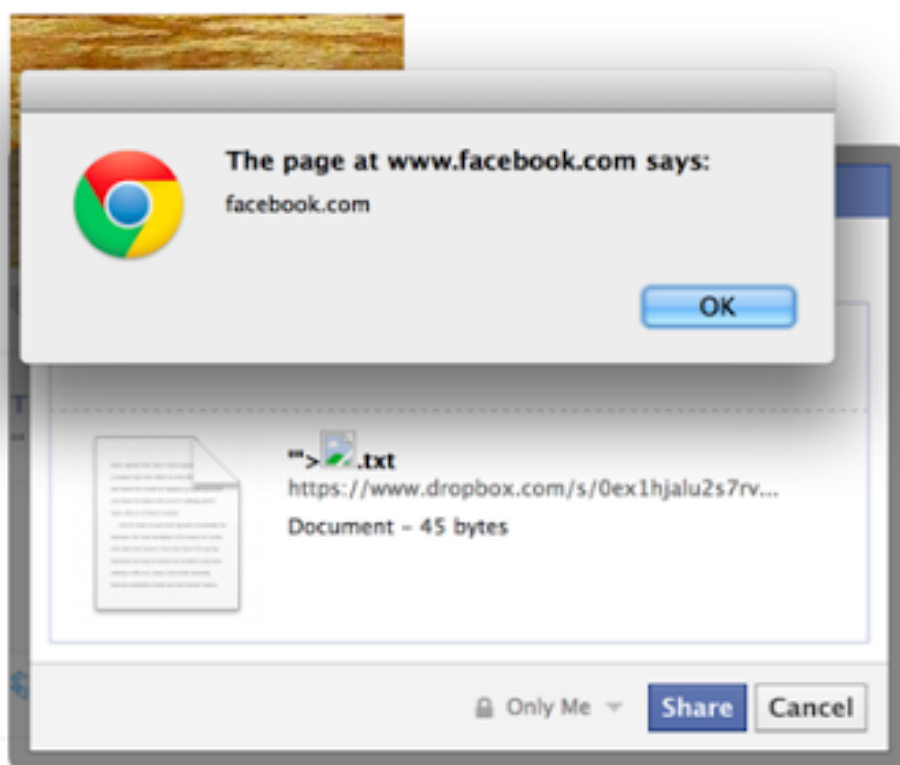
I was actually working on finding flaws on Dropbox to begin with. I noticed that when using their web interface there were some restrictions on what filenames that were allowed. If you tried to rename a file to for example:

```
' "><img src=x onerror=alert(document.domain)>.txt
```

it was not possible. You got this error:

The following characters are not allowed: \ / : ? * < > " |

onerror=alert(/xss/)>



Ted Rosén updated a file on [Dropbox](#).

">



"> <img src=x
onerror=alert(document.domain)>.jpg
Image - 9.97 KB

👍 Like · 💬 Comment · 🔄 Unfollow Post · ➦ Share · 15 minutes ago



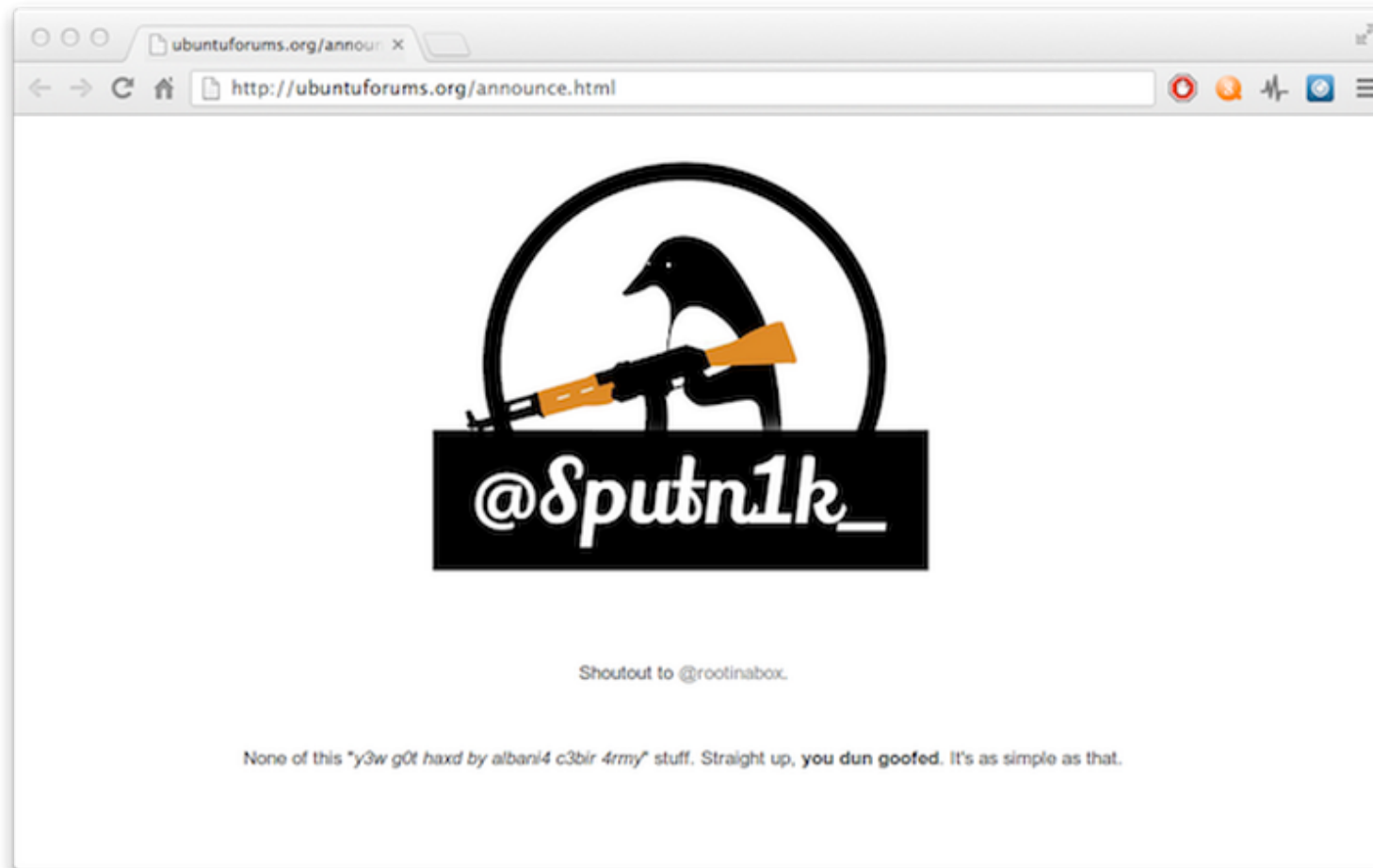
Ted Rosén created the group.

Like · 💬 Comment · 🔄 Unfollow Post · 16 minutes ago



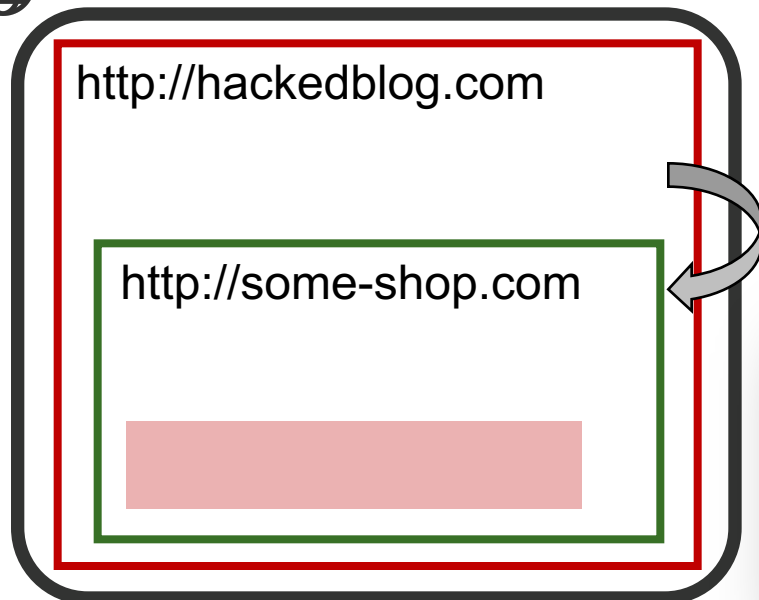
Ubuntu forums hacked; 1.82M logins, email addresses stolen

Canonical, the company behind the Ubuntu operating system, has suffered a massive data breach on its forums. All usernames, passwords, and email addresses were stolen.

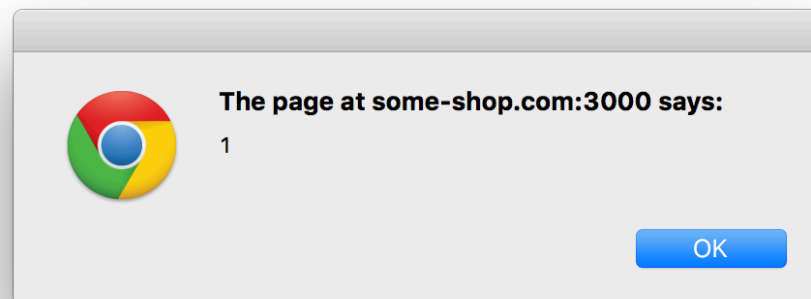


Cross-Site Scripting

- Started as code injection from one site into another
 - Hence the *cross-site* and *scripting*
 - Introduced by Microsoft in 2000

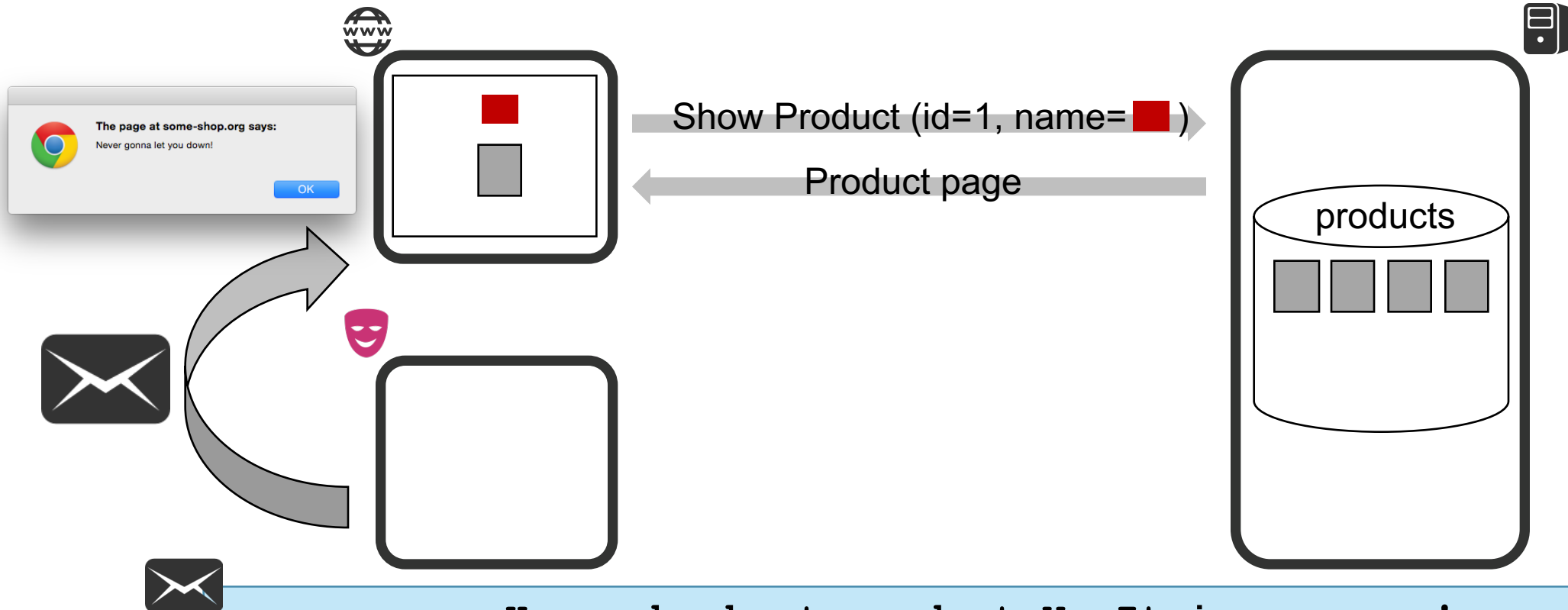


Load iframe from `http://some-shop.com?prod=5&color=blue<script>alert(1)</script>`



Reflected XSS

```
<html><body><h1> ■ </h1><p> □ </p></body></html>
```

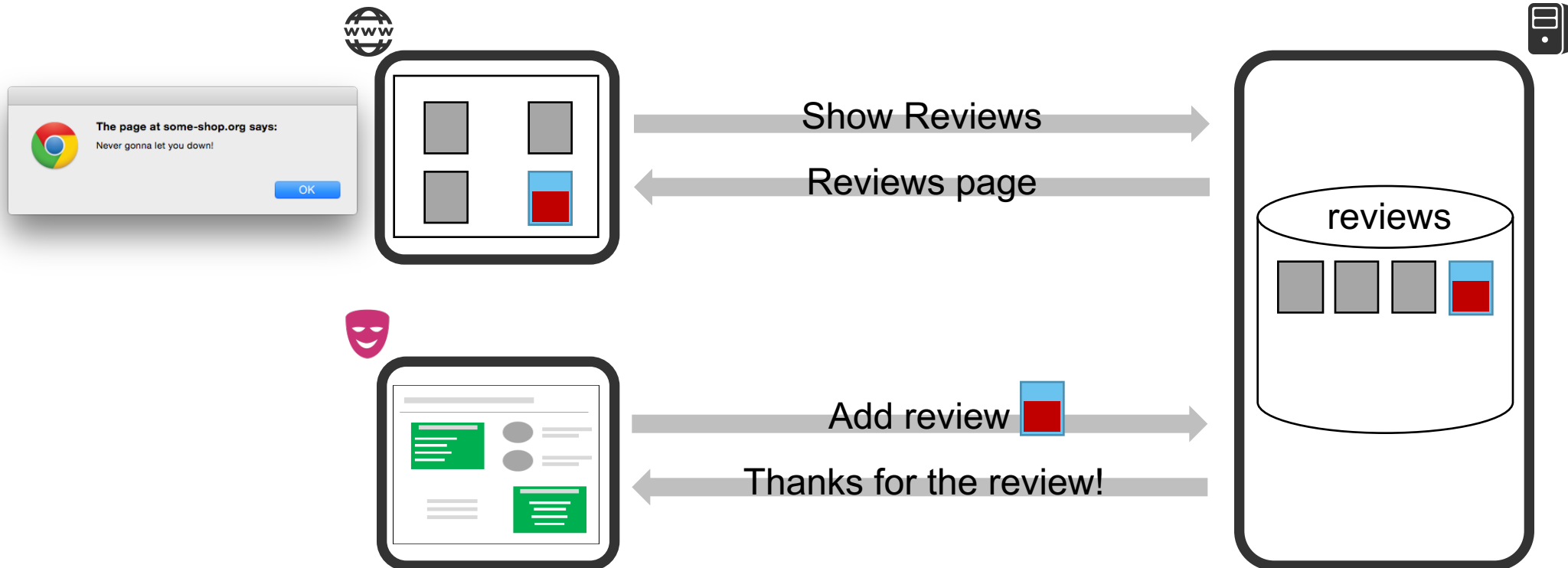


Hey, checkout product X. It is awesome!

```
<a href="http://some-shop.org/product.php?id=1&name=  
<script>alert('Never gonna let you down!')</script>
```


Stored XSS

```
<html><body>...   ...</body></html>
```

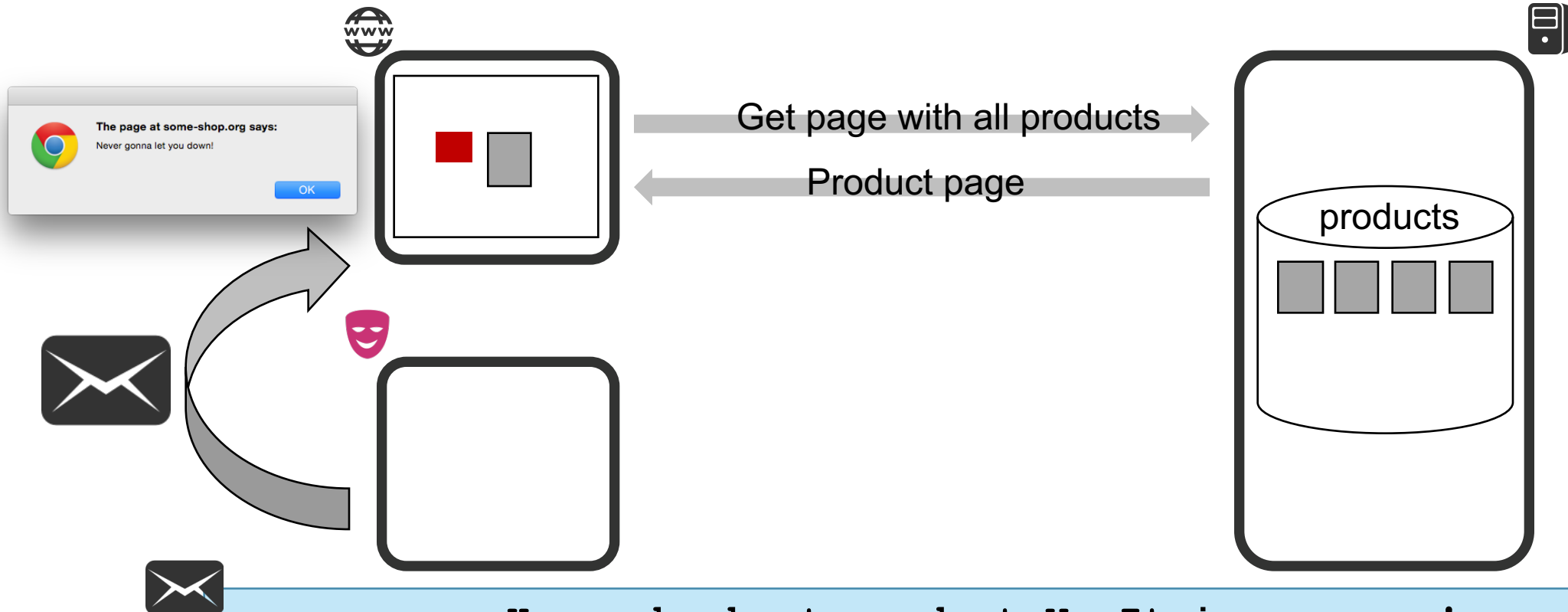


I can really recommend product X. It is awesome!
`<script>alert('Never gonna let you down!')</script>`

DOM-Based XSS

```
permalink.innerHTML = '<a href="" +  
window.location.hash.split('#')[1] +  
' .php">Link to product</a>';
```

```
<html><body><p>  
  █ █ █ █  
</p></body></html>
```



```
Hey, checkout product X. It is awesome!  
<a href="http://some-shop.org/allproducts.php#1  
<script>alert('Never gonna let you down!')</script>
```

The Truth about DOM-Based XSS

- DOM-based XSS is essentially XSS in a JS context
 - But the injection happens at the client-side, in the browser
 - Therefore, the server can **not** protect against DOM-based XSS
- It can be reflected, persistent or neither
 - If embedded in the fragment identifier, it stays within the browser
 - The server will never see the payload
- It's often considered exotic, and not a real risk
 - This is just really ignorant of DOM-based XSS

What Can an Attacker Do With XSS

- Anything within the context of the application
 - Modify the DOM
 - Read the page contents
 - Extract username/password from a form
 - Steal autocomplete values
 - Use geolocation/webcam/... permissions
 - Read cookies to do session hijacking attacks
 - Run a JavaScript-based port scanner
- Elevate privileges through the XSS attack
 - By getting hold of an administrator account

Apache.org Compromise

1. Report bug with obscured URL containing reflected XSS attack
`http://tinyurl.com/XXXXXXXX`

2. Admin opens link, compromising their session

3. Attacker disable notifications for a hosted project

4. Attacker changes upload path to location that can execute JSP files

5. Attacker added new bug reports with JSP attachments

6. Attacker browses and copies filesystem through JSP. Installs backdoor JSP with webserver privileges

Apache.org Compromise

7. Attacker installs JAR to collect passwords on login

8. Triggered logins by sending out password reset mails

9. One of the passwords matched an SSH account with full sudo access

10. The accessible machine had user home folders, with cached subversion credentials

11. From the subversion machine, privilege escalation was unsuccessful

XSS Is a Stupid Problem to Have

- XSS is an injection vulnerability
 - Boils down to confusion between data and code
 - Untrusted data is interpreted as application-provided code
- Other injection vulnerabilities suffer from the same problems
 - SQL injection, Command injection, ...
- Solution is simple: Separate data from code

PREPARED STATEMENTS IN SQL

```
query = "SELECT * FROM users WHERE login='" + login + "'";
```

```
query = "SELECT * FROM users WHERE login=:user;"
```

```
data = { "user": login }
```

Separating Data and Code on the Web

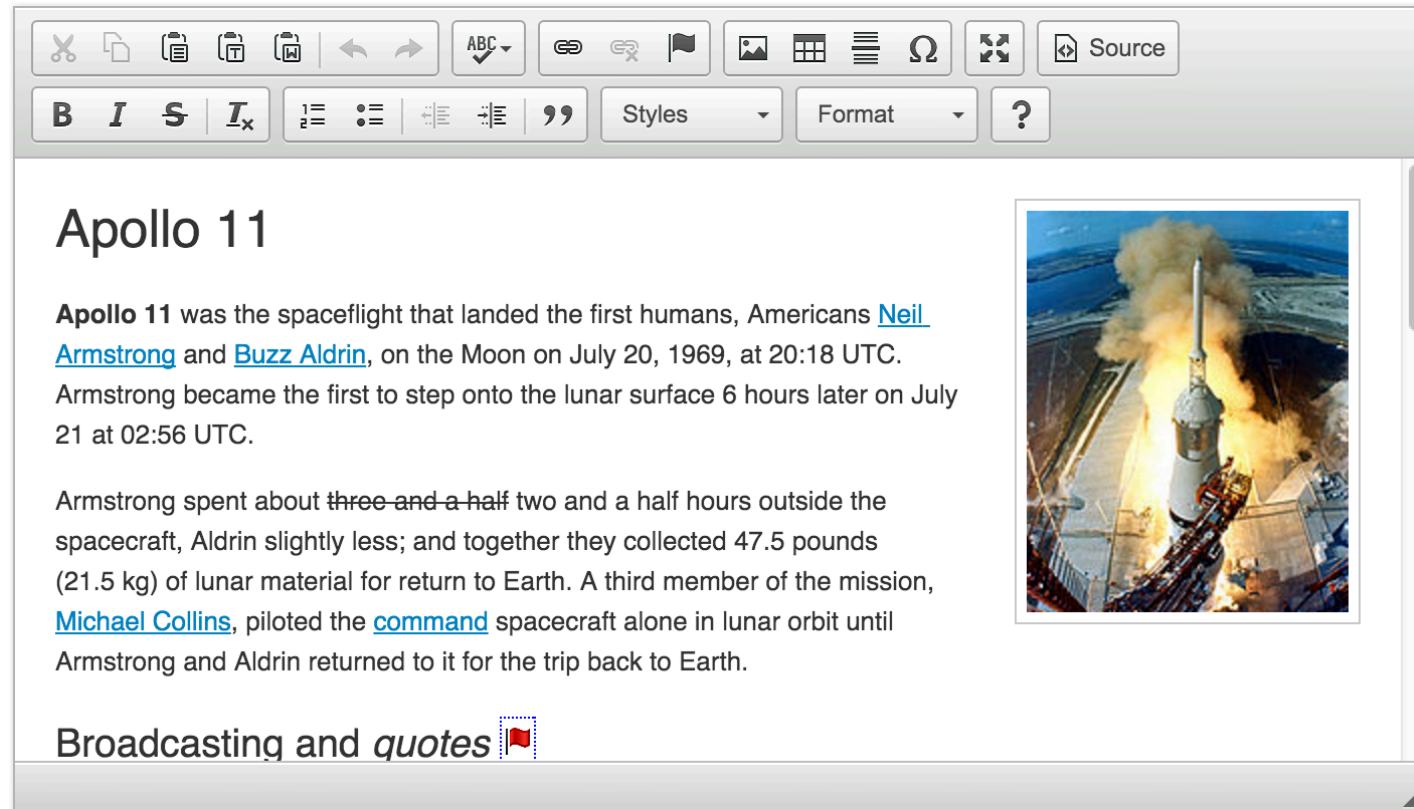
- This is virtually impossible to achieve
 - Code is all HTML, JS, CSS, ... content
 - Data is all the rest, which can be trusted or untrusted
- Server-side composition versus client-side processing
 - The server knows what is data and what is code
 - But the browser only receives one HTML page
- Design flaw that has caused a lot of grief


Context-Sensitive Output Encoding

- There are many different contexts in an HTML page
 - HTML body `<h1>DATA</h1>`
 - HTML attributes `<div id='DATA'>`
 - Stylesheet context `body { background-color: DATA; }`
 - Script context `alert("DATA");`
 - URL context ``
- Each context has specific encoding needs
 - E.g. translating `< > & ...` to `< > & ...`
 - E.g. Escaping `"` and `'`

But Sometimes, Encoding Is Not an Option

- Many sites allow the use of HTML in user-provided data
 - Image inclusions in forums
 - Styles from WYSIWYG editors



The screenshot shows a WYSIWYG editor interface. The top toolbar includes icons for undo, redo, bold, italic, strikethrough, bulleted list, numbered list, link, unlink, flag, image, table, text color, background color, and source. Below the toolbar are buttons for bold (B), italic (I), strikethrough (S), and strikethrough with underline (I_x), followed by list icons, a quote icon, and dropdown menus for Styles and Format. The main content area displays a post titled "Apollo 11". The text reads: "Apollo 11 was the spaceflight that landed the first humans, Americans [Neil Armstrong](#) and [Buzz Aldrin](#), on the Moon on July 20, 1969, at 20:18 UTC. Armstrong became the first to step onto the lunar surface 6 hours later on July 21 at 02:56 UTC." Below this, it says: "Armstrong spent about ~~three and a half~~ two and a half hours outside the spacecraft, Aldrin slightly less; and together they collected 47.5 pounds (21.5 kg) of lunar material for return to Earth. A third member of the mission, [Michael Collins](#), piloted the [command](#) spacecraft alone in lunar orbit until Armstrong and Aldrin returned to it for the trip back to Earth." To the right of the text is an image of the Apollo 11 rocket launching. At the bottom of the editor, the text "Broadcasting and *quotes* " is visible.

HTML Sanitization

- We can do that with a regular expression!

- Let's go ask  stackoverflow

Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML.

Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp.

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The <center> cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the transgression of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a breach between this world and the dread realm of corrupt entities (like SGML entities, but *more corrupt*) a mere glimpse of the world of regex parsers for HTML will instantly transport a programmer's consciousness into a world of ceaseless screaming, he comes, the pestilent slithy regex-infection will devour your HTML parser, application and existence for all time like Visual Basic only worse he comes he comes do not fight he comes, his unholy radiance destroying all enlightenment, HTML tags leaking from your eyes like liquid pain, the song of regular expression parsing will extinguish the voices of mortal man from the sphere I can see it can you see if it is beautiful the final snuffing of the lies of Man ALL IS LOST ALL IS LOST the pony he comes he comes he comes the ichor permeates all MY FACE MY FACE oh god no NO NOOOO NO stop the angles are not real ZALGO IS TONY THE PONY, HE COMES

Proper HTML Sanitization

- First, you need to parse the HTML the proper way
 - Most languages offer a parser or have parsing libraries available
- You can filter dangerous content from the parsed HTML
 - Explicitly whitelist allowed elements and strip the rest
 - Do not forget about attributes (e.g. JS event handlers)
 - Make sure you're up to date with the latest specs (e.g. HTML5)
- Best solution: use a well-vetted sanitization library

Can JS MVC Frameworks Help?



Create New Task

Description:

Deadline:

Add to List



Overview

<u>Deadline</u>	<u>Task</u>
30/03/2015	B-day party
25/02/2015	Cooking

Add New



Parse request

Store data

Retrieve all data

Generate HTML

Send response

Sorting API

GET /tasks?sortBy=name

[{...}, {...}]

Server-Side Template Composition

- JavaScript MVC frameworks change how the DOM works
 - Extensions through elements, attributes, etc.
 - New interfaces
 - Often in combination with templating



EXTENDING THE DOM

```
<graph class="visitor-graph">
  <axis position="left"></axis>
  <axis position="bottom"></axis>
  <line name="typical-week" line-data="model.series.typicalWeek"></line>
  <line name="this-week" line-data="model.series.thisWeek"></line>
  <line name="last-week" line-data="model.series.lastWeek"></line>
</graph>
```

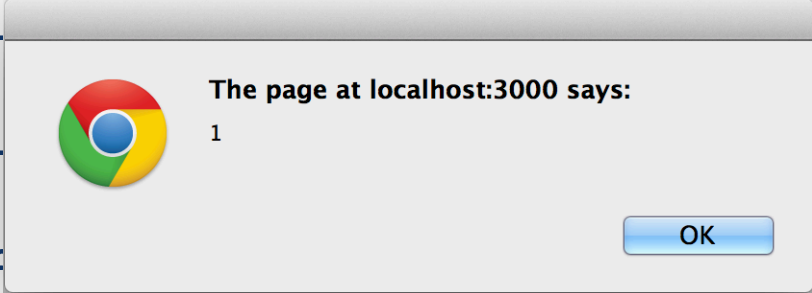
Server-Side Template Composition

- Traditional Web applications are based on HTML pages
 - They often integrate a JS MVC framework to improve the UI
 - E.g. Embedding AngularJS in dynamically constructed JSP pages
 - Server applies context-aware XSS protection



KNOCKOUT.JS EXAMPLE

```
<scri  
<div  
<scri  
</scr
```

A Chrome browser alert dialog box is overlaid on the code. It has a white background and a grey border. On the left is the Chrome logo. The text inside reads "The page at localhost:3000 says:" followed by the number "1" on the next line. At the bottom right is a blue button with the text "OK".

```
/script>
```

Mustache Security



Mustache Security

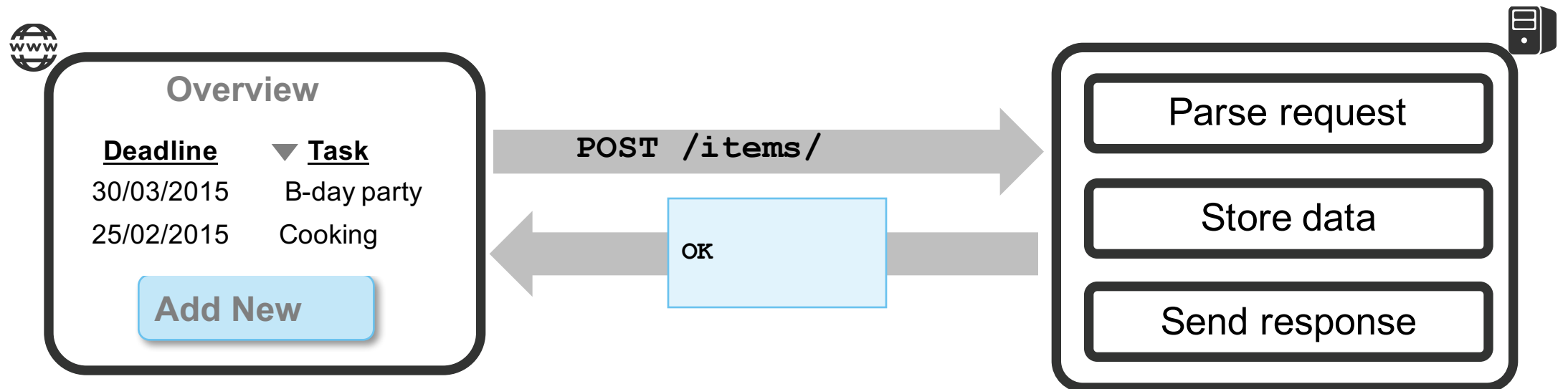
- Project dedicated to JS MVC security pitfalls
 - Assuming there is an injection vector
 - Assuming there is conventional XSS filtering in place
 - What can an attacker do?

- New behavior often breaks existing security assumptions
 - Bypass currently used security mechanisms
 - Script injection possible whenever a data attribute is allowed

Separating Front End and Back End

- Beware of server-side composition of templates
 - Generally a bad idea, because of dynamic behavior
 - If you must do this, AngularJS 1.2+ enforces quite a good sandbox
- Separating the front end from the back end
 - Server provides client-side application as static files
 - Server offers data through a well-designed API
 - Client-side application contains the dynamic behavior

Single Page Applications



Single Page Applications

- Run on a client-side JavaScript MVC framework
 - Backed by a data-driven REST API
- Back end has no context knowledge
 - So can also not provide useful input filtering and output encoding
 - Client-side application will have to take care of this
- So how does this work in AngularJS?

Example Case – User-Provided Images



ANGULARJS TEMPLATE

```
<textarea ng-model="x"></textarea>  
<div>{{x}}</div>
```

USER INPUT

```

```



RENDERED HTML

```

```

Example Case – User-Provided Images



ANGULARJS TEMPLATE

```
<textarea ng-model="x"></textarea>  
<div ng-bind="x"></div>
```

USER INPUT

```

```



RENDERED HTML

```

```

Example Case – User-Provided Images



ANGULARJS TEMPLATE

```
<textarea ng-model="x"></textarea>  
<div ng-bind-html="x"></div>
```

USER INPUT

```

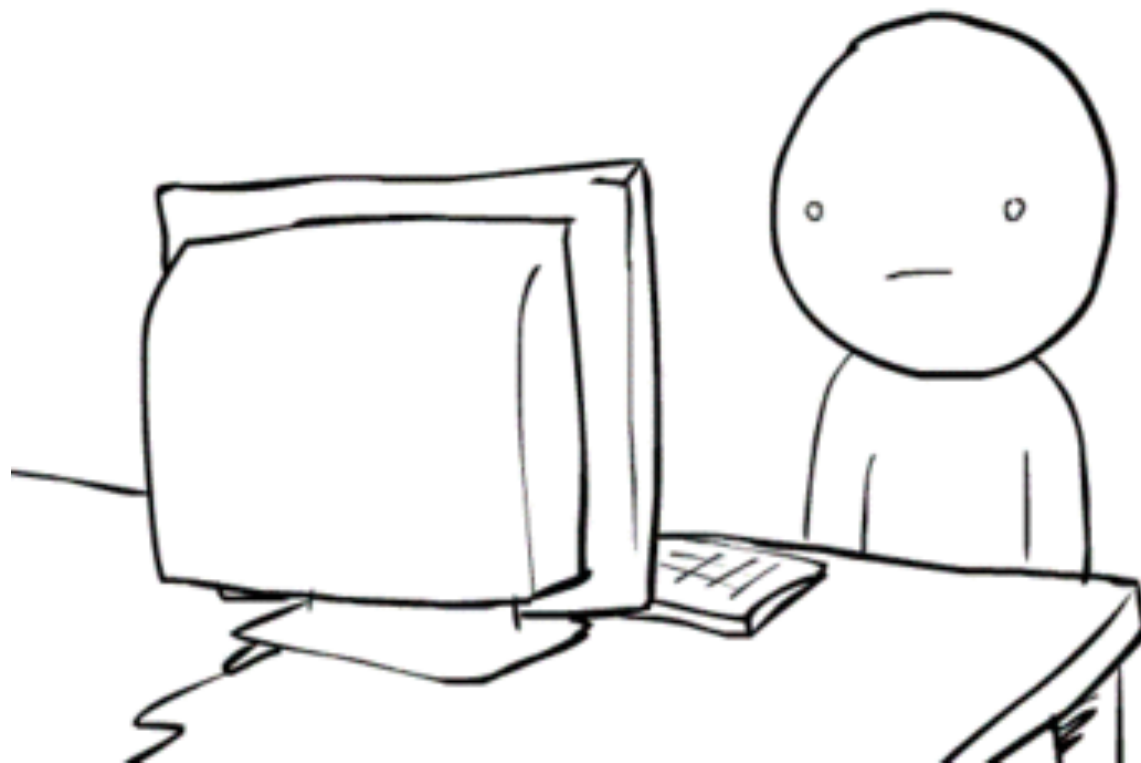
```



RENDERED HTML

Error: [\$sce:unsafe] Attempting to use an unsafe value in a safe context.

Dammit



Check Documentation

Error: \$sce:unsafe Require a safe/trusted value

 [Improve this Doc](#)

```
Attempting to use an unsafe value in a safe context.
```

Description

The value provided for use in a specific context was not found to be safe/trusted for use.

Angular's [Strict Contextual Escaping \(SCE\)](#) mode (enabled by default), requires bindings in certain contexts to result in a value that is trusted as safe for use in such a context. (e.g. loading an Angular template from a URL requires that the URL is one considered safe for loading resources.)

This helps prevent XSS and other security issues. Read more at [Strict Contextual Escaping \(SCE\)](#)

You may want to include the ngSanitize module to use the automatic sanitizing.

Go to StackOverflow



And you Find This Little Gem



You can also create a filter like so:

59



```
var app = angular.module("demoApp", ['ngResource']);  
  
app.filter("sanitize", ['$sce', function($sce) {  
  return function(htmlCode){  
    return $sce.trustAsHtml(htmlCode);  
  }  
}]);
```

Then in the view

```
<div ng-bind-html="whatever_needs_to_be_sanitized | sanitize"></div>
```

share improve this answer

answered Aug 26 '14 at 18:52



[Katie Astrauskas](#)

591 • 2 • 2

3 Fantastic! And this answer is cleaner and more angular-esk than the others. – [snumpy](#) Sep 1 '14 at 18:12

5 Gorgeous. Thank you. This is the correct answer. – [blrbr](#) Sep 22 '14 at 20:29

Awesome thanks! – [Mitch Glenn](#) Sep 22 '14 at 23:18

Beautiful solution! Thank you! – [the_critic](#) Dec 29 '14 at 21:36

@Katie Astrauskas, thank you for the anwer! Very clean way. BTW `ngResource` dependency is not necessary. – [Athlan](#) Mar 7 at 12:09

Example Case – User-Provided Images



ANGULARJS TEMPLATE

```
<textarea ng-model="x"></textarea>  
<div ng-bind-html="x | sanitize"></div>
```

USER INPUT

```

```



RENDERED HTML





Example Case – User-Provided Images



ANGULARJS TEMPLATE

```
<textarea ng-model="x"></textarea>  
<div ng-bind-html="x | sanitize"></div>
```

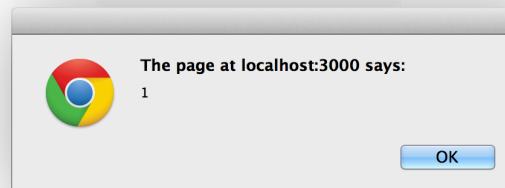
USER INPUT

```

```



RENDERED HTML



How Did That Happen?

Error: \$sce:unsafe Require a safe/trusted value

 [Improve this Doc](#)

```
Attempting to use an unsafe value in a safe context.
```

Description

The value provided for use in a specific context was not found to be safe/trusted for use.

Angular's [Strict Contextual Escaping \(SCE\)](#) mode (enabled by default), requires bindings in certain contexts to result in a value that is trusted as safe for use in such a context. (e.g. loading an Angular template from a URL requires that the URL is one considered safe for loading resources.)

This helps prevent XSS and other security issues. Read more at [Strict Contextual Escaping \(SCE\)](#)

You may want to include the ngSanitize module to use the automatic sanitizing.

Strict Contextual Escaping

- AngularJS tries to protect you from injection attacks
 - Let it, it's really good at it!
- *ng-bind* will never produce HTML



ANGULARJS TEMPLATE

```
<textarea ng-model="x"></textarea>  
<div ng-bind="x"></div>
```



GENERATED HTML

```
<div ng-bind="x">  
  &lt;img src="http://some-shop.com/coolcar.png"  
    onerror="alert(1)" /&gt;  
</div>
```

Strict Contextual Escaping

- AngularJS tries to protect you from injection attacks
 - Let it, it's really good at it!
- *ng-bind-html* can produce HTML, but not without protection



GENERATED HTML

Error: [\$sce:unsafe] Attempting to use an unsafe value in a safe context.



ANGULARJS TEMPLATE

```
<textarea ng-model="x"></textarea>  
<div ng-bind-html="x"></div>
```


Strict Contextual Escaping

- AngularJS tries to protect you from injection attacks
 - Let it, it's really good at it!
- *ng-bind-html* can produce HTML, but not without protection
 - Enable automatic sanitization with *ngSanitize*
 - Removes dangerous features from content

Example Case – User-Provided Images



ANGULARJS TEMPLATE

```
<textarea ng-model="x"></textarea>  
<div ng-bind-html="x"></div>
```



ANGULARJS CODE

```
angular.module("test", ["ngSanitize"])..
```

USER INPUT

```

```



RENDERED HTML



```

```

Strict Contextual Escaping

- AngularJS tries to protect you from injection attacks
 - Let it, it's really good at it!
- *ng-bind-html* can produce HTML, but not without protection
 - Enable automatic sanitization with *ngSanitize*
 - Removes dangerous features from content
- If you really really want raw **trusted** HTML ...
 - `$sce.trustAsHtml()` marks a string as trusted, disabling sanitization

Strict Contextual Escaping - trustAsHtml

▲ You can also create a filter like so:

59



```
var app = angular.module("demoApp", ['ngResource']);  
  
app.filter("sanitize", ['$sce', function($sce) {  
  return function(htmlCode){  
    return $sce.trustAsHtml(htmlCode);  
  }  
}]);
```

Then in the view

```
<div ng-bind-html="whatever_needs_to_be_sanitized | sanitize"></div>
```

share improve this answer

answered Aug 26 '14 at 18:52



Katie Astrauskas

591 • 2 • 2

3 Fantastic! And this answer is cleaner and more angular-esk than the others. – [snumpy](#) Sep 1 '14 at 18:12

5 Gorgeous. Thank you. This is the correct answer. – [blrbr](#) Sep 22 '14 at 20:29

Awesome thanks! – [Mitch Glenn](#) Sep 22 '14 at 23:18

Beautiful solution! Thank you! – [the_critic](#) Dec 29 '14 at 21:36

@Katie Astrauskas, thank you for the anwer! Very clean way. BTW `ngResource` dependency is not necessary. – [Athlan](#) Mar 7 at 12:09

Strict Contextual Escaping - trustAsHtml



ANGULARJS TEMPLATE

```
<textarea ng-model="x"></textarea>  
<div ng-bind-html="x | i_really_know_my_security"></div>
```



ANGULARJS CODE

```
angular.module("test", [])  
  .filter("i_really_know_my_security",  
    ['$sce', function($sce) {  
      return function(htmlCode) {  
        return $sce.trustAsHtml(htmlCode);  
      }  
    }  
  ]);
```

Data Binding Best Practices

- You should always use the default binding mechanism
 - This will produce safe output, depending on the context
 - The framework is really good at this, so let it do its job
- If you need a safe set of HTML tags in the output
 - Use sanitization, either within the framework or from a library
 - Do not try to write this yourself
- Use the trusted HTML features **for static code only**

CSP and JS MVC Frameworks

- Default behavior of MVC frameworks is not CSP compatible
 - Dependent on *string-to-code* functionality
 - Requires *unsafe-eval* in CSP, which kind of misses the point
- However, frameworks are catching up quickly
 - EmberJS enables CSP by default when you create a new app

```
✘ 2015-04-23 13:14:10.245
  ▶ Refused to evaluate a string as JavaScript because 'unsafe-eval' is not an allowed source of script in the following Content Security Policy directive: "default-src 'self'". Note that 'script-src' was not explicitly set, so 'default-src' is used as a fallback. angular.js:956
✘ 2015-04-23 13:14:10.248 Refused to execute inline script because it violates the following Content Security Policy directive: "default-src 'self'". Either the 'unsafe-inline' keyword, a hash ('sha256-8t0yDUzWXHTRAW8ZleNf3KCXv-LC9-qIQhEogC01I7g='), or a nonce ('nonce-...') is required to enable inline execution. Note also that 'script-src' was not explicitly set, so 'default-src' is used as a fallback. localhost/:7
✘ 2015-04-23 13:14:10.295 Uncaught Error: [$injector:modulerr] Failed to instantiate module csrf due to: angular.js:4138
Error: [$injector:nomod] Module 'csrf' is not available! You either misspelled the module name or forgot to load it. If registering a module ensure that you specify the dependencies as the second argument.
http://errors.angularjs.org/1.3.15/$injector/nomod?p0=csrf
  at http://localhost:3000/bower_components/angular/angular.js:63:12
  at http://localhost:3000/bower_components/angular/angular.js:1774:17
  at ensure (http://localhost:3000/bower_components/angular/angular.js:1698:38)
  at module (http://localhost:3000/bower_components/angular/angular.js:1772:14)
  at http://localhost:3000/bower_components/angular/angular.js:4115:22
  at forEach (http://localhost:3000/bower_components/angular/angular.js:223:20)
```

EmberJS Enables CSP by Default

- Taken care of by *ember-cli-content-security-policy*
 - CSP policy can be updated through *environment.js*

UPDATING THE EMBERJS CSP POLICY

```
ENV.contentSecurityPolicyHeader = "Content-Security-Policy"  
ENV.contentSecurityPolicy = {  
  'default-src': "'none'",  
  'script-src': "'self' https://",  
  ...  
}
```


EmberJS Enables CSP by Default

EMBERJS DEFAULT CSP POLICY

```
Content-Security-Policy-Report-Only:  
  default-src 'none';  
  script-src 'self';  
  font-src 'self';  
  img-src 'self';  
  style-src 'self';  
  media-src 'self';  
  connect-src 'self' http://0.0.0.0:4200/csp-report;  
  report-uri http://0.0.0.0:4200/csp-report;
```

CSP and JS MVC Frameworks

- Default behavior of MVC frameworks is not CSP compatible
 - Dependent on *string-to-code* functionality
 - Requires *unsafe-eval* in CSP, which kind of misses the point
- However, frameworks are catching up quickly
 - EmberJS enables CSP by default when you create a new app
 - AngularJS offers a special CSP mode, making it compatible with CSP



CSP-COMPLIANT ANGULARJS

```
<html ng-app ng-csp> ... </html>
```

Enabling Dynamic Behavior with CSP

- So how does AngularJS process event handlers?
 - Parse 'ng'-attributes
 - Create anonymous functions, connected with events
 - Wait for event handler to fire

```
$element.onclick = function($event) {  
    $event['view']['alert']('1')  
}
```

- Technically, **not inline**, and no **eval()**

Do Not Underestimate XSS

- XSS is a vulnerability with serious consequences
 - If you get defaced, you got away easy
 - Look at Apache.org
- XSS is practically a certainty in a traditional Web application
 - Very hard to have systematic defenses
- Proper defense is context-sensitive output encoding
 - Use a well-vetted library to do get it done

Aim to Separate Data and Code

- JS frameworks are actually very successful in doing this
 - Allow them to fully shine as a client-side framework
 - Do not mix with server-side code, but use a clean REST API
- AngularJS and EmberJS eradicate developer-originated XSS
 - Unless you really want to shoot yourself in the foot
- Combined with CSP, they make a strong team
 - Get ahead of XSS attacks
 - Covers various vectors, including JS and CSS

Conclusion

Conclusion

- Single page applications are here to stay
 - Great user experience
 - The architecture empowers the client-side
 - Clear separation of concerns between client and server
 - Really awesome in the battle against XSS
- Security responsibilities have been reassigned
 - The server has little to no context anymore
 - **Authorization decisions can never leave the server!**
 - More data comes from the client, so less data can be trusted

Progressive Web Security course



**State-of-the-art
technologies**



**Hands-on labs
included**

- 1. Why simply deploying HTTPS will not get you an A+ grade**
- 2. How to avoid common pitfalls in authentication and authorization**
- 3. Why modern security technologies will eradicate XSS**
- 4. Four new browser communication mechanisms, and how they affect you**

3rd edition starts on April 12th 2016

<https://www.websec.be>

Securing Single Page Applications

Acknowledgements

Securing Single Page Applications

Philippe De Ryck

 philippe.deryck@cs.kuleuven.be

 @PhilippeDeRyck

 /in/philippederyck

 <https://www.websec.be>

